The rise of eBPF for non-intrusive performance monitoring

Cyril Cassagnes, Lucian Trestioreanu, Clement Joly and Radu State
University of Luxembourg
Interdisciplinary Centre for Security, Reliability and Trust (SNT)
Luxembourg, Luxembourg
{firstname.lastname}@uni.lu

Abstract—**In this paper, we explain that container engines are strengthening their isolation mechanisms. Therefore, non-intrusive monitoring becomes a must-have for the performance analysis of containerized user-space application in production environments. After a literature review and background of Linux subsystems and container isolation concepts, we present our lessons learned of using the extended Berkeley packet filter to monitor and profile performance. We carry out the profiling and tracing of several Interledger connectors using two full-fledged implementations of the Interledger protocol specifications.**

Index Terms—**Performance, Cloud computing, Interledger, eBPF, Profiling, Tracing**

I. Introduction

The requirement to monitor computer software at different levels of the software stack appeared seamlessly with the introduction of computers in industry. Monitoring helps and is the only way to diagnose different types of problems or anomalies [1]. Nonetheless, this activity is not a new requirement or a new problem. Cloud service providers require to monitor further and at scale what is happening in their data center [2]. What has changed in the last few years is that new tools at the operating system level and new tools to package and deploy software appeared (e.g. micro-services). This has been made possible thanks to Linux kernel evolution (e.g. namespace, cgroups) [3]. However, the non-intrusive management of containers relies on the observability of the underlying operating system. For this reason, we explore the observability capability offered by the extended Berkeley Packet Filter (eBPF) to the Linux kernel. To do this, we analyze the potential of eBPF-based tools, which offer unique capability. Beyond networking functions, eBPF instruction set allows for monitoring in-kernel IO subsystems, i.e. tracing, analytics, and security functions. We specifically assess the added-value of the extended Berkeley packet filter tracing framework. Our contribution is the experimental study of eBPF-based tools in the context of non-intrusive profiling and diagnoses of a user-space application in production. Indeed, all parties hosting and/or operating an application cannot rely on predefined metrics externalize by developers. Moreover, to find root causes after an incident[1], cloud users may also want to monitor the internals of the system.

[1]https://landing.google.com/sre/sre-book/chapters/monitoring-distributed-systems, valid in August 2019

Therefore, in this paper, we address the challenge of monitoring a new generation of user-space application at the deepest level without any support from the application.

The structure of the paper is as follows. The Section II discusses the state of the art and the capability of eBPF for networking and beyond. Section III presents the eBPF tracing tools against process isolation mechanisms. Section IV presents and analyzes information extracted for Interledger Connectors. This Section explains also the rationale behind the design of our performance tool-chain. Section V discusses challenges, future works and concludes.

II. Background and Related Work

In the last few years, BPF has evolved with the extension of its instruction set. BPF becomes the extended Berkeley Packet Filter (eBPF), which is the Linux subsystem that allows safely executing untrusted user-defined eBPF programs inside the kernel [4]. Since, eBPF is completely redefining the scope of usage and interaction with the kernel. It offers the possibility to instrument most parts of the kernel. eBPF adds the ability to inject code at specific tracepoints. This goes from network tracing to process or I/O monitoring like proposed in the I/O Visor project[2]. A key added-value of eBPF is that it does not require a new kernel module in comparison to other tools (e.g. LTTng or SystemTap). eBPF is by default in the Linux kernel. The possible attachment points to collect metrics are: Kernel functions with kprobes, Userspace functions with uprobe, System calls with seccomp, and tracepoints. Thanks to SECure COMPuting with filters (seccomp), eBPF changes also the way we can perform security monitoring in the system. Like explained in the Linux kernel documentation, seccomp provides a means for a process to specify a filter for incoming system calls. The filter is expressed as a BPF program. For instance in [5], the authors use it to spot malicious system activities. Instead of using packets as traditional applications to intercept communications, they monitor network activities by exploiting trace points and Linux kernel probes. In this paper, we explore these capabilities to trace and profile user-land applications.

