# Introduction to eBPF and XDP support in Suricata

By Éric Leblond, Peter Manev

White Paper

WP-eBF-XDP-092021-1

STAMVS NETWORKS

# CONTEXT

## Suricata

Suricata is a free, open source, mature, fast and robust network threat detection engine. The Suricata engine is capable of real time intrusion detection (IDS), inline intrusion prevention (IPS), network security monitoring (NSM) and offline pcap processing. Suricata inspects the network traffic using a powerful and extensive rules and signature language and has powerful Lua scripting support for detection of complex threats.

Suricata's fast paced, community driven, development focuses on security, usability and efficiency. The Suricata project and code is owned and supported by the Open Information Security Foundation (OISF), a non-profit foundation committed to ensuring Suricata's development and sustained success as an open source project.

## Stamus Networks

Stamus Networks helps enterprise security teams know more, respond sooner and mitigate their risk with insights gathered from cloud and on-premise network activity. The Stamus Network Detection (Stamus ND) and Stamus Network Detection and Response (NDR) are commercial enterprise-scale solutions developed and supported by Stamus Networks. Stamus ND/NDR are advanced network detection and response systems that expose serious and imminent threats to critical assets and empower rapid response.

**Stamus ND** is a Suricata-based intrusion detection (IDS) and network security monitoring (NSM) system, that delivers:

- Correlated IDS (signature-based) and NSM (protocol transaction logs) data
- Open interfaces for SIEM
- Turn-key Splunk app
- Support for third-party signatures and threat intelligence
- Tagging & classification for automated triage and alert reduction
- Integrated guided threat hunting

**Stamus NDR** is a broad-spectrum, open network detection and response (NDR) system built on top of Stamus ND that adds:

- Declarations of Compromise™ - response-ready threat detection from machine learning, stateful logic, and signatures
- Asset-oriented attack insights with kill chain timeline
- Open interfaces for SOAR, SIEM, XDR, IR
- Includes Stamus threat intelligence and custom threat detection
- Explainable and transparent results with evidence

## INTRODUCTION to eBPF and XDP

eBPF stands for extended Berkeley Packet Filter but you probably already knew that.  The old BPF system is used to filter packets on raw sockets and it has been extended to increase its area of usage. It is indeed now possible to plug an eBPF filter in various places of the Linux kernel to extract information or act on kernel behavior (See Brendan Gregg's page for more information). eBPF renders this possible by adding multiple kernel and userspace exchange methods. The main one is called maps. A map is a data structure shared between the kernel and the userspace. Both the kernel and userspace can access the map making this a powerful way to exchange information. Another capability of the eBPF filter is action. Depending on where it is run, it can tell the kernel to accept a system call or data transfer.

As you can guess, if the kernel can act based on eBPF filter content, it means it can be programmed in some way. eBPF is in fact a pseudo machine with a custom set of instructions. As it is not convenient to develop in machine language, a subset of C can be used and the Clang compiler (and soon gcc) has a special mode that can be used to generate eBPF bytecode from C code. In terms of code, the kernel passes various data as a parameter (depending on the hook) to a function and the function implements the wanted algorithm using, and updating maps if needed, and returns a decision.

eBPF and XDP support was added to Suricata 4.1 thanks to work by your servitor (Eric Leblond) and contributions from Jesper Dangaard Brouer (RedHat Principal Engineer).  This work has been completed for Suricata 5.0 with the addition of some powerful features.

There are 3 ways eBPF can be used in Suricata. In all of them, the eBPF filter can access the packet data and parse them to extract information. The first way is eBPF load balancing. The eBPF filter contains a load balancing function that will be used to dispatch a packet coming on an interface between all the sockets listening on that interface (fanout). The second way is eBPF filtering. Instead of using a traditional Berkeley Packet Filter, the kernel will use an eBPF filter instead. This offers an interesting perspective as maps can be used to introduce dynamic components into the filtering. We will detail that later. The third usage is XDP for eXtreme Data Path. In XDP, a eBPF filter is attached to a network interface and it is run against all received datagrams. The main point here is that it is done before the regular networking that will not be reached if the eBPF filter asks for a drop.

In Suricata, the main use case is currently to implement a bypass on a Linux raw socket but some other meaningful use cases can also be implemented.

