When regular expressions meet XDP

Ivan Koveshnikov
Gcore
Luxembourg
ivan.koveshnikov@gcorelabs.com

Sergey Nizovtsev
Tempesta Technologies Inc.
USA
sn@tempesta-tech.com

Abstract

A key to an effective way to mitigate DDoS attacks – is to know the protocol, that is going to be protected. Effective packet parsers allow discarding garbage traffic at high speeds. Understanding protocol state machines allows building stateful filters, that can spot and block malicious activity. However, such an approach requires a lot of programming work, especially if the DDoS protection system must be able to quickly adopt new protocols.

In such cases filtering by regular expressions helps to deliver coarse packet filtering by payload content. Extremely flexible, regular expressions allow to completely skip programming work and define packet filters by an end user.

Evaluation of regular expressions at network speeds is usually done in Deep Packet Inspection software, which is mostly a transparent appliance installed somewhere on the packet path. Being transparent DPI solutions doesn't need a real network stack for packet processing, allowing to offload regular expressions to the user-space network stack.

While building a rich filtering engine capable to work on the same servers that do provide services we concluded, that offloading regular expressions to user space is not as flexible as we need. In this article and talk we will show, how regular expression filtering can be done in the XDP context, what is the performance of the resulting solution, and how it affects other parts of network processing. We will also explain our motivations and use for the community.

Improving network scalability with XDP

How DDoS trends affect network design

Although DDoS attacks may have different natures and can be not only L3 flooding, but also sophisticated L7 layer attacks, specially designed for the target application, — all of them have common trends. We have noticed that a common DDoS attack is doubled every year in terms of capacity. If average attacks to our networks in Q1-Q2 of 2021 were close to 300Gbps, in 2022 it came to almost 700Gbps [3][1].

In practice, it means that the capacity of the DDoS protection system in our network must be scaled at least twice annually. But at the same time, an average attack duration is not very high, and it leads to the opposite situation. Most of the time the DDoS protection system is not under attack, processes much less amount of traffic and needs fewer resources.

Balancing between the capacity of the protection system and its power and cost efficiency, we had to change our network design, make it more flexible, and able to handle higher loads.

Limitations of centralized DDoS protection

In our traditional design (fig. 1) we use servers with a dedicated "DDoS protection" role. Filtering malicious traffic, the servers remain transparent for the rest of the network. Services, hidden behind such protection, may act as there is no presence of an attack on levels lower than L7. DDoS protection in this approach is centralized and works stand-alone.

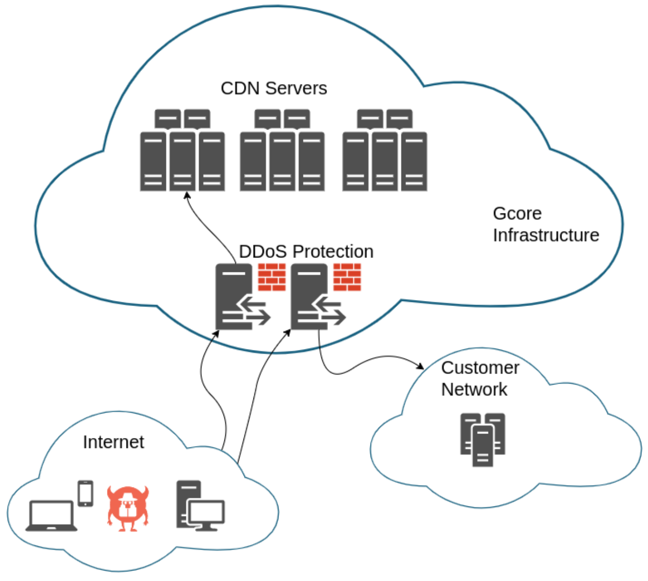


Figure 1: Centralized DDoS protection layer

A transparent filtering node is a well-studied approach that has a lot of benefits. First, the kernel-bypass technique can be a base for a fast packet processing engine, allowing it to process traffic with high throughput and low latency. Second, assigning dedicated CPU cores to specific tasks or applications provides performance guaranties on various loads, especially on regular expression evaluations – one of the most CPU-intensive loads. As a part of the traditional approach, we use a filtering solution provided by a 3rd-party vendor and developed on top of DPDK. It has all those advantages.