Because of its origin, eBPF is already used heavily for networking subsystems [6]. In this context, eBPF is used in

[2]www.iovisor.org/, valid in August 2019

conjunction with the eXpress data path (XDP) framework. XDP has been created to process packets before they enter the kernel, unlike the traffic control subsystem (TC). TC operates in the network stack, which means that the packet has been already processed by the kernel. Indeed, after a packet passes the XDP layer, the kernel allocates a buffer to parse the packet and store metadata about it. Then, to access the packet, the eBPF program uses a pointer to *sk_buff*, and not *xdp_buff*, which is not in the kernel. Nonetheless, the primary goal of these frameworks is to perform efficient switching, traffic classification [7], virtualized networks [8], [9], routing, traffic generation or communication optimization [10]. For instance, in [11] the main technical contribution of the authors is to show how this nascent technology can be used to not only build in-kernel programmable VNFs but also how to interconnect them on a single system. Indeed, the Linux kernel limits the size of eBPF programs to 4096 instructions or 32 kbytes, i.e. 4096 instructions of 64 bits. To get around this instruction limit imposed by Linux for security, Tail-calls are possible. This means that one program triggers another filter/program to create a chain of eBPF programs. In this area, Polycube is a framework that enables the creation and deployment of arbitrary virtual network function[3].

Another new feature offered by this tool allows the authors of [12] to deal with the increase in encrypted traffic. To gain access to the clear text payload transparently for end-to-end encrypted applications they propose to use the recently merged kernel TLS functionality. This method does not require to decrypt and re-encrypt any data and reduces the overhead and latency. However eBPF/XDP is not the only system enabling programmable packet processing [13]. The Data Plane Development Kit (DPDK) is the main alternative to use accelerators (hardware or software) to manage transport protocols of the future and minimize the impact of the network service chain [14]. XDP takes an approach that is the opposite of kernel bypass. Next, like explained by the authors of [15], network traffic monitoring is traditionally based on packet analysis. While this approach is still useful in many contexts, it does not solve our problem to provide detailed visibility when containers or virtual systems are used. Like they suggest in their work, a merge between system and network monitoring is required. That is why in this paper we explore the limit of eBPF to monitor system-wise a containerized user-space application. Indeed, most of the research work on eBPF is related to the primary goal of the tool, i.e. packet processing. Therefore, talking of eBPF without talking about packet processing was until recently not relevant. However, eBPF is now valuable for security monitoring, network filtering but also for program tracing, profiling, and debugging.

With regards to the opportunity for Cloud infrastructures, today, Cloud data centres with providers at physical level and customers at virtual level both monitor their hard and software infrastructure to understand load patterns and to detect malfunctions [16] and bottlenecks. The motivation for cloud monitoring on both the virtual and physical levels can be summarized to the three following opportunities: alerting, resource allocation, and visualization [17]. The added-value of eBPF is performing those tasks for applications in production environment, even when micro-services are used. Indeed, in [18] the authors explain how micro-services are challenging the classical methodology and performance tools. Unlike traditional client-server applications, resolving performance issues requires to determine which micro-service is the root cause of performance degradation. The major contribution of the authors is their benchmarks, which unfortunately does not consider payment or micro-payment infrastructure, like Interledger [19].

A. *Profiling and Tracing tools*

The recommended front-end for using BPF tracing framework is the BPF Compiler Collection (BCC). BCC was created by Brenden Blanco in April 2015 but the work of Brendan D. Gregg is now more than influential [20]. Performance tools allow performing two major tasks:

- Tracing, to report when events occur over time.
- Profiling, to report the number of occurrences of each type of event tracked.

There is a broad list of ways to perform Tracing in Linux [21]. From the original mainline Linux tracer, Ftrace, to profiling tools like perf. Over the years, more complex customized tracing tools appeared in the kernel, like BPF, eBPF, and out of tree tracers like LTTng, systemtap and Dtrace [22]. The most intrusive method uses only static instrumentation: tracepoints and user application statically defined tracepoints (USDT). However, this has several implications. First, this required access to the source code to add the user markers (USDT). Then, in any case, it requires recompilation to activate pre-defined marker. Second, user markers will have to be maintained over time based on the evolution of the code based. However, eBPF opens a new world of possibilities with dynamic instrumentation: kernel dynamic tracing (kprobes), user-level dynamic tracing (uprobes). For instance, in Java the dynamic instrumentation known as dynamic tracing lets you trace any software function in a running binary without restarting it. Nonetheless, one downside of this approach is the continuity of service over time because instrumented functions can be removed or renamed depending of the maturity of the software interface observed.

eBPF being introduced, we present now the mainstream front-ends for using it, which have been disseminated in a conference organized by the Linux foundation [23]. Moreover, we specifically used them in the context of our experimentation.

