eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond)

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 5 May 2023, accepted 21 May 2023, date of publication 30 May 2023, date of current version 13 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3281480

RESEARCH ARTICLE

eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond)

DAVID SOLDANI[1], (Senior Member, IEEE), PETRIT NAHI[1], HAMI BOUR[1], (Member, IEEE), SABER JAFARIZADEH[1], (Member, IEEE), MOHAMMED F. SOLIMAN[1], LEONARDO DI GIOVANNA[2], FRANCESCO MONACO[2], GIUSEPPE OGNIBENE[2], AND FULVIO RISSO[2], (Member, IEEE)

[1]Rakuten Mobile Inc., Setagaya-ku, Tokyo 158-0094, Japan
[2]DAUIN Department, Politecnico di Torino, 10129 Turin, Italy

Corresponding author: David Soldani (david.soldani@rakuten.com)

ABSTRACT Modern mobile communication networks and new service applications are deployed on cloud-native platforms. Kubernetes (K8s) is the de facto distributed operating system for container orchestration, and the extended version of the Berkeley Packet Filter (eBPF) – in the Linux (and MS Windows) kernel – is fundamentally changing the approach to cloud-native networking, security, and observability. In this paper, we introduce what eBPF is, its potential for Telco cloud, and review some of the most promising pricing and billing models applied to this revolutionary operating system (OS) technology. These models include schemes based on a data source usage model or the number of eBPF agents deployed on the network, linked to specific eBPF modules. These modules encompass *network observability*, *runtime security*, and *power dissipation* monitoring. Next, we present our eBPF platform, named *Sauron* in this work, and demonstrate how eBPF allows us to write custom code and dynamically load eBPF programs into the kernel. These programs enable us to estimate the *energy consumption* of cloud-native functions, derive *performance counters and gauges* for transport networks, 5G applications, and non-access stratum protocols. Additionally, we can detect and respond to *unauthorized access* to cloud-native resources in real-time using eBPF. Our experimental results demonstrate the *technical feasibility of eBPF* in achieving highly performant monitoring, observability, and security tooling for current mobile networks (5G, 5G Advanced) as well as future networks (6G and beyond).

INDEX TERMS eBPF, extended Berkeley packet filter, cloud-native observability, cloud-native security, cloud-native networking, cloud-native monitoring, 5G, 5G Advanced, 6G, Kubernetes, K8s.

I. INTRODUCTION

Cloud-native architecture and technologies are an approach to building and running scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, pods, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this methodology [1].

The associate editor coordinating the review of this manuscript and approving it for publication was Nurul I. Sarkar.

Kubernetes, also known as K8s, is an open-source distributed operating system for container orchestration, i.e., for automating deployment, scaling, and managing containerized applications. Kubernetes is the pilot (from Greek) of a ship of, e.g., Docker containers [2].

A Cloud Native Network Function (CNF) is a virtual network function designed and implemented to run inside containers. CNFs inherit all cloud-native architectural and operational principles, including Kubernetes lifecycle management, agility, resilience, and observability. The Cloud

Native Computing Foundation (CNCF) provides a detailed definition of ''CNF'' and a list of essential tests for obtaining a CNF Certification [1].

An example of a cloud-native radio access network is the Open virtual Radio Access Network (Open vRAN), defined by O-RAN Alliance [3]. The disaggregated architecture and containerized approach to run virtual Centralized Unit (vCU) and virtual Distributed Unit (vDU) – designed to be easily installed and to interoperate with existing system components – on commercial off-the-shelf (COTS) hardware bring many benefits to network carriers, particularly in terms of total cost of ownership (TCO), automation, and innovation.

The next generation of Mobile Communication System (6G) is expected to be cloud native and secure by design powered by the extended version of the Berkeley Packet Filter (eBPF). However, new capabilities and what differentiates 6G from previous generations depend on stakeholders and what standard development organizations adhere to [4]. The international Working Party (WP) for the overall radio system aspects of International Mobile Telecommunications (IMT) systems is the ITU-R WP5D [5], but the 3rd Generation Partnership Project (3GPP) is also discussing 5G Advanced/6G within the scope of its release process with 6G requirements to be defined as part of Releases 19 and 20 starting in 2024.

The extended version of the Berkeley Packet Filter (BPF) is an abstract virtual machine (VM) with its own instruction set that can execute user-defined programs inside a sandbox in the Linux (and MS Windows) kernel. *eBPF enables programs to run in the kernel of the host operating system and to instrument the kernel without changing the kernel source code* [6].