Of course, such a solution is not a silver bullet and has several drawbacks. User space networking engines usually prefer poll mode drivers for the immediate processing of network packets, but monitoring and predicting the performance of PMD drivers cannot be done directly, instead side subsystems like memory pools provide more performance insights. 3rd-party solutions are hard to extend and update, this limits flexibility and the ability to follow customer needs.

In addition to these common problems, our scalability requirements have brought us new ones. Every time a new CDN server is added to one of our locations, the server group used for DDoS protection also needs to be updated. The more servers you have the higher the probability that at least one of them is not in an operational state. DDoS protection servers are not an exception here.

Enabling and disabling DDoS protection on demand can help, but must be used with caution. Some countermeasures can be enabled during attacks without harming legitimate users, while others – not. Of course, several attacks may happen at the same time and an on-demand policy must take into account.

Distributed DDoS protection

To improve the scalability of our network, we had to rethink the centralized approach and move to distributed DDoS protection (fig. 2). In distributed design, the special 'DDoS protection' role disappears, and every CDN server starts to provide DDoS protection services.

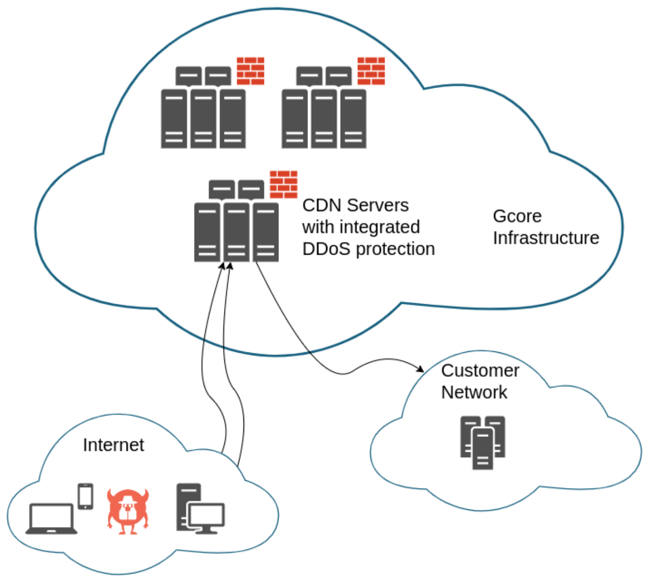


Figure 2: Distributed DDoS protection layer

In this scheme DDoS protection can be distributed across a bigger amount of servers. On the one hand, each protection node has more relaxed performance requirements to provide the same level of overall throughput. On the other, efficient implementation can lead to an even better overall result.

Moving DDoS protection closer to the protected application also allows efficient information exchange between the application and the protection system. E.g., if a web server detects an L7 attack, it can push additional rules to the locally running protection system almost immediately.

But the biggest change from the centralized approach – locality of some protected applications. If previously all the protected applications were deployed on some other servers, and DDoS protection could be implemented using any kind of technology stack, now some of them run on the same host. In our case, all of these applications have intensive network I/O and require efficient network stack.

DDoS protection must process packets delivered to every service and application, leading to the usual problem of coexisting user-space network stack and common applications with socket API. Especially if an application relies on advanced kernel features like `KTLS` or `MSG_ZEROCOPY`. Although various projects provide userspace network stack [2] or socket API [6], it complicates the overall solution significantly.

Because of that, we could not reuse the existing solution for mitigating DDoS attacks on CDN servers and had to switch to XDP for traffic filtering.

REX in the packet pipeline

Creating complex eBPF applications is not simple. Cycles and branches quickly raise complexity for the kernel verifier, and it becomes too difficult to add new features and pass all the sanity checks. The only way to create a complex eBPF program is to combine BPF-to-BPF and tail calls.