First, the **BPF Compiler Collection** provides a BPF library and Python, C++, and Lua interfaces for writing programs. BCC[4] front-end is a toolkit great for complex tools with a bound scope or for agents. In this respect, BCC provides a large set of predefined tools. Brendan just released most of his work in a book [24] where he presents all these tools.

[3]https://github.com/polycube-network/polycube, valid in Sept. 2019

[4]https://github.com/iovisor/bcc, valid in August 2019

An interesting fact is the audience targeted by the book, i.e. system administrators and reliability engineers. This confirms the relevance of the tools for cloud providers or in-house critical IT infrastructure. BCC requires a program built in three parts:

1) Importing the dependencies to use the eBPF framework.
2) The eBPF program always written in C-like language and stored in a variable.
3) The processing script allowing to load, execute and retrieve data from the eBPF program injected in the kernel.

Second, the *Bpftrace* for quick ad hoc instrumentation (e.g. for detecting zero-day vulnerabilities). Bpftrace[5] is suited for decomposing metrics into distributions or per-event logs to create new metrics. Ultimately, the tool helps to uncover blind spots. Bpftrace was created by Alastair Robertson. It uses LLVM as a backend to compile scripts to BPF-bytecode and makes use of BCC for interacting with the Linux BPF system, as well as existing Linux tracing capabilities: kprobes, uprobes, and tracepoints. The bpftrace language is inspired by awk and C, and predecessor tracers such as DTrace and SystemTap. Bottom line, Bpftrace is best for short scripts and ad-hoc investigations.

Third, the *Performance Co-Pilot* provides a range of services that are used to monitor and manage system performance. PCP[6] is a system-level suite of tools for performance analysis. It has been field-tested in RedHat distributions and provides a framework. PCP uses a daemon and relies on PMDA (Performance Metric Domain Agent) to collect any metrics.

eBPF collects the data in the kernel and transfers it in the user-space thanks to BPF Maps. This is a key/value store inside the kernel. This allows to collect metrics at each iteration of the eBPF program. A single BPF program can currently access up to 64 different maps directly. eBPF is changing drastically the capability of the Linux kernel [25], [26].

III. NON-INTRUSIVE MONITORING AND PROFILING

Today performance monitoring for Cloud service providers is complex; while they do not have control of the application layer, they must provide the technical means to align with new regulations such as data protection in Europe. Moreover, containers have become a commonly-used method for deploying services on many operating systems, providing short application start-up times, resource control, and ease of deployment. Consequently, Cloud service providers have to deal with Continuous integration (CI) and continuous delivery (CD), application elasticity, and distributed data processing workloads. In this Section, we explain the complexity of undertaking performance evaluations for containerized user-space applications even when using Linux based profiling and tracing tools. We explain the rationale behind the aim of performing non-intrusive performance monitoring and we demonstrate the added value of eBPF-based tools to perform non-intrusive performance monitoring of containerized user-space applications.

[5]https://github.com/iovisor/bpftrace, valid in August 2019

[6]https://pcp.io, valid in August 2019

A. Container isolation

The isolation boundaries of a container can vary. In the case of Linux, namespaces and cgroups are used to set those boundaries. The first meaning that we attach to the principle of isolation implies protecting each process from other processes within the operating system. In consequence, this segregate the memory space of process A from the memory space of process B. To enforce this Docker container and other container technologies use a collective noun for a group of isolation and resource control primitives. For instance, the Google gVisor, the IBM Nabla secure container systems, and others propose a promising approach to containment [27]. The authors propose X-Containers as a new security paradigm for isolating single-concerned cloud-native containers. They explain that replacing some of the isolation primitives with local system call emulation sandboxes is not enough guarantee.

In this context, where isolation of software components is strengthening, we advocate for non intrusive performance monitoring with in-kernel facilities. Security and performance are the primary concern but gathering indicators for performance or to guarantee security becomes a challenge. On the one hand, user level tools are required to monitor the full stack of their application in a outsourced environment and, on the other hand, more advance sysadmin tools are required to monitor black-box containers. Indeed, it is not clear yet how dynamic instrumentation will be able to safely bore in the containment boundaries where the business code is. The added-value of non-intrusive monitoring has been proven in many other works [28], [29], [30], [31], [32] and [2]. For instance, Cilium[7] leverages eBPF in a cloud native and micro-services context. However, Kubectl-trace is a Kubernetes command line front end for running bpftrace across worker machines in Kubernetes cluster [33]. While at this stage and to the best of our knowledge there is no solution yet to activate metric capturing during the deployment of a containerized application with Kubernetes or equivalent.