## PACKET FILTERING and eBP

### eBPF socket filtering

In this case, the eBPF filter is attached to the raw socket and is deciding what packets need to be dropped or accepted. The flexibility of eBPF is one of the key points. Programmatic access to the packet data allows any single pass analysis to be done. Things that were impossible with traditional BPF can be done easily via eBPF with some C code. One example provided in Suricata is filtering following a list of VLANs; that was impossible via BPF.

eBPF filtering really shines by leveraging maps. A typical map is a hash table that contains a set of keys and values. Length in bytes of the keys and values have to be unique in a map and is defined at declaration. This is really flexible and allows arbitrary keys and values to be used. Maps can be updated from the eBPF code making it possible to build persistence in the implemented algorithm. Additionally, the userspace can also dynamically update the maps, making it possible to completely change the behavior of the filter. One example, with Suricata, is a filter that blocks all traffic coming from IPv4 addresses stored in the maps. By using a Linux kernel feature, named pinned maps, it is possible to access the map content via an external tool. It is thus possible to manage the list of IP address to drop via an external tool like bpfctrl developed and made public by Stamus Networks.

This is just an example, some more interesting features can be implemented, such as maintaining a list of services to monitor (IPv4:protocol:port). This could be used to monitor a farm of containers on a subset of the provided services.

### XDP filtering

With XDP filtering, we enter a different dimension. The eBPF code runs in various places depending on the configuration. If software is just an emulation, the performance advantages come from driver and hardware modes. In the case of driver mode, the filtering is done inside the driver code before the Linux networking stack. An early drop is thus really performant. In the case of hardware, the eBPF code is run inside the network card. This is currently only supported by a few cards including Netronome.

All the techniques described in the previous section are still valid, the only real change is the efficiency with which packets are dropped. Primary this process is done before any heavy treatments are performed on the packet. Suricata 5.0 supports almost all XDP features including hardware mode.

eBPF and XDP are greatly extending the capabilities of packet filtering but it requires a certain degree of C coding skills.  Hopefully some use cases will be directly implemented in Suricata source code to enable some of the benefits from this technology without having to invest time and effort into C coding.

## SURICATA FLOW BYPASS with eBPF and XDP

### Motivations and concept

Let's start this subject with something pretty self-evident, Suricata is CPU intensive. Tasks such as analyzing data on 30,000 different signatures is a highly consuming task despite traditional optimization methods.  So, in most cases, the bottleneck is the CPU computation. By consequence, one core is only able to handle a limited bandwidth, traditional numbers are around 250mbps to 500mbps per core.  Furthermore, as part of its work, Suricata is reconstructing flow as seen by the hosts. This has an implication with respect to threading. A packet treatment thread must handle all the packets of a single flow to avoid out of order packets that would cause the reconstruction to be invalid. If you combine the two elements, you come to the conclusion that Suricata cannot properly handle a flow that is faster than the cores supported bandwidth (between 250mbps and 500mbps). We named these flows, elephant flows.  On top of that, the elephant flows are also breaking other flows because it causes overflow in various ring buffers linked to packet acquisition.

To limit the impact of elephant flows, Eric Leblond and the Stamus Networks team, implemented a flow bypass (See Stamus Networks blog post announcing the feature).  The concept is to speed up the process for packets locally from bypassed flows or to collaborate with the capture method to bypass certain packets.

In the case of Linux raw socket, eBPF maps are the technology that can be used to maintain a table containing the flows to be bypassed.  As filtering can be done on a socket (via eBPF filter) or on the interface via XDP, there are two different mechanisms even if they share some part of the implementation such as maps management.

### Bypass strategy

Suricata comes with a set of bypass trigger methods. One consists of bypassing as soon as the stream reassembly depth has been reached (which means when Suricata is stopping full inspection of a stream).  The second method is bypassing encrypted packets.  Suricata parses the beginning of the TLS session doing a handshake analysis, once the session switches to encrypted it triggers a bypass.

One other method is based on the `bypass` keyword. A rule can be written to trigger a bypass when it matches allowing greater flexibility, for instance, only bypassing for certain websites or for services such as a backup service or Netflix. In the later case, it is possible to use Suricata traffic ID ruleset to bypass all flows coming from a service like Netflix.

