Combining System Visibility and Security Using eBPF

Luca Deri[1,2], Samuele Sabella[1] and Simone Mainardi[2]

[1] IIT/CNR, Via Moruzzi 1, Pisa, Italy

[2] ntop, Via Ponte a Piglieri 8, 56122 Pisa, Italy
luca.deri@iit.cnr.it, {deri,sabella,mainardi}@ntop.org

Abstract. Network security is traditionally based on the analysis and dissection of network packets. The widespread use of data encryption and the increase of network traffic created many challenges in terms of visibility and performance, making security tools less effective and both hard to deploy and maintain as network size and speed increase. The advent of eBPF in modern Linux systems enables introspection and adds the ability to inject code in the kernel at specific tracepoints. This work leverages eBPF to combine system introspection with a novel system-level security policer that enables the creation of fine-grained security policies tailored for specific users, processes and containers. This is a major advance for network security applications that can benefit from system introspection to enrich information extracted from network packets, paving the way for the implementation of system- and network-aware security polices that combine visibility and security at a fraction of the computational cost of existing solutions.

Keywords: Traffic Monitoring, Network Security, eBPF, Software Containers.

CEUR-WS.org/Vol-2315/paper05.pdf

1. Introduction and Motivation

Network-based Intrusion Detection Systems (IDSes) such as Snort, Suricata and Bro [1,2] passively monitor network traffic obtained from mirror ports or network Terminal Access Points (TAPs). Similar to antiviruses, most IDSes are signature-based: they identify issues and emit alerts by extracting patterns - often referred to as signatures - from the captured traffic and comparing them against a database of patterns of well-known attacks. In order to carry on these activities, network packets have to be captured, defragmented and reassembled in streams, before their content can be compared against the known signatures. Intrusion Prevention Systems (IPSes) are basically IDSes with the additional capability of passively bridging traffic across two network interfaces: they let legitimate traffic go through, while dropping traffic matching known signatures. Over the last decade, the increase of network speed combined with the widespread use of data encryption has created many challenges to IDSes and IPSes not only in terms of network performance, but also in terms of visibility as most signatures cannot be detected when the traffic is encrypted. The following is an ex-

ample of an IDS rule that triggers an alert by searching the pattern configured in the `pcre` section of the rule.

```
drop tcp $HOME_NET any -> $EXTERNAL_NET any (msg:”ET TROJAN Likely Bot
Nick in IRC (USA +..)”; flow:established,to_server;
flowbits:isset,is_proto_irc; content:”NICK “; pcre:”/NICK .*USA.*[0-9]
{3,}/i”; reference:url,doc.emergingthreats.net/2008124; classtype:tro-
jan-activity; sid:2008124; rev:2;)
```

In addition to the challenges above, the widespread use of operating-system level virtualization also known as containerization, has created additional challenges to network-based security application that rely on network packets. Containers rely on operating system features that enable the creation of multiple isolated instances able to see only a portion of the allocated resources believing them to be all that is available in the system. As containers are very lightweight with respect to virtual machines (VMs), they have become popular to implement fine-grained systems [4] based on microservices [3, 5]. For instance it is common to run hundred of containers on a single physical computer, often part of a larger cluster. Virtualization has taken place also at the level of networks, where physical networks have been replaced with software components such as Open vSwitch [6] implementing distributed virtual multilayer switches used by containers and VMs to communicate. This means that most of the traffic exchanged across containers and processes running on a physical system never hits the network wire as it stays inside the operating system, thus making IDSes and IPSes blind to such virtualized communications. This lack of visibility limits the use of packet-based security tools only to the traffic entering/leaving the physical host, unless an IDS/IPS is deployed per container or VM, option that is unfeasible as this would jeopardize the whole system performance. Alternatively, one can configure a mirror port on Open vSwitch [7], but this determines a significant degradation of the virtual switch performance. In addition, beside speed considerations, it is worth to remark that containerized environments vary continuously as containers are spawned or torn down dynamically [8], whereas network-security tools have a pretty static configuration that might be unable to cope with these dynamically-changing topologies. Finally, it is beneficial for system-level security to keep track of a wider context and not just look at the packets. For instance, packets neither carry the user nor the process responsible for originating/receiving the traffic, information that would be available at the system-level but that do not fit in packets. Inside a system, applications do not see packets at all, as they think in terms or peers to which the information is delivered without having to delve in IP streams and packets.