B. eBPF integration

Our goal is to dissect the behavior of Interledger connectors. This means that at minimum, we want to perform the full scope of classic system resources monitoring, e.g. CPU, memory, TCP connections. Then, two other requirements are the possibility to filter traffic at the XDP level and the possibility to filter the metrics gathered per container. For this, we use the PID or a port number for a container running a specific network service. The key features that we consider for our tool chain are the extensibility of the tools used and the possibility to store collected data. We want the ability to archive metrics to perform a posteriori analysis.

Like explained previously, BCC and bpftrace are meant to be used for the creation of higher level tools such as

[7]cilium.io, valid in August 2019

eBPF_exporter or Performance Co-Pilot (PCP). At first, we experiment with eBPF_exporter, open sourced by Cloudflare, to extract metrics and feed the main Prometheus server which scrapes and stores time series data. Prometheus is supported by the Linux foundation and relies on special-purpose exporters for services like the eBPF_exporter. Prometheus has a multi-dimensional data model with time series data identified by metric name and key/value pairs and so a query language to leverage, PromQL. Then, we were not satisfied with the module provided by the eBPF_exporter. Indeed, we did not have the possibility to monitor the garbage collector or any facility to add IP filtering. Between modifying the parser of the YAML files used by the exporter written in Golang and switching to PCP, we decide to switch to PCP. Another argument in favor of PCP is that it provides support for the creation and management of archive logs.

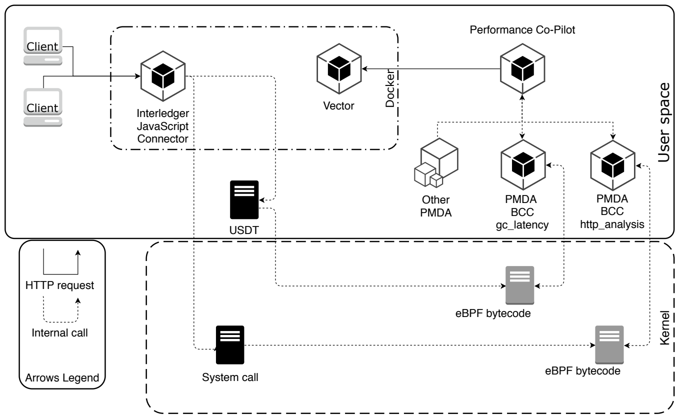

Fig. 1. Overview of the performance monitoring setup

Figure 1 shows the selected tool chain. PMDA modules are in charge of injecting eBPF bytecode in the Kernel. Then, PMDA modules gather and transmit collected metrics to a PCP web daemon. Next, a visualization layer is required. In this context, we try the following tool composition for the realization of our performance monitoring tool chain:

- eBPF Exporter: integration with Prometheus and Grafana.
- Vector and Performance Co-Pilot (PCP): for remote BPF monitoring.
- Grafana with PCP: for remote BPF monitoring.

Regarding data visualizations, we experiment with Grafana and Vector. Vector can ”visualize and analyze system and application-level metrics in near real-time”. These metrics include CPU, memory, disk and network, and application profiling using flamegraphs. Most important, Vector is built on top of PCP. Vector pulls performance data from the PCP web daemon, which in turn talks to the metric collector daemon. This model is lightweight as no data is stored across browser sessions, and there is no metrics aggregation across hosts.

Finally, in spite of our critics on static program instrumentation, we try user application statically defined tracepoints as shown in Figure 1. This approach is simple, easy to parse but completely intrusive and lacks features such as typed arguments. This is only suitable for the development and debugging phase.

IV. Profiling and Tracing of Interledger

As explained before, on-line monitoring is required in our case. The default toolkits do not fulfill our needs. That is why we have to create new eBPF programs, which will be integrated in our monitoring solution based on Vector and PCP, like presented Figure 1. We add dedicated modules for the purpose of monitoring the connector during its execution. For each module, parameters can be set through a common configuration file for the BCC PMDA of PCP. Therefore, we filter the monitoring to the connector itself, by setting the ports, IP address and PID of the network stream to be monitored. It is even possible in some cases to limit to a process selected by regular-expression matched name. Each module was systematically added in three steps. First, write the corresponding user-space program to inject and perform the measurement with the eBPF program. Second, gather and store data with PCP, by adding the corresponding module in the BCC PMDA. Third and last step, display the data in one or more widgets in Vector, to allow for on-line monitoring.