This technology enables dynamic programming of Linux and Windows kernels, allowing behavior changes without reboot. Moreover, it ensures the safety and stability of loaded programs by preventing kernel crashes. However, making changes to the kernel can be a complex process and requires significant time for acceptance, as community consensus is necessary to incorporate changes into the upstream kernel distribution for production use. The vast majority of kernel versions deployed in production environments provide robust and extensive support for the essential eBPF features.

eBPF programs can be verified at load time to prevent kernel crashes and other instabilities. The verifier examines the program as it is loaded into the kernel and ensures that it runs to completion, all the memory access is safe, and the machine does not crash. (There are ways to exceed the theoretical limit of a million instructions by writing loops and chaining the program codes.) These sandbox programs are then triggered by kernel events, receiving pointers to the kernel or user space memory. Maps allow sharing information between the kernel and user space as well as exchanging data between eBPF programs in the kernel.

Beyond that, when Kubernetes or other cloud platforms are used to run multiple applications on a virtual machine (VM), eBPF provides excellent visibility of all programs on the host machine and policy enforcement capabilities in the kernel space to mitigate vulnerabilities.

Among other benefits discussed in the following section, due to its low overhead, eBPF programs enable us to simplify the kernel networking stack and reduce latency between end points. eBPF is ideal for Telco cloud as its programs can be made portable between kernel versions, with little effort, and can be updated atomically, avoiding disruption of workloads and the need to reboot nodes.

This article is organized as follows: Section II presents the powers of eBPF for cloud-native systems and Telco clouds. Section III introduces the concept and architecture of our eBPF solution for Telco, denoted as *Sauron,* in this article. (The name was inspired by The Lord of the Rings author J.R.R. Tolkien's writings, where Sauron's goal was to use the One Ring to influence and control the minds of those who wielded the lesser rings.) In this section, we also introduce some examples of Sauron's modules, namely eBPF for Transport, eBPF for Observability, eBPF for Energy and eBPF for Security. The experimental validation of the corresponding case studies and obtained results are discussed in Section IV. Finally, conclusions are drawn in Section V.

II. THE POWERS OF eBPF

The eBPF project layout is illustrated in Figure 1 and it consists of three pillars:

1) ***eBPF programs***, that run in the kernel and react to events.
2) ***User space programs***, which load a) to the kernel and interact with them.
3) ***BPF Maps***, that allow data storage and information sharing between a) and b).

In the ***development phase***: 1) A program, typically written in *C* or *Rust*, is compiled into an object Executable and Linkable Format (ELF) file, which can be analyzed using tools such as *readelf* – since the Linux kernel expects eBPF programs to be loaded in the form of *bytecode*, a compiler suite like clang/LLVM[1] can be used to compile restricted-*C* code into eBPF *bytecode*. 2) The *ELF file* – which contains the program *bytecode*, definitions of *Maps* and *BPF Type Forma*t (BTF) – is loaded into the kernel using a monolithic approach enabled by *bpf system calls*; 3) *Maps* are created into the kernel space before loading the eBPF programs that refer to them – the eBPF programs can utilize eBPF maps to share and retrieve data in a variety of formats and preserve the state. eBPF maps can be accessed from user space applications and through helper functions from eBPF programs. Hash tables, Arrays, Least Recently Used (LRU), Ring Buffer, Stack Trace, and Longest Prefix Match (LPM) are examples of maps supported in the kernel space.

In the ***runtime phase***: 4) The eBPF program (map re-wring and instructions) is loaded into the kernel, where it is verified (eBPF Verifier) and, in most cases, compiled (JIT Compiler)