For every customer we use a different set of filtering rules and only a subset of available countermeasures. To achieve both flexibility of configuration and coping with problems of increasing complexity, we have split countermeasures into separate eBPF programs (fig. 3). For every customer we build a pipeline of tail calls that suit customer needs.

While some countermeasures are simple and can be easily implemented in XDP, others can be very complicated and require implementation inside the kernel with an eBPF helper function. The processing of regular expressions is a perfect example of such a countermeasure.

We need to filter packets by regular expressions for both allow and block lists. Some of our customers have a fixed protocol format, that can be validated using regular expressions. Only packets that match at least one of the defined expressions are passed to protected servers. In this mode a protected server receives only packets, that it can parse and identify.

Block list is the opposite mode, where the protection system blocks all the packets, that can be matched by the expressions. This mode is intended to react to security events when malicious traffic can be identified by some pattern inside a payload.

Because regular expressions are heavy operations, we try to use them closer to the end of the filtering pipeline, when all other countermeasures already have dropped whatever was possible. But such behaviour is not always possible, and the REX countermeasure can be not the last step in the pipeline.

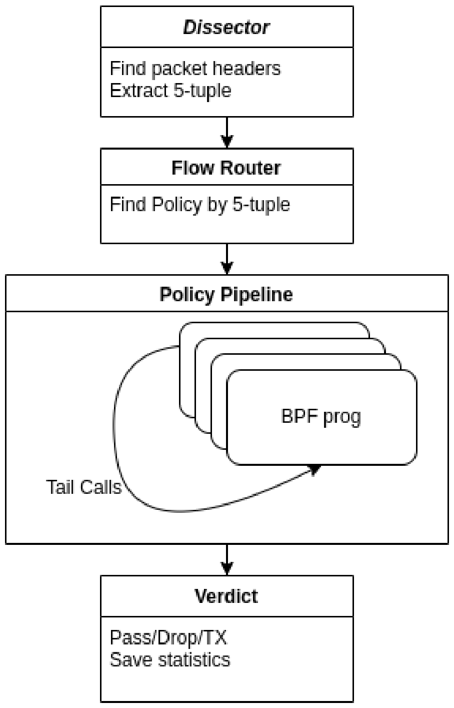


Figure 3: Distributed DDoS protection layer

Problems of REX in eBPF

Packet processing engine

The major problem with regular expressions – the complexity of the engine [7]. The only correct solution is to choose the fastest and the most suitable implementation and use it together with eBPF. The fastest and the most flexible implementation is DPDK-based Hyperscan. Alike all other implementations, it runs at the user-space level and cannot be called from XDP directly.

As noticed above, REX is not the final countermeasure, and we still may want either to push the passed packets into the kernel or to perform some other processing. Thus, integration of Hyperscan into XDP via `AF_XDP` sockets API was not possible, and providing an eBPF helper function was the only option.

The Linux kernel 5.16 has introduced an initial support of in-module eBPF helpers[4], and wrapping Hyperscan's runtime into a loadable kernel module becomes possible without the need of compiling it into the kernel itself. However, this work has been finished only in the 5.18 kernel, and it was not possible to register XDP helpers in prior versions. During our work, the latest release of the kernel was 5.17 and 5.18 haven't yet been released, and we had to apply a little patch to the 5.17 kernel to support this.

However, this wasn't the only problem that required kernel patching. One of the reasons, that makes Hyperscan so fast, – use of the vector instructions like AVX512 or AVX2. Although some kernel subsystems use vector instructions, they are avoided as much as possible because saving and restoring the FPU state on context switch is an expensive operation [5]. Two approaches are possible: save/restore FPU state per packet or during the whole NAPI process.