A. Interledger connector

The goal of Interledger is to provide an architecture and a minimal set of protocols to enable interoperability for any value transfer system. The Interledger protocol is literally a protocol for Interledger payments. Interledger Connectors aim to realize the vision of an international friction-less payments routing system. In other words, a standard for bridging diverse financial systems [19]. In this Section, we use two full-fledged implementations of the Interledger protocol. Each of them runs over the last long-term support version of nodejs (v10) in a Docker container. The reference implementation of the protocols is in github[8]. This is a JavaScript implementation of an Interledger connector. The second implementation of an Interledger connector is also in JavaScript and all the source code is on github[9].

The main business risk connectors face in Interledger is being unable to fulfill the incoming transfer after their outgoing transfer has been executed [34]. This is what they call the *Fulfillment Failure*. To diminish as much as possible the odds of that occurring, they propose some mitigation measures[10]. For two of them eBPF has a clear added-value:

- Packet filtering - White-listing or denial of service protection
- Redundant Instances - Difficulty to interfere with program instance(s)

Therefore, monitoring is a must in this case and in the case of an unexpected attack. Indeed, a worldwide payments routing system will certainly be a clear target.

[8]https://github.com/interledgerjs/ilp-connector, valid in Sept. 2019
[9]https://github.com/interledgerjs/rafiki, valid in Sept. 2019
[10]https://interledger.org/rfcs/0018-connector-risk-mitigations/

B. *Performance analysis and flamegraph analysis*

In this Subsection we profile both Interledger connectors to point out performance flaw. All the material and code produced in the context of the experimentation carried out for this paper are available on a github repository[11].

In [35], we explain how to setup an Interledger test-bed connected to a private RippleNet and Ethereum PoA. For this paper we used a simplified version of our test-bed to first generate traffic only through two connectors based on the reference implementation. This simplified test-bed is composed of a private RippleNet and three interconnected connectors, i.e. forming a triangle where one is only observer/idle. Then, using the same setup, we generate traffic through the Rafiki connector. To generate the workload, we trigger payments between two users connected to two different connectors and exchanging 50000 XRP at a rate of 1 XRP per ILP packet. In another work, we are currently evaluating the impact of the parameters used by Interledger to route and settle payments. This means that we create a proper payment channel and carry well formed ILP packet.

To perform the stack trace profiling, i.e. flamegraphs, we use a tool called *0x*[12]. The stack trace profiling can be resource intensive. Therefore, *0x* proposes a method to generate a flamegraph on a production server. By default 0x uses the internal profiler of the JavaScript engine (V8). This means native frames are omitted.