This paper describes the design and the implementation of a novel tool that enables communications visibility at the operating-system level by monitoring network activities. This is implemented by exploiting Linux kernel probes and tracepoints to intercept communications instead of using packets as traditional applications such as IDSs do. This allows the tool to enrich network communications visibility with system-level metadata, including source and destination processes, users and, optionally, containers. With this rich set of information it is now possible to rethink security by exploiting the correlation of network and system level metadata to implement fine-grained security. This is the main contribution of this work.

The rest of the paper is structured as follows. Section 2 evaluates related work, Section 3 describes the architecture design, Section 4 discusses the tool implementation and experiments, Section 5 highlights some future work activities, and finally Section 6 concludes the paper.

2. Related Work

Sysdig [9] is a Linux kernel module that intercepts system calls and other system level events by leveraging on Linux tracepoints. This information is delivered to user-space applications using packet-like marshalling. This allows tcpdump-like tools to capture, store and replay them as it happens with network packets. As captured events can be many, Sysdig features filters, named chisels, to let Sysdig-based applications receive only the events that are relevant for them, thus reducing the amount of information that needs to be moved from kernel to user-space. While Sysdig allows applications to easily have access to system-level information and for instance explore the network communications without having to deal with packets, its design is sub-optimal for network analysis. Indeed, to opportunely follow network communications, typical calls to `socket()`, `accept()`, `connect()`, and `bind()` need to be tracked in a way similar to what happens with packets that have to be defragmented and reassembled in IP streams, activity that requires CPU cycles and memory. eBPF [10], short for "enhanced Berkeley Packet Filter", is a VM embedded in the Linux kernel that allows applications to be loaded from user-space and injected for run into the Linux kernel. eBPF was originally designed to filter network packets inside the kernel, but then it has been extended to interact with Linux tracers called kprobes (kernel probes) and uprobes (user-space probes). Contrary to Sysdig and pre-eBPF Linux kernel tracers that deliver all collected information to user-space for analysis, eBPF enables the creation of applications that can run inside the Linux kernel on the eBPF VM and thus that delivers to user-space applications the requested information metrics in a consolidated fashion. This has a positive impact on the overall performance as only the necessary information is delivered to user-space and thus eBPF probes are lightweight and do not affect the overall system performance, contrary to what usually happens with Sysdig. By leveraging on the BPF compiler collection (BCC), user-space applications can compile C applications to VM byte code and inject them into the kernel for run. Results are reported to the application in a kernel-to-userspace queue. The drawback of this approach is that BCC has to prevent applications from creating issues to the overall system when running inside the kernel. For this reason, compiled applications have several limitations including the inability to create loops (e.g. a `while()` or `for()` loop) or allocate a significant amount of memory, thus limiting BCC applications to a sort of glue software between the kernel and user-space. Both Sysdig and eBPF have enabled the creation of tools to inspect network communications and enforce network policies. Cilium [11] is a platform that allows eBPF programs to be created and injected on a container virtual ethernet for enforcing network communications at the system-call level and not at the packet level as the Linux firewall does. Cilium does not focus on visibility of container communications but it is designed to make sure network policies are enforced. Sysdig Falco [12] instead is a sort of IDS for containers as it allows container actions (e.g. "start application X", "check if user root executes command Y") to be monitored and alerts to be triggered. As Falco is a passive tool, it is not possible to prevent unwanted actions to be executed but only to be detected and reported.

What is currently missing in all the above tools, is the ability to match network

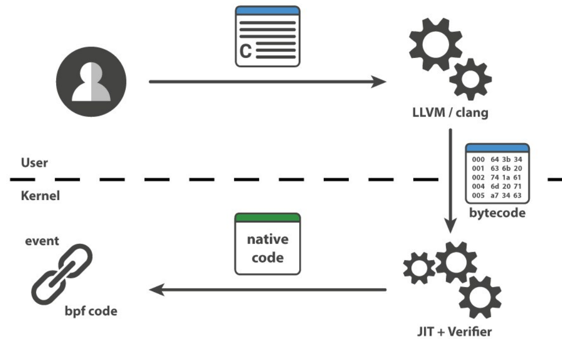

Fig. 1. eBPF Components Interaction.

with system events in order to provide complete visibility both at system and network level, while preventing unwanted network communications to take place similar to what IPSs currently do. This work tries to fill this gap.