[1]The name ''LLVM'' is the full name of the project. See: https://llvm.org

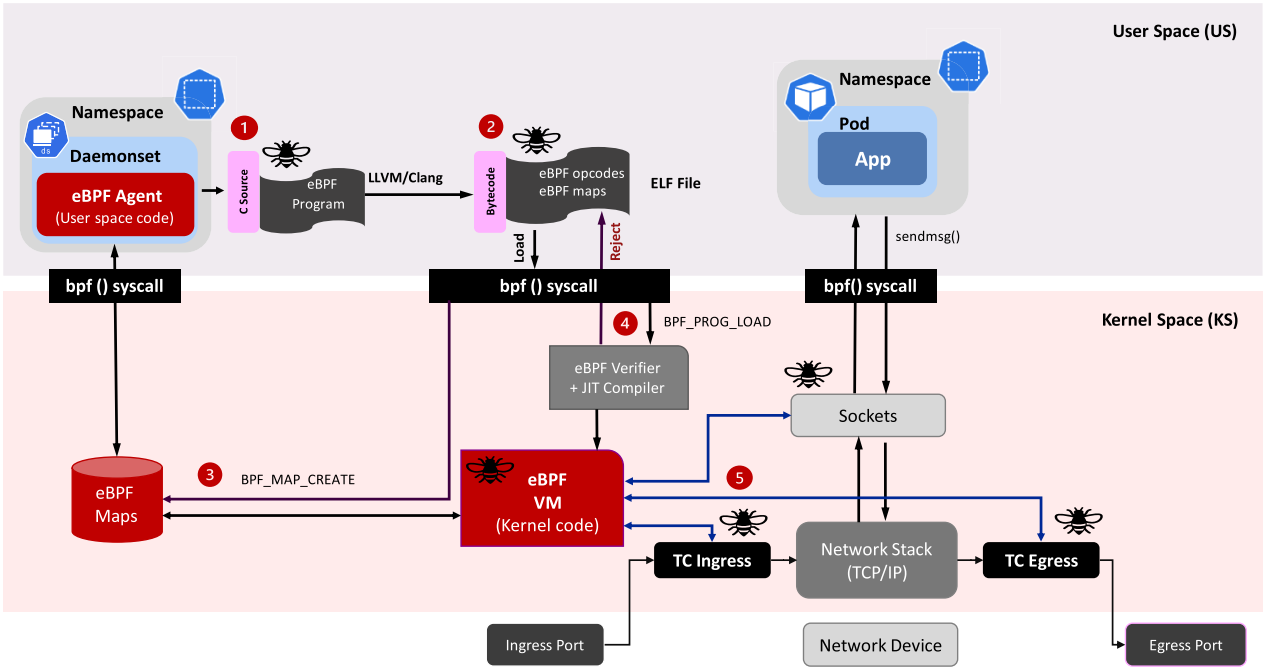


FIGURE 1. eBPF project layout.

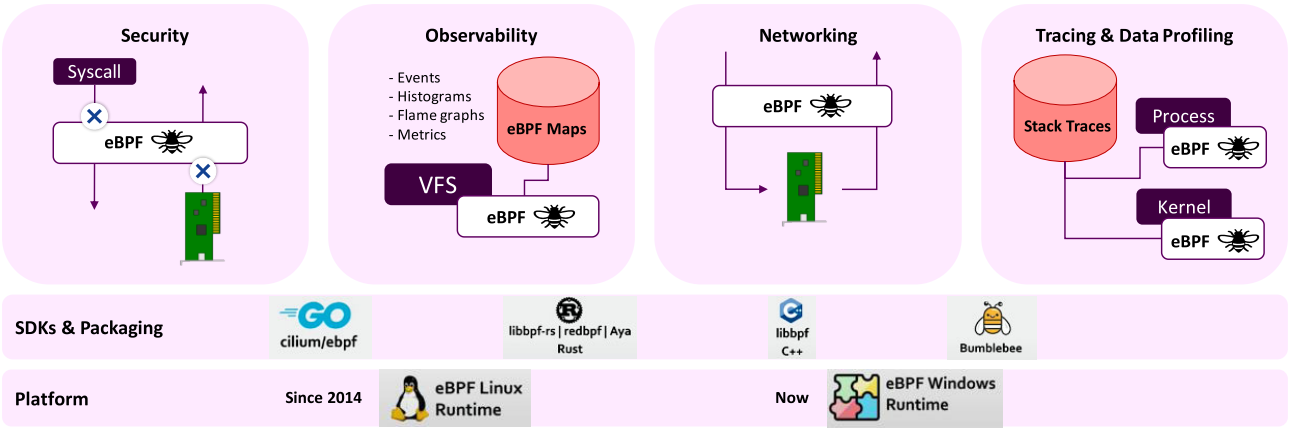


FIGURE 2. eBPF stack: platform, SDKs, and packaging, and use cases [9].

into native instructions – typically, the loading phase and the interaction with the kernel structures is accomplished by using one of the available eBPF libraries and several development toolchains exist to achieve this goal. For instance, the *Go ebpf-go* library or the *C libbpf* library may be used for this purpose. The *libbpf* library is a C/C++-based generic eBPF library that allows you to load the ELF file into the kernel by abstracting the interaction with the BPF system call; see Figure 1. Some examples of eBPF programs (in *C* and *Rust*) and user space code to load eBPF object (ELF) files in the kernel using *libbpf-bootstrap* can be found in [7]. Another toolkit for creating efficient kernel tracing and manipulating programs is BPF Compiler Collection (BCC). BCC uses a Python interface (and program) in the user space to generate the eBPF bytecode and to load that into the kernel [8]. 5) The eBPF program is attached to selected events (injected in the desired kernel hooks) – each eBPF program loaded into the kernel is triggered by an event using a program file descriptor, i.e., the attached eBPF program is executed once the event has occurred. In other words, eBPF programs allow custom code to execute in the kernel when the kernel or an application passes a certain hook point (e.g., TC ingress/egress, and sockets, in Figure 1), such as, but not limited to, a *system call*, a *function entry/exit*, a *kernel tracepoint*, a *network event*, etc. The eBPF programs provide visibility between the kernel and user space using BPF maps.