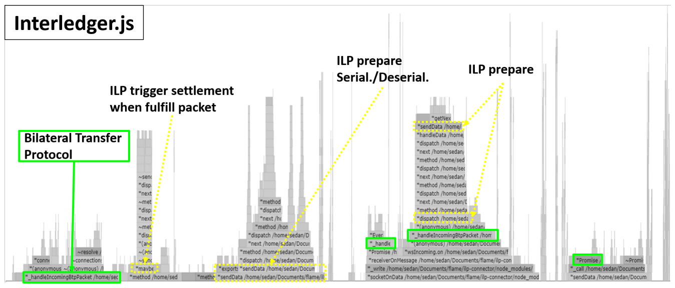

Fig. 2. Reference implementation at work - full flamegraph

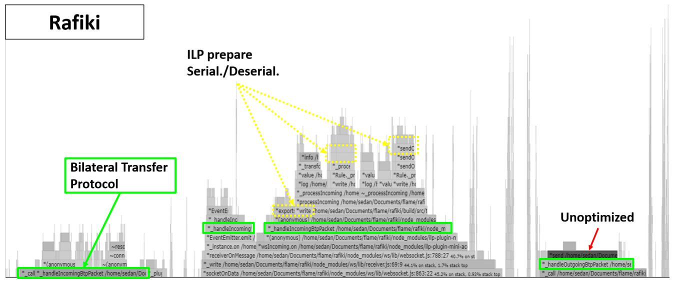

Fig. 3. Rafiki implementation at work - full flamegraph

Figure 2 and Figure 3 show respectively the flamegraph generated for the Reference implementation and for Rafiki, both under workload, i.e. sending ILP packets. We present these flamegraphs to emphasize first the differences in terms of behavior. Conceptually, the two connectors do the same thing. Indeed, both implementations are based on the same specification and the same JavaScript libraries. The main thing that distinguishes Rafiki is its software architecture. To compare the Interledger connectors, we also carry their stack trace profiling while in idle state, to have a baseline. Besides, in our Vector dashboard we are also able to monitor application specific metrics: garbage collector, TCP sessions lifetime, HTTP traffic (HTTP verbs, code), Websocket sessions, network throughput and other classic metrics. At this scale, Figure 2 and Figure 3 only allow us to point out major performance differences between the two implementations.

One of the major added-value of such flamegraphs is the interactive nature of the plot that let you drill-down and zoom on any function call. This is helpful when you do not know what you are looking for. That is why, we primarily used the flamegraph as a map to find out where Bilateral Transfer Protocol (BTP) and the Interledger Protocol v4 are active.

Based on these Figures, which are at this scale suitable for two things:

- Observe the general shape of the flamegraph. Knowing that functions are ordered alphabetically
- Observe the proportion at runtime of each function level, i.e. native, business, etc.

For the reference implementation we notice several prominent columns pointing out deep and long function calls at the application level. The software stack at issue is managing part of the ILP packets processing and part of the settlement process. Therefore, we investigated further how ILP packet are (de-)serialize and how the settlement process is triggered. It turns out that in Rafiki developers decouple the settlement engine from the packet processing logic. Therefore, results are consistent with their architectural modification. In Rafiki, the columns pattern disappears.

C. *New eBPF program created*

a) Monitoring the garbage collector: BCC provides an example to count the number of executions of the Garbage Collector (GC) and time them. This was however not integrated into PCP and Vector. Therefore, we have built on the example to create a PMDA module and a heatmap widget for Vector. The heatmap presents the lifetime and the frequency of all the call to the garbage collector.

b) HTTP traffic Identification: We get the packet in a raw form, quite close to the bytes circulating on the network. Even though some processing has been performed by the network card at this stage. For instance, on card TCP checksum validation leads to an incorrect value of the corresponding field when reading the packet at our stage. All the following process is performed in the kernel at TC level. Our goal was to show how far we can go in the packet analysis at this level to, for instance, filter packet as early as possible. Consequently, we process each network layer to locate the HTTP content. First, we read the ”type” field of the Ethernet layer header. If the value is the one associated with IP protocol, the packet is

[11]https://github.com/Oliryc/monobpf, valid in Sept. 2019
[12]https://github.com/davidmarkclements/0x, valid in Sept. 2019

candidate, i.e. it could be an HTTP packet and we process it further. The next header is then from the Internet protocol. We know the position of its first byte because the Ethernet layer is fixed-size. The ”nextp” field of the IP header is checked for correspondence with TCP. If so, the packet is still candidate. To know where the first byte of this TCP header is, we use the ”ihl” field of the IP header. Similarly, to know where the first byte of the HTTP header is, the ”doff” field of the TCP header can be used to know the length of this last header. Finally, we can check the packet for typical HTTP content, like methods (GET, POST). If this matches, we send the packet to the user-space program, where a similar task is performed to locate the HTTP payload. From this payload, features of the HTTP protocol can be extracted, like methods, headers, return codes. Note that as soon as the packet is not candidate anymore, the treatment is interrupted.

Header	How to know the size?
Ethernet	Fixed size in spec.
IP	Size inferred from the ”ihl” field
TCP	Size inferred from the ”doff” field
HTTP	Sequence-delimited size

TABLE I
PROTOCOL LAYERS TO DECAPSULATE MANUALLY TO LOCATE THE HTTP PAYLOAD.

c) IP Whitelisting and Denial of Service Attack: As a hardening measure, we leveraged eBPF and XDP to improve the resilience of the connector against denial of service (DoS) attacks. It has been observed in the industry that rejecting packets with iptables rules was not efficient enough to successfully handle medium-sized DoS attack. The reason is that once iptable decides to drop a packet based on one of its rules, it is quite late already. Some copy and processing already went through the kernel stack and situations where all the CPU time is used to merely drop the packets arise. To solve this, a new good practice is to rely on XDP. However, this security layer is not perfect but clearly improve the capability of the kernel. With XDP, we can decide to drop a packet right away, on the networking card, thus without entering the kernel stack. Indeed, this technique becomes even more relevant for the new generation of network cards.