3. Architecture Design And Implementation

The designed solution can be logically divided into two components:
- A library, *libebpfflow*, responsible for the generation of network-related eBPF events;
- An application, *ntopng*, in charge of enriching network traffic data with events collected from libebpfflow.

The following section describes the architecture and implementation of libebpfflow and ntopng.

3.1. libebpfflow

The aim of libebpfflow is to intercept and propagate network-related system events. In other words, it leverages eBPF to detect attempts to use the network by processes running in the system. Specifically, libebpfflow:
- Detects Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) communications involving processes executing in the system.
- Marshals detected communications into events with the details of the communication, not only from a network perspective (e.g. source and destination IP addresses) but also from a system perspective (e.g. source and destination processes and system users).

libebpfflow is implemented as C++ library that:
- Wraps eBPF C code necessary to interact with the kernel;
- Detects TCP/UDP communications and generate the corresponding events.

3.1.1. Kernel Interaction with eBPF

During initialization, libebpfflow compiles the eBPF C code into bytecode. If the bytecode satisfies the in-kernel verifier, which checks if the program terminates and is safe to execute, native code is produced by using a Just in Time (JIT) compiler. The native code is then activated into the kernel. This is also schematically represented in Figure 1. Specific kernel probe handlers, named as *kprobes*, can be registered by the library at a certain kernel instruction address to have them executed by the kernel when the function is invoked:

- The *entry* handler, executed by the kernel right before the function call;
- The *return* handler, executed by the kernel right after the function return.

In the return handler it is possible to inspect the return value of the kernel function, whereas in the entry hander it is possible to inspect the parameters passed to the function and make decisions before the function is actually called. libebpfflow can also be instructed to prevent a certain function to be called, pretending it has failed the execution. For example, a web browser could be blocked when it tries to connect to remote destinations on ports different from 80 and 443. This is possible using eBPF `bpf_override_return()`, that provides the ability to artificially tell the kernel to a report failure upon certain function calls. However the Linux kernel supports override returns only on a small group of functions, which does not include the ones concerning network activities. For this reason, the kernel has been patched to enable this mechanism network communications. This mechanism works only in the entry handler, as it would be too late to activate it into the return handler as the function would have already been executed. Using this practice it is possible to prevent selective actions to be executed, and thus enforce arbitrarily-defined security policies. Once the eBFP code has been uploaded, the injected code starts capturing network events generated by the kernel, and pushes them on an event queue shared between kernel and user-space. Each event is delivered in binary format wrapped in a C data structure and includes:

- The time in nanoseconds since the first event has been recorded, useful to report latency information that is computed as difference in time from kernel probe return and kernel probe enter.
- Information about user, application, and container that has triggered the event, as well other metrics such as the network latency for TCP communications.
- The return value of the called function, useful to trace calls that reported an error. This value could help to monitor processes and users with a high-rate of failed calls that might indicate anomalies or suspicious behavior.

3.1.2. Detecting TCP communications

In order to detect TCP communications, we have attached eBPF code to the following functions:

- `inet_csk_accept()`: used by the Linux kernel to accept the next outstanding connection and used by libebpfflow to detect accept events for IPv4 /v6 TCP connections.

- `tcp_connect()` and `tcp_v6_connect()`: same as above but for TCP connect events.

The kprobe attached to the `inet_csk_accept()` return handler, is able to report network information such as addresses and ports, as well as the user id, task, and container (if any) that handled the event. All this data is packed into a data structure that is then copied to user space to be consumed by the application using libebpfflow. As `inet_csk_accept()` returns the socket that will be used for the communication, no state has to be kept between the invocation and the function return. Instead with `tcp_connect()` and `tcp_v6_connect()`, it is necessary to keep a state between the function entry (where the socket is stored), and return (where the return value is collected). For this purpose, an eBPF hash function is used to glue these pieces together and thus be able to compute the TCP connection latency.