### eBPF bypass

Initial implementation of eBPF bypass was done via socket filtering that occurred after the network stack handles the packet. There isn't much to be said about this method. It works but it is not optimal as the filtering is done in a late step as shown on the following figure:

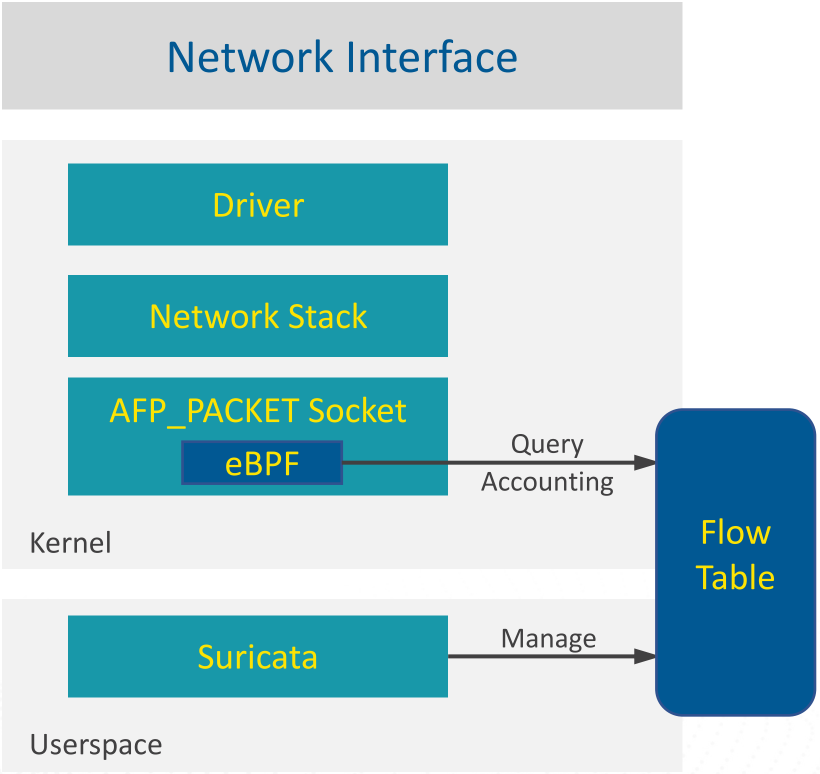

**Figure 1.** eBPF bypass in Suricata

The kernel has already done a complete treatment of the packet creating internal data structure. This single step is costly at high rate and can even be a bottleneck. Thus performance could be improved by doing the filtering at an earlier stage of packet ingestion inside the kernel.

### XDP bypass

This is what is done via XDP bypass as shown on the following figure:

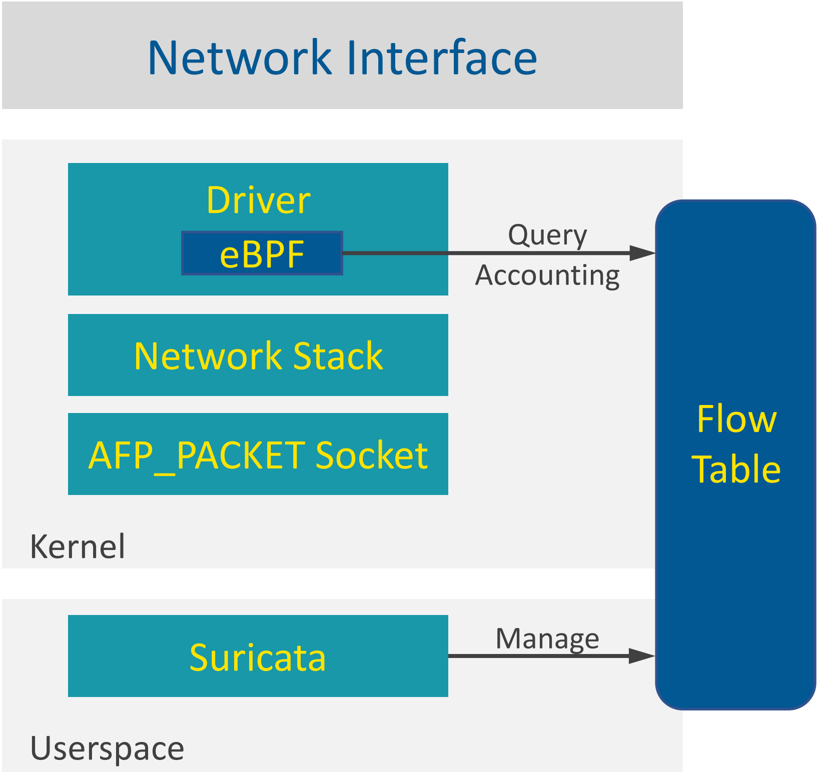

**Figure 2.** XDP bypass in Suricata

If socket mode is not interesting, the driver and moreover the hardware bypass are really interesting.   Driver mode is supported for all network cards with XDP support (this includes Intel, Mellanox, Netronome). The only cards supporting the bypass code in hardware mode are Netronome cards.  Getting it to work requires a custom eBPF filter but it can be obtained via the code provided in Suricata 5.0 and beyond via the define settings in eBPF source code.

## XDP bypass and IPS mode

Suricata can create a software bridge between two interfaces.

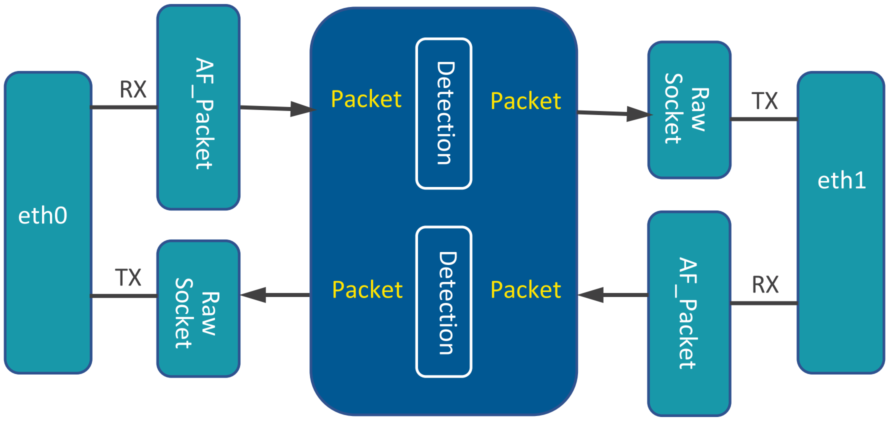

**Figure 3.** Suricata IPS mode in AF_PACKET

This method is performant and often used to provide high speed IPS capabilities. As dropping a packet in IPS mode has much more impact than in IDS mode, getting the help of the bypass technology seems really tempting but there is a problem. Suricata is doing the copy from one interface to another so as eBPF is dropping the packet this will not work.

XDP supports a special decision to send a packet to another interface. By updating the eBPF code to forward instead of dropping, we manage to have bypass in XDP mode. What is even more interesting is that we get super high-speed forwarding that bypasses the kernel and realize, in fact, a direct transfer from one network card to another card.