A key point of XDP compared to iptable is that there are no costs associated to the re-injection of the packet into the kernel when we want to keep it. This is paramount to avoid slowing down legitimate packets during an attack. Nonetheless, since it is based on IP white-listing, the risk for DoS attacks still exists if an attacker success to find an IP in the white-list. Finally, the code of the eBPF program used for DoS protection is presented in Figure 4.

In this Section, we presented a part of our results. Indeed, to perform a precise analysis and diagnosis of the program under workload the cross-validation of all the different metrics collected is required. Our experimentation are twofold. We use eBPF to better understand Interledger connectors when treated as a black-box program. Then, we assess the potential of eBPF probes to monitor the full stack from operating system

```
#define WHITE4SIZE 6

static int ip4white[] = { 3137448128, 1644275904,
  ↪ 16885952, 2516691136, 2197924032,
  ↪ 1140959424};

int xdp_prog1(struct CTXTYPE *ctx) {
    nh_off = sizeof(*eth);

    if (data + nh_off  > data_end) {
        return rc;
        }
    h_proto = eth->h_proto;
    if (h_proto == htons(ETH_P_IP)) {
        // Allow packet to pass if its IP is in
          ↪ the whitelist
        int ip = get_ipv4(data, nh_off,
          ↪ data_end);
        if (ip == NOIP) {
          return XDP_DROP;
        }
        #pragma unroll
        for (int i = 0; i < WHITE4SIZE; i++) {
          if (ip4white[i] == ip) {
            return XDP_PASS;
          }
        }
        return XDP_DROP;
    } else if (h_proto == htons(ETH_P_ARP)) {
      return XDP_PASS;
    } else {
      return XDP_DROP;
    }
}
```

Fig. 4. Code snippet to prevent DDOS with XDP

to application layer when the application is containerized and not modifiable.

V. CONCLUSIONS

eBPF is a major change in the Linux kernel, which impacts a large audience of users. Indeed, as researchers, we are always interested to perform precise measurements for our experimentation and this can be achieved with a better observability of the kernel. For system engineers, this offers the possibility to better point out the applications’ critical behavior, and for administrators this is the opportunity to better secure infrastructure and profile third-party applications.

In this paper, we realized experiments with a Linux subsystem to monitor containerized user-space applications. We performed these experiments in the context of our work with Interledger connectors where the capability to monitor the software stack in production is a must-have. With this work we explored and assessed the tool landscape created to support eBPF. The contribution of this paper encompasses:

- The use of eBPF/XDP programs with one of the industry standard tools, Performance co-Pilot
- Experimentation of the tools on Interledger connectors

In our future work, we will assess how this type of tools can create an end-to-end view of a distributed system. Nowadays, in complex n-tier architecture the challenge is to trace a request all along its journey.

Acknowledgment

We thankfully acknowledge the support from the RIPPLE University Blockchain Research Initiative (UBRI) for our research.

References

[1] B. Beyer and R. Ewaschuk, *Monitoring Distributed Systems*, O'Reilly, Ed. O'Reilly Media, Inc., 2016.

[2] G. Liu and T. Wood, "Cloud-scale application performance monitoring with sdn and nfv," in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, March 2015, pp. 440–445.

[3] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[4] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv, "Simple and precise static analysis of untrusted linux kernel extensions," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: ACM, 2019, pp. 1069–1084.

[5] L. Deri, S. Sabella, and S. Mainardi, "Combining system visibility and security using ebpf," in *Proceedings of the Third Italian Conference on Cyber Security (ITASEC)*, ser. ITASEC'19, vol. Vol-2315, 2019, pp. 50–62.

[6] Cilium Authors community, "BPF and XDP Reference Guide," https://docs.cilium.io/en/v1.6/bpf.

[7] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, "Performance implications of packet filtering with linux ebpf," in *2018 30th International Teletraffic Congress (ITC 30)*, vol. 01, Sep. 2018, pp. 209–217.

[8] T. Nam and J. Kim, "Open-source io visor ebpf-based packet tracing on multiple network interfaces of linux boxes," in *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, Oct 2017, pp. 324–326.

[9] K. Suo, Y. Zhao, W. Chen, and J. Rao, "vnettracer: Efficient and programmable packet tracing in virtualized networks," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, July 2018, pp. 165–175.