3.2. ntopng: Network Traffic Visibility

This section describes ntopng and how it has been extended to use libebpfflow. ntopng [13] is an open source web-based network monitoring software coded by the authors of this paper. It provides a real-time view of network traffic as well as analytics and Key Performance Indicators (KPI) useful for timely identifying cybersecurity flaws, troubleshooting connectivity issues, and analyzing the root cause of outages. At its core, ntopng classifies each packet into a traffic flow. The goal of this classification is to create meaningful summaries out of the raw network packets. Indeed, going through every single packet can be overwhelming for the analyst - at 1Gbps one would have to inspect 1 million packets for every second of analysis. A traffic flow, that can be thought of as stateful representation of an ongoing connection [14], is uniquely identified using the 5-tuple [15] composed of source/destination IP address/port, and protocol. Such connections take place at the transport layer of the Internet layered protocol suite [16] to provide end-to-end communications facilities between remote parties. Therefore, having visibility into the flows equals to having visibility into the communications that are taking place over the network. A web client such as a browser requesting and fetching a page from a web server represents an example of two remotely communicating parties. The initiator, that is, the party that initiates the connection, is also referred to as the client. Similarly, the responder, that is, the party that responds to the initiator, is also referred to as the server. At the flow level, a connection is summarized into one flow, with associated metadata, including:

- Bytes and packets exchanged from the client to the server, and from the server to the client.
- Cybersecurity and application performance monitoring indicators, as well as or other KPIs.
- Layer-7 application protocol of the communication (e.g., HTTP, HTTPS, YouTube, NetFlix).

This contribution capitalizes on libebpfflow with the aim of extending ntopng to enrich flow metadata with:

- Client and server processes information, including Process IDs (PIDs), Thread IDs (TIDs), and process names.

- Client and server users information, including User IDs (UIDs) and Group IDs (GIDs).

To enrich flow metadata with processes and users, ntopng has been extended as follow:
- eBPF event polling and dispatching;
- eBPF event-to-flow classification.

3.2.1. eBPF Event Polling and Dispatching

In order to poll eBPF events, ntopng calls libebpfflow `ebpf_poll_event()`, passing it an event handler that gets called every time a new event is available. Inside the handler, ntopng dispatches the event to a target monitored interface. Dispatching is a necessary operation as ntopng can monitor multiple interfaces simultaneously, whereas events do not carry information on the originating network interface. Dispatching, i.e., associating an event with its originating interface, is done using the event source IP address. For instance events that are originated by loopback address 127.0.0.1 are dispatched only to the monitored loopback interface.

3.2.2. eBPF Event-to-Flow Classification

Ntopng interfaces continuously check for dispatched events before the processing of every packet and even during idle periods. For every event received, event-to-flow classification is performed to attach processes and users information contained in the event to the flow that already contains network data. The event-to-flow classification is performed as follow:
- A flow is searched in the interface cache using eventIP addresses, ports, and protocol.
- Process and user information is attached to the client or the server of the flow.

In order to search for a flow in the cache, source IP address and port, destination IP address and port, and the transport protocol are extracted from the event. When an existing flow is not found in the cache, a new flow is created and added to the cache. Failing to find a flow in the cache is perfectly normal. Indeed, packets processing and events handling are independent and non-synchronized one with the other. Therefore, an eBPF event for a given flow can be processed by ntopng before the first packet of that particular flow has been processed. For instance the eBPF TCP connect event is sometimes observed before the three-way handshake SYN/SYN-ACK/ACK packet sequence (that triggered it) is received, this because packet and event processing have different queueing systems and timings. Depending on the event received, ntopng is able to attach process and user information to the client or the server of the flow. Specifically:
- A connect (TCP) or sendmsg (UDP) event is used to attach information to the flow client.
- An accept (TCP) or recvmsg (UDP) event is used to attach information to flow server.