Per packet FPU state operations are easier to implement because they require no patching of the kernel, all the FPU-related operations will be isolated in the eBPF function helper. The per-packet approach also may be more beneficial, when only a small fraction of the traffic requires REX operations. On the other hand, NAPI-wide operations may be more effective if the majority of network packets require REX operations, but at the same time, they may harm latency, if no REX operations are required at all. The decision, on the approach to use, depends on target traffic scenarios. Later in this article, we are going to determine those scenarios and estimate performance penalties. Anyway, both approaches happen during the SoftIRQ context, while other kernel subsystems may use FPU in different contexts and don't expect that SoftIRQ may change FPU state, thus kernel functions `kernel_fpu_begin()/kernel_fpu_end()` must be paired with disabling of task preemption and SoftIRQs.

As the XDP program runs to completion on every packet, the budget for processing a single packet with regular expressions is very limited. Some regular expressions can be evaluated faster, some – slower, and some can get lead to heavily degraded performance on some traffic flows. Understanding budgets at runtime is very crucial, since going beyond the limits on traffic for one customer may lead to service degradation for other customers.

Hyperscan runtime requires a scratch area to work. This area needs to be allocated NUMA-aware, otherwise, significant performance degradation will happen. Per-CPU allocator in the kernel can allocate NUMA-aware scratch area, bit its allocation is limited to 32Kb, which can result in limited abilities of the runtime. For the moment we have not faced any problems with the scratch area size.

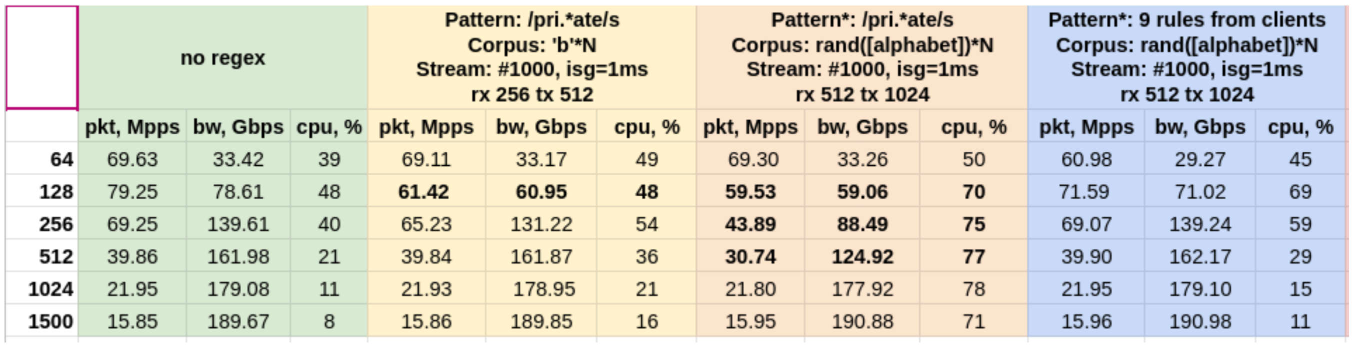
The eBPF verifier rejects calls with variable-sized buffer pointers. To get around this problem, we pass XDP context along with packet offsets to the module.

The easiest to solve issue is related to the eBPF code organization. When an eBPF program depends on a function from a loadable module, it cannot be loaded to the kernel if the module is not yet loaded. This requires extra work for conditional load of this eBPF program, when loading the rest of eBPF programs can be performed on systems without such a loadable module.

Processing in stream mode can be problematic in the XDP, since the XDP buffer lifetime is limited by the XDP program routine. In our use case, REX operations are applied only to UDP datagrams and only to packets that can fit public internet MTU, i.e., packets with a length lower than 1518 bytes, so we have not worked or even paid attention to this problem.