For example, we can run eBPF programs using a set of eBPF hooks that are supported in the networking stack of the

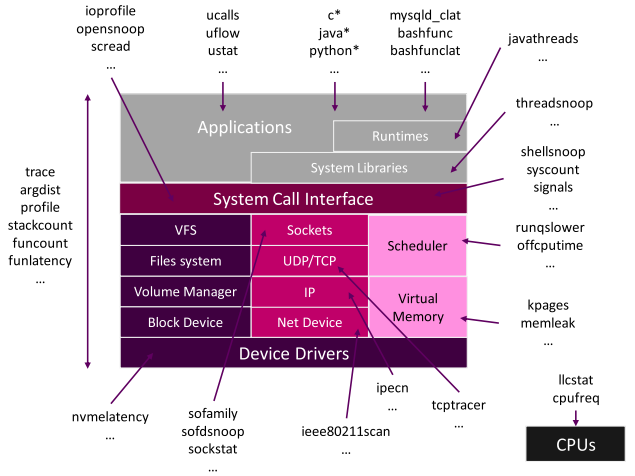


FIGURE 3. Linux BCC/BPF tracing tools [10].

Linux kernel. In addition, higher-level networking constructs can be created by combining the hooks below:

- ***Express Data Path (XDP)***: the networking driver is the earliest point to which the XDP BPF hook can be attached. The eBPF program is triggered to run when a packet arrives in the driver.
- ***Traffic Control (TC) ingress/egress***: like XDP, eBPF programs are attached to a networking interface by hooking the eBPF programs to the traffic control ingress. In contrast to XDP, the eBPF program will run following the initial packet processing in the networking stack.
- ***Socket operations***: the socket operations hook attaches the eBPF programs to a specific *cgroup* and triggers them based on TCP events.
- ***Socket send/recv***: the socket *send/recv* hook triggers and runs the attached eBPF programs for each TCP socket send operation.

Since some network cards support XDP offload, we can run the eBPF program on a network card and process network packets when this event is triggered, i.e., before the packet arrives in the CPU. This allows a "zero CPU use" to investigate incoming packets for firewalling, DDoS mitigation etc.

If a predefined hook does not exist for a particular need, it is possible to instrument function calls, using *kprobes* for kernel space functions and *uprobes* for user space functions, and attach eBPF programs almost anywhere in the kernel or user applications. Security-related actions can be taken using Linux Security Module (LSM) hooks (Linux kernel version 5.7 and later releases).

Thanks to the wide variety of events in the kernel, eBPF programs can be utilized for efficient ***networking***, ***tracing and data profiling***, ***observability***, and ***security*** tooling, such as threat defense and intrusion detection, as shown in Figure 2, where the eBPF stack (eBPF Linux runtime and eBPF MS Windows runtime, SDKs, and packaging examples, and use cases) is detailed [9].

Figure 3 depicts a comprehensive set of eBPF tools for observability. The figure represents the broad range of system components that can be instrumented by means of eBPF programs. In [10], the author provided more details on BPF performance tools.

More information on eBPF and its applications can be found in [11], where authors restricted their focus to four key application domains related to networking, security, storage, and sandboxing. They discussed various solution techniques for each application domain to enable researchers and practitioners to adopt eBPF into their designs easily.

A. WHY eBPF?

According to Gartner, by 2025, over 95% of new digital workloads will be deployed on cloud-native platforms, up from 30% in 2021 [12]. Simultaneously, Kubernetes is becoming the de facto standard for cross-cloud orchestration and a pillar of cloud-native architectures [13]. Cloud-native architecture is one of the critical drivers of eBPF-based applications; as more kernel subsystems become extensible using eBPF, drivers and kernel modules could be written in eBPF soon.

Figure 4 consolidates some of the values brought by eBPF to cloud-native environments. This revolutionary technology shows amazing programmability, and the eBPF verifier ensures that the loaded programs are safe and guarantees that they do not crash the kernel. Beyond that, eBPF provides excellent visibility and enforcement control of policy and allows operators to observe all programs running in the user space, which is hardly ever achieved by applications that operate in the same space. Moreover, eBPF presents low overhead, making it ideal for containers networking, observability, and Telco cloud security in production.

Table 1 outlines ten advantages of eBPF, demonstrating how this technology is expected to evolve to support a wide variety of additional use cases. Those include the possibility of using *XDP* and *AF_XDP* direct path, which integrates naturally with Kubernetes, as opposed to Data Plane Development Kit (DPDK), and achieve sustainable computing projects like Kubernetes-based Efficient Power Level Exporter (KEPLER), an open-source power level exporting tool that shows energy consumption data via eBPF for Kubernetes clusters [14]. KEPLER collects data based on eBPF programs that attach to Linux *tracepoints* and performance counters to collect information such as *process id*, *cgroup id*, *cpu cycles*, *cpu time*, *cpu instructions*, *cache misses* etc. The aggregated data are then used in conjunction with other stats in the user space to estimate the energy consumption of pods, as described in Section III-D.

B. eBPF FOR KUBERNETES

eBPF enables the programmability of the kernel in Kubernetes too. There is only one kernel on a host, and any application running in a container, inside a pod in the case of K8s, must still use the kernel whenever it requests access to hardware, a file, or receive a message from the network

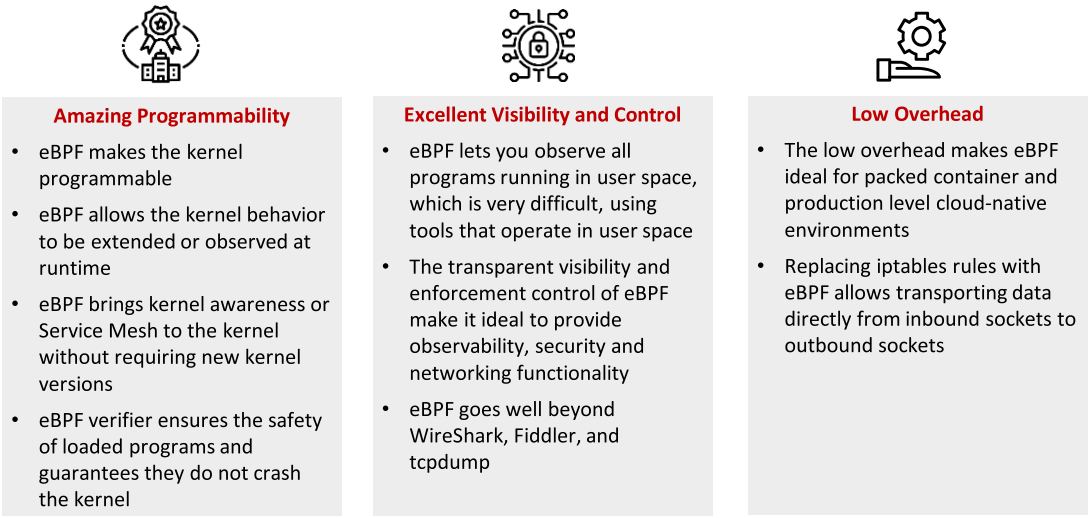


FIGURE 4. Examples of eBPF advantages.

(networking) as depicted in Figure 5. The kernel is always involved, and there is only one kernel regardless of how many pods are deployed on a machine (BareMetal or virtual machine). Even containers do not have their own kernel, they employ the existing kernel running on the host machine.

Therefore, through proper eBPF instrumentation in the kernel, an agent can be made aware of everything occurring in the user space across all applications or cloud-native functions (micro-services). This enables complex eBPF tools to have a view across the entire node, enabling deep observability in the cluster, as shown in Figure 5 [6].

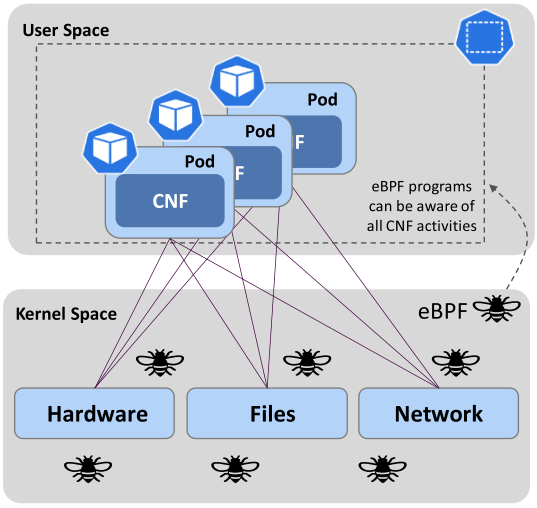


FIGURE 5. How eBPF