Therefore, attaching process and user information to both the client and the server requires two events. However, this information is not necessarily available for both

ntop

Flows Hosts Interfaces Devices Search Host

Flow Alerts

Flow Alerts

10 Blacklisted Flow Severity

Date/Time	Severity	Alert Type	Chart	Description	Actions
Sun Nov 18 18:18:05 2018	Error	! Blacklisted Flow		Client, server or domain is blacklisted [Flow: ubuntu18:35700 [simone >_ curl] ⇄ 188.247.135.53 :80] [L4 Protocol: TCP]	

Fig. 2. An Alert for a Flow with Blacklisted Server.

the client and the server. Information availability depends on whether the client, the server or both processes are local to the resource running ntopng:

- When a local client communicates with a local server, then both client and server process and user information is available.
- When a local client communicates with a remote server, ntopng has no eBPF event visibility on the remote server. In this case, only client process and user information is available.
- When a remote client communicates with a local server, ntopng has no eBPF event visibility on the remote client. In this case, only server process and user information is available.

Recall that eBPF events come from the kernel of the resource running ntopng. Therefore, only processes and users that are local to the resource will be visible through eBPF. As a future work, small ePBF probes will be created and disseminated in the network with the aim of circumventing this limitation and have visibility also on remote clients and servers.

4. Validation and Experiments

In this section, the implemented architecture is validated against some real-world scenarios, and evaluated its performance. The results contained in this section also show how this work is different from similar approaches such as Sysdig Falco designed only for container monitoring. In fact ntopng with libebpfflow not only allows to glue network with system events, but also features long term monitoring capabilities. This enables ntopng for instance to report an alert when a user or a process perform a network scan, issue that cannot be reported by Falco as it currently focuses on single events without clustering them with users and processes. This not to mention that Falco is a pure passive tool that is unable to block unwanted actions, contrarily to the proposed architecture that is well-suited for this. Blocking unwanted actions is an ongoing activity as discussed in Section 5.

4.1. Identify Processes and Users Contacting Malware Hosts

As already discussed in the introduction, signature-based IDSs have been traditionally used to detect cybersecurity flaws but, as traffic is becoming more and more encrypted, they are falling short. A simple way to effectively monitor connections involving

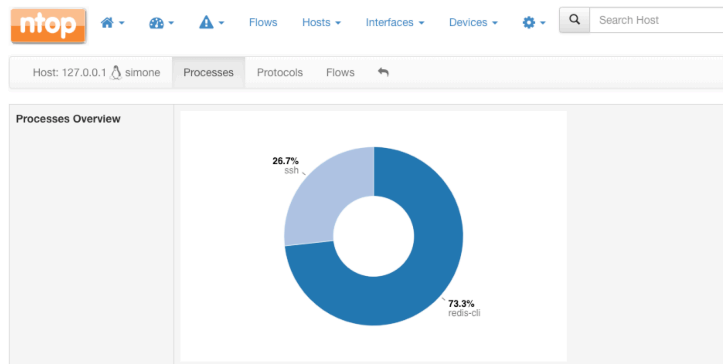

Fig. 3. ntopng: User's Processes Generating Network Traffic.

malware hosts, is by means of IP blacklists. ntopng has built-in support IP blacklists. By default, a malware list is loaded, and alerts are generated every time a host in the blacklist tries to contact or is contacted by any other host. Other custom blacklists can be loaded by the user. Custom and default blacklists are periodically refreshed to always be up to date. However, being able to identify a connection involving a malware host, although fundamental to highlight the cybersecurity anomaly, fails to tell what is the root cause of the trouble. For example, an infected host can be quickly identified with the aforementioned alerts, as soon as it tries to contact a malware host, but none can be said on which process or on which user is actually manifesting the infection via the connection to the malware host. Being able to identify the actual process (and the actual user) that is connecting to a malware host greatly reduces the time required to remove the infection. To this aim, this contribution extends ntopng malware alerts and enriches them with process and user information.
Figure 2 shows a malware alert as reported by ntopng. As it can be seen from the figure, ntopng is not only able to tell that host ubuntu18 has tried to contact blacklisted host 188.247.135.53 on port 80, it is also able to tell that the user in the system that is responsible for such contact is *simone*, who tried to reach the blacklisted host using a *curl* process.