[10] S. Baidya, Y. Chen, and M. Levorato, "ebpf-based content and computation-aware communication for real-time edge computing," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Apr. 2018, pp. 865–870.

[11] Z. Ahmed, M. H. Alizai, and A. A. Syed, "Inkev: In-kernel distributed network virtualization for dcn," *SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 3, pp. 4:1–4:6, Jul. 2018.

[12] T. Graf, "Accelerating envoy with the linux kernel," in *CloudNativeCon Europe and KubeCon Europe*, 2018.

[13] 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 *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: ACM, 2018, pp. 54–66.

[14] S. Jouet and D. P. Pezaros, "Bpfabric: Data plane programmability for software defined networks," in *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, May 2017, pp. 38–48.

[15] L. Deri and S. Sabella, "Merging system and network monitoring with bpf," in *Open Source Developers' European Meeting (FOSDEM)*, 2019.

[16] J. Hong, S. Jeong, J.-H. Yoo, and J. W. Hong, "Design and implementation of ebpf-based virtual tap for inter-vm traffic monitoring," in *2018 14th International Conference on Network and Service Management (CNSM)*, Nov 2018, pp. 402–407.

[17] C. B. Hauser and S. Wesner, "Reviewing cloud monitoring: Towards cloud resource profiling," in *Proceedings of the IEEE 11th International Conference on Cloud Computing (CLOUD)*, July 2018, pp. 678–685.

[18] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 3–18.

[19] E. Schwartz, "A payment protocol of the web, for the web: Or, finally enabling web micropayments with the interledger protocol," in *Proceedings of the 25th International Conference Companion on World Wide Web*, ser. WWW '16 Companion. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2016, pp. 279–280.

[20] B. Gregg, "Performance superpowers with enhanced BPF," in *Proceedings of USENIX Annual Technical Conference (ATC)*. Santa Clara, CA: USENIX Association, Jul. 2017.

[21] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead," *ACM Comput. Survey*, vol. 51, no. 2, pp. 26:1–26:33, Mar. 2018.

[22] M. Marchini, "Enhancing user defined tracepoints," in *Linux Plumbers Conference (LPC)*, 2018.

[23] B. Gregg, "System observability with bpf," in *Linux Storage, Filesystem, and Memory-Management Summit (LSFMM)*, 2019.

[24] B. Gregg, *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[25] T. Høiland-Jørgensen and J. D. Brouer, "Xdp - challenges and future work," in *Linux Plumbers Conference (LPC)*, 2018.

[26] V.-H. Tran and O. Bonaventure, "Making the linux tcp stack more extensible with ebpf," in *Netdev 0x13*, 2019.

[27] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon, "X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 121–135.

[28] J. Wei and C.-Z. Xu, "smonitor: A non-intrusive client-perceived end-to-end performance monitor of secured internet services," in *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ser. ATEC '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 21–21.

[29] M. Wagner, J. Doleschal, A. Knpfer, and W. E. Nagel, "Selective runtime monitoring: Non-intrusive elimination of high-frequency functions," in *Proceedings of the International Conference on High Performance Computing Simulation (HPCS)*, July 2014, pp. 295–302.

[30] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng, "Racez: A lightweight and non-invasive race detection tool for production applications," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 401–410.

[31] B. Sengupta, N. Banerjee, A. Anandkumar, and C. Bisdikian, "Non-intrusive transaction monitoring using system logs," in *Proceedings of the IEEE Network Operations and Management Symposium (NOMS)*, April 2008, pp. 879–882.

[32] C. E. T. de Oliveira and R. F. Junior, "A transparent and centralized performance management service for corba based applications," in *Proceedings of the IEEE Network Operations and Management Symposium NOMS (IEEE Cat. No.04CH37507)*, vol. 1, April 2004, pp. 439–452 Vol.1.

[33] A. Crequy, "bpftrace meets kubernetes with kubectl-trace," in *Open Source Developers' European Meeting (FOSDEM)*, 2019.

[34] A. Hope-Bailie and S. Thomas, "Interledger: Creating a standard for payments," in *Proceedings of the 25th International Conference Companion on World Wide Web*, ser. WWW '16 Companion. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2016, pp. 281–282.

[35] L. A. Trestioreanu, C. Cassagnes, and R. State, "Deep dive into interledger: Understanding the interledger ecosystem," University of Luxembourg, Interdisciplinary Centre for Security, Reliability and Trust (SnT), Tech. Rep., 2019.