Configuration

Hyperscan run-time requires a database – a compiled set of patterns – to work. When we provide a configuration for a customer, we define a set of patterns to be evaluated for each traffic flow. The Hyperscan compiler processes the patterns and creates a database for the run-time. The loadable module needs to access this database somehow.



	no regex			Pattern: /pri.*ate/s Corpus: 'b'*N Stream: #1000, isg=1ms rx 256 tx 512			Pattern*: /pri.*ate/s Corpus: rand([alphabet])*N Stream: #1000, isg=1ms rx 512 tx 1024			Pattern*: 9 rules from clients Corpus: rand([alphabet])*N Stream: #1000, isg=1ms rx 512 tx 1024		
	pkt, Mpps	bw, Gbps	cpu, %	pkt, Mpps	bw, Gbps	cpu, %	pkt, Mpps	bw, Gbps	cpu, %	pkt, Mpps	bw, Gbps	cpu, %
64	69.63	33.42	39	69.11	33.17	49	69.30	33.26	50	60.98	29.27	45
128	79.25	78.61	48	**61.42**	**60.95**	**48**	**59.53**	**59.06**	**70**	71.59	71.02	69
256	69.25	139.61	40	65.23	131.22	54	**43.89**	**88.49**	**75**	69.07	139.24	59
512	39.86	161.98	21	39.84	161.87	36	**30.74**	**124.92**	**77**	39.90	162.17	29
1024	21.95	179.08	11	21.93	178.95	21	21.80	177.92	78	21.95	179.10	15
1500	15.85	189.67	8	15.86	189.85	16	15.95	190.88	71	15.96	190.98	11

Figure 4: Hyperscan benchmark results

The database lifetime is managed by the user-space process. There are many variants, of how this can be achieved, but the most suitable ones are `configfs` and eBPF maps. eBPF maps already have everything to provide necessary synchronization and to take care of the lifetime of each database. But the entry size of the eBPF map needs to be defined beforehand, which may lead to the inability to load a new database if its size can not fit into the existing table. Such situations can be only resolved by recompilation of all the users of such map. On the other hand, each entry, loaded into the `configfs`, is more flexible with its size, and such an issue is not related.

The `configfs` approach seemed for us more generic and suitable for any needs, so we decided to choose this way.

Benchmarks

TODO: more benchmarks and conclusions will come with the release version of the document. For now, refer to fig. 4.

Lessons learnt

The ability to add eBPF helpers in loadable modules adds remarkable abilities to implement very complex features, that can not fit into eBPF limitations. The only reason to pack the kernel was caused in our case by the requirement of running vector instructions during packet processing. Upstreaming this patch is doubtful though. The FPU state save/restore operations are expensive and will cause performance degradation for all other workloads except ours.

Running the REX in the XDP

This section is to be filled once the final benchmarking results will be achieved.

eBPF problems

Although we have succeeded in the development of a DDoS protection system on top of XDP, we have faced several problems.

First, eBPF has a lack of offloading support, while DPDK and the kernel itself get a lot of benefit from that. The XDP Hints initiative moving forward, and we expect a lot of advantages there.

Despite the XDP has become one of the standards in networking, it is still very loosely supported by the NIC vendors. There is a lot of documentation on DPDK, how to achieve the same level of performance as vendors do, and how to tune the system for the best performance. All that documentation is updated with every new version of DPDK or NIC. None of that documents exists for XDP. A developer has to either experiment on a production server or build a test lab to understand the correct settings. The lack of performance reference is frustrating.

References

[1] Comparitech. 20+ ddos attack statistics and facts for 2018-2022. `https://www.comparitech.com/blog/information-security/ddos-statistics-facts/`.

[2] F-stack github repository. `https://github.com/F-Stack/f-stack`.

[3] Gcore. Ddos attack trends in q1–q2 of 2022. `https://gcore.com/blog/ddos-attack-trends-in-q1q2-of-2022/`.

[4] bpf: Introduce bpf support for kernel module function calls. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2357672c54c3f748f675446f8eba8b0432b1e7e2`.

[5] Krizhanovsky, A. 2019. Fast http string processing algorithms. `https://tempesta-tech.com/research/http_str.pdf`.

[6] Mellanox libvma github repository. `https://github.com/Mellanox/libvma`.

[7] Wang, X.; Hong, Y.; Chang, H.; Park, K.; Langdale, G.; Hu, J.; and Zhu, H. 2019. Hyperscan: A fast multi-pattern regex matcher for modern cpus. *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*. `https://www.usenix.org/conference/nsdi19/presentation/wang-xiang`.