4.2. Binding Network Traffic with Processes, Containers and Users

Modern systems typically run processes for several users at the same time. System administrators rely on users to create fine-grained Access Control Lists (ACLs) or to allow only certain users to perform sensitive tasks. Different users have different privileges, ranging from nobody with the least rights and permissions on the system, to the superuser root who has all rights and permissions. Being able to discover the user's processes that are actively generating network traffic is fundamental to confirm users are behaving as planned by the system administrator and to unveil the following flaws:

- Users abusing certain processes to generate anomalous volumes of traffic.
- Users generating network traffic through processes that are not supposed to create network connections or exchange data over the Internet.

- Users that are not supposed to generate network traffic using certain processes.
- Unprivileged users running processes that should communicate over the network only when run by a superuser.
- Superusers running processes that would require lower privileges to communicate over the network.

To possibly unveil these flaws, ntopng has been extended with the visualization of users' processes that generate or receive network traffic. For example, the donut chart shown in Figure 3 shows all processes run by *simone* on host 127.0.0.1, both as client and as a server. From the figure, it is possible to immediately determine that user *simone* is currently running two processes that generate network traffic, namely *ssh* and *redis-cli*. Process *ssh* accounts for the 26.7% of *simone's* network traffic, whereas *redis-cli* accounts for the 73.3%.

4.3. Find The Layer-7 Application Protocols of a Process or a User

ntopng can detect over 250 Layer-7 application protocols, including those that are considered potentially malicious. This list includes protocols such as Tor or even long-term acceptable protocols such as SSH or SSL that in certain scenarios can hide something more dangerous such as a VPN. By monitoring such protocols, it is possible to leverage ntopng to create a taxonomy of the system processes and users to see how they use the network. In addition, being able to find the Layer-7 application protocols of a process or a user is fundamental for example to unveil the following issues:

- A compromised process or user can use the network with certain suspicious Layer-7 application protocols
- A normal process or user is abusing the network by performing certain unexpected Layer-7 application protocols.

4.4. Performance Evaluation

In order to validate this work, some testing tools have been coded to both measure the loss of eBPF events and their latency in case of heavy system load. The tool is made of a server application that accepts multiple TCP connections from different clients, and a multithreaded client that connects to the server, sends a short message and immediately closes the connection. With 10,000 client threads all running on the same Ubuntu VM, event losses have been measured to be below 0.01% that is satisfactory considered that the host was very unresponsive due to the heavy number of connections. This means that on typical system load, usually no eBPF events are lost.

Another experiment has measured the latency between the time a message is queued by the kernel in the event queue, and the time it appears in the user-space event queue. The kernel time has been read from the user-space using function `clock_gettime(CLOCK_MONOTONIC)`. The measurements reported an average latency of about 5 usec with peak latency of 37 usec. This is a rather low latency compared to libpcap used to capture the traffic. The conclusion is that eBPF events are dropped only under heavy load and that the average latency is acceptable for our use case in case of network and system monitoring.

5. Ongoing Activities

As stated in the previous sections, the goal of this work is not just to provide visibility but also to prevent unwanted activities to happen. As stated previously, Cilium implements security policies in containers by registering policies based on IP address and ports, with some limited Layer-7 policies currently limited to HTTP and Kafka. Cilium enforces policies by attaching eBPF programs to the lowest possible and most performant point in the networking stack to both the Linux tc (traffic control) and eXpress Data Path (XDP) layer. The drawback of this solution is that does not have visibility in terms of users and processes as it is down the networking stack and thus blind to system-level information. Other solutions such as libseccomp (Edge) are very primitives and thus unusable to prevent unwanted activities. For this reason, libebpfflow is being extended with the ability to block functions and system calls that are not compliant with the overall policy. In order to achieve this, Linux kernel had to be patched to enable eBPF to block selected kernel functions. For instance, in order to block calls to `tcp_v4_connect()` used to initiate a TCPv4 connection, the Linux kernel source file tcp_ipv4.c has to be patched to enable eBPF error injection by adding an extra line `ALLOW_ERROR_INJECTION(tcp_connect,ERRNO).` Currently libebpfflow allow to implement policies such as:

- User *simone* cannot start application curl when inside a container.
- Firefox cannot connect to the Internet.
- User *paolo* cannot accept incoming TCP connections outside of business hours.