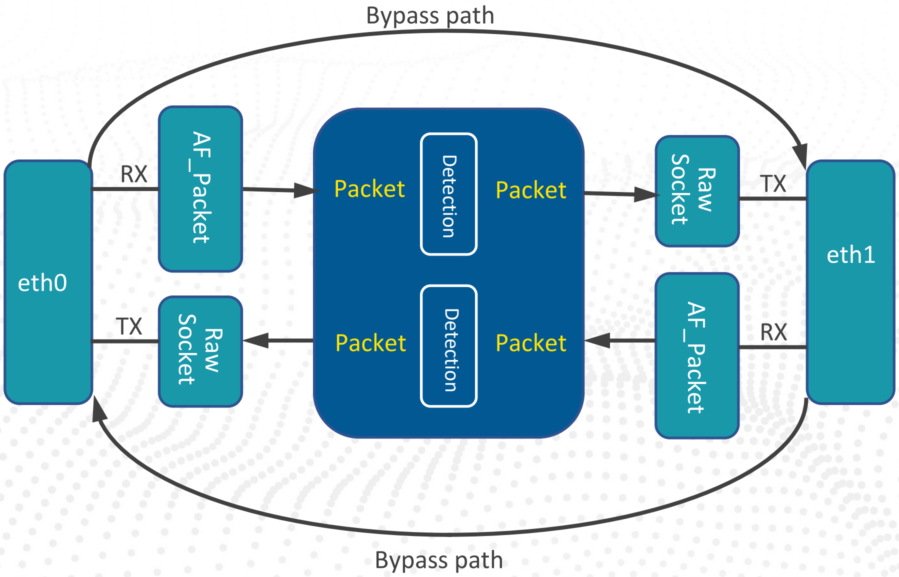

**Figure 4.** Suricata IPS mode with XDP Bypass

One of the problems of Suricata in AF_PACKET IPS mode is that without running Suricata the bridge is not active anymore and traffic is blocked. Thus, stopping or even restarting Suricata will result in a service interruption. To fix this issue, a global switch in the XDP filter can be activated via a pinned map.  When the value in the map is not zero, the XDP filter transfers all packets from card to card.  When the value is 0, the packets are sent to Suricata if they are not part of a bypassed flow. A tool like bpfctrl can be used to change the value in the global switch map allowing a watchdog process to limit the impact of a failing Suricata.

## ADVANCED XDP USAGES in SURICATA

### Pinned maps and XDP bypass

When Suricata starts, it only sniffs the end of the flows already in progress. This means there are a bunch of incomplete flows that will not be fully understood by Suricata. This is something we want to avoid as this will cause some unnecessary work by Suricata. If we look at that problem from a bypass perspective, we clearly see that it would be useful for Suricata to have bypassed flows surviving a Suricata restart. This is in fact possible with XDP because the XDP filter is attached to the interface and does not disappear with Suricata process exit. The only problem is that we will end up with ghost flows if Suricata does not have a way to restore the flow table. It is currently done in Suricata 5.0 by using pinned maps and by dumping the flow tables at start to reconstruct the internal state inside Suricata.  Doing so, Suricata manages to keep the flow table upon restart and avoid getting really confused at start.

### Content bypass

One of the bypass methods relies on bypassing a TLS session as soon as it switches to encrypted mode. This allows Suricata to analyze the TLS handshake properties and to avoid seeing encrypted packets where not much can be done.  Furthermore, thanks to the counters available in the flow table, Suricata is not losing the accounting of the bypassed TLS sessions. This is quite convenient and at least for long flows, really efficient.

Efficiency is really lower for short TLS flows because they mostly fit into the socket ring buffer. A partial fix consists of programming a basic TLS parser in the eBPF code to detect encrypted messages and drop them in XDP.  This allows early drop and releases pressure on the ring buffer.  This method needs to be used with care as it offers a way to bypass by using a traffic pattern.  Limiting that to a list of trusted servers may be a nice idea.

### CPU redirect

Network cards can do flow load balancing based on a hash method. This mechanism is known as Receive Side Scaling (RSS) and is used to distribute the packets on different queues thus splitting the workload in multiple units. The issue here is that the flow load balancing designed in the network card is not properly structured for Suricata. This methodology uses asymmetric load balancing whereas Suricata requires symmetric load balancing to get all the packets of a single flow handled within a single thread. The result is that "NIC default" RSS load balancing cannot be used on most cards.

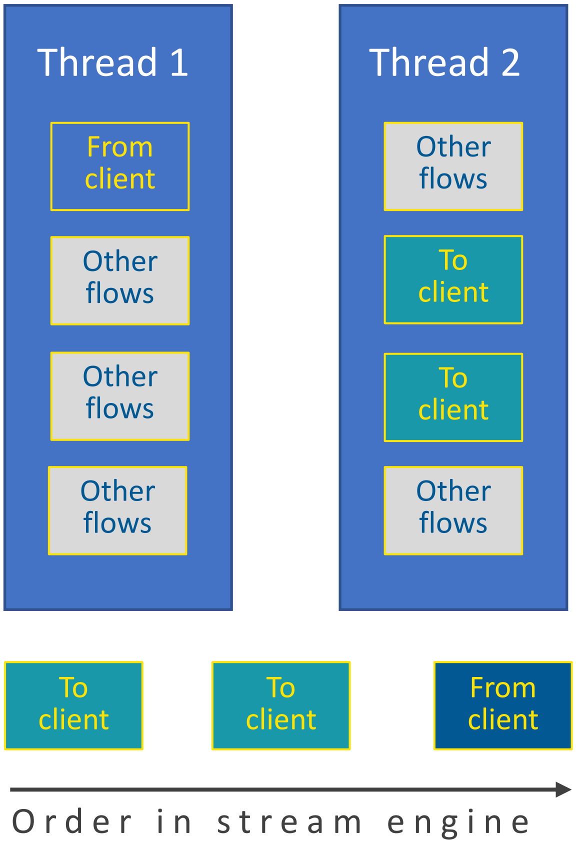

Figure 5. Problem of asymmetric hashing

By getting only one queue this means one single CPU will do the whole kernel work on the incoming packet. This will be a problem.

Jesper Dangaard Brouer came up with a solution for this problem by introducing a new feature in XDP named CPU redirect. By using this feature the eBPF code can distribute the raw datagram to selected CPUs that will handle the kernel tasks. Consequently, the CPU that receives the packet from the NIC is not the bottleneck anymore as it is now only dispatching the work to other cores.

### Netronome RSS load balancing

The Netronome card allows RSS load balancing to be programmatically done in the eBPF code. This is used in Suricata to realize an IP pair load balancing. The advantage of this load balancing is that it is fragmentation resistant as even IP fragmented packets will have the same hash. This avoids the issue seen with IP port load balancing where fragments do not have the same hash. The efficiency of the load balancing may be lower than the one with port but at least we don't end up with packets on wrong threads.

### Tunnel decapsulation

This feature is also related to load balancing. In some cases, the network is using tunnel like GRE or similar to transport traffic from a set of flows. This results in elephant flows that cannot be properly load balanced by Suricata.  XDP offers the capacity to shift the start of a packet. By doing that we manage to strip out the tunnel part in XDP and directly send to the kernel the inner packets.  The result is that load balancing is done on the inner flows and the elephant flow is not seen anymore.  This technique is currently implemented for the GRE protocol via a dedicated XDP filter, but other protocols could be implemented too.

## CONCLUSION

eBPF and XDP are a way to fix some problems associated with high performance. It is not the magical answer to every performance issue, and it requires some work.  It is, however, a key component to redesign the way traffic capture is built.  This area is evolving fast and the next big step will be the new AF_XDP capture method that provides a fast and efficient capture method. It is currently lacking some important features for Suricata capture but they should be added soon to the Linux kernel and this new capture method should be available in a coming release of Suricata.

Technical details and configuration information can be found in the eBPF and XDP page of the official Suricata documentation.

eBPF and XDP features present in Suricata 5.0 introduce interesting possibilities in terms of dynamic behavior and they will help Suricata to behave more correctly when under stress.  A lot of use cases are yet to reveal themselves; we hope this document will give you some ideas.

Don't hesitate to contact us at contact@stamus-networks.com if you wish for more information about the subject or want to discover Stamus Networks products.

## ABOUT THE AUTHORS

### Éric Leblond

Chief Technology Officer

Éric is an active member of the security and open source communities. He is a Netfilter Core Team member working mainly on communications between kernel and userland. He works on the development of Suricata, the open source IDS/IPS since 2009 and he is currently one of the Suricata core developers.

### Peter Manev

Chief Strategy Officer

Peter has 15 years of experience in the IT industry, including enterprise-level IT security practice. He is an adamant admirer and explorer of innovative open source security software. He is the Lead QA on the development of Suricata, the open source IDS/IPS.

## ABOUT STAMUS NETWORKS

Stamus Networks believes in a world where defenders are heroes, and a future where those they protect remain safe. As defenders face an onslaught of threats from well-funded adversaries, we relentlessly pursue solutions that make the defender's job easier and more impactful. A global provider of high-performance network-based threat detection and response systems, Stamus Networks helps enterprise security teams accelerate their response to critical threats with solutions that uncover serious and imminent risk from network activity. Our advanced network detection and response (NDR) solutions expose threats to critical assets and empower rapid response.

STAMVS NETWORKS

5 Avenue Ingres
75016 Paris
France

450 E 96th St. Suite 500
Indianapolis, IN 46240
United States

✉ contact@stamus-networks.com
www.stamus-networks.com