While libebpfflow is proving to work also to block calls, there are still a few issues that still need to be addressed. The main one is the inability to create loops in eBPF that libebpfflow from handling more than 14 rules. Alternatives for this limitation are actively being searched. Among the viable alternatives is the ability to store rules in various kernel BPF hashes: one hash for user-based policies, another for processes, and another for communications. While this looks an interesting solution, it has the drawback that hash lookups would have to be performed for every tracked kernel function, with a possible negative impact on the overall performance. Another alternative is the ability to generate policies in C code in user-space, compile them on-the-flight and pass them to the kernel JIT. All alternatives are viable, but the main issue is that none of them allows to setup thousand of rules due to the eBPF limitations in terms of memory and lookup speed.

6. Final Remarks

This paper covers the design and implementation of a novel eBPF-based tool that combines system and network visibility for the detection of security flaws and the enforcement of security policies.. In the validation section, some real use cases have been discussed in order to demonstrate that the integration of network and system information is a step ahead with respect to pure packet-based approaches. In essence this paper shows that the ability to go beyond IP addresses, protocols and ports en-

ables the creation of a new generation of security tools that keep into account system-level metadata such as users, processes, and containers.

Code Availability

The ntopng source code is available at https://github.com/ntop/ntopng. As the libebpf-flow is coded in C++, we have created a Python version of the library available at https://github.com/samuelesabella/ebpflow that retains the same performance as the eBPF code is injected in the Linux kernel, but that is easier to read and modify than the original version.

References

1. Park, W. & Ahn: Performance Comparison and Detection Analysis in Snort and Suricata Environment. In S. Wireless Pers Commun (2017) 94: 241.
2. Mehra P.: A brief study and comparison of Snort and Bro Open Source Network Intrusion Detection Systems. In International Journal of Advanced Research in Computer and Communication Engineering Vol. 1, Issue 6 (2012).
3. Bass, L., Weber, I., & Zhu, L.: DevOps: A software architect's perspective. Addison-Wesley Professional (2015).
4. Newman, S.: Building microservices: designing fine-grained systems. O'Reilly Media, Inc (2015).
5. Thönes, J.: Microservices. IEEE software, 32(1), 116-116 (2015).
6. Pfaff, B., Pettit, J., Koponen, T., Jackson, E. J., Zhou, A., Rajahalme, J., & Amidon, K.: The Design and Implementation of Open vSwitch. In NSDI (Vol. 15, pp. 117-130) (2015).
7. Shanmugalingam, S., Ksentini, A., & Bertin, P.: DPDK Open vSwitch performance validation with mirroring feature. 23rd International Conference on ICT (pp. 1-6) (2016).
8. Bernstein, D.: Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing, (3), 81-84 (2014).
9. Borello, G.: System and Application Monitoring and Troubleshooting with Sysdig. Usenix Lisa 2015 conference (2015).
10. Gregg, B.: Linux Performance Analysis New Tools and Old Secrets. Usenix Lisa 2014 conference (2014).
11. Makowski, Ł., & Grosso, P.: Evaluation of virtualization and traffic filtering methods for container networks. Future Generation Computer Systems (2018).
12. Stemm, M.: SELinux, Seccomp, Sysdig Falco, and you: A technical discussion. https://sysdig.com/blog/selinux-seccomp-falco-technical-discussion/.
13. Deri, L., Martinelli, M. and Cardigliano, A.: Realtime High-Speed Network Traffic Monitoring Using ntopng. Usenix Lisa 2014 conference (pp. 70-80) (2014).
14. Brownlee, N., Mills, C. and Ruth, G.: RFC 2722 Traffic Flow Measurement: Architecture.
15. Bagnulo, M., Matthews, P. and Beijnum, I.V.: RFC 6146 Stateful NAT64: Network address and protocol translation from IPv6 clients to IPv4 servers.
16. Stevens, W.R.: RFC 1122 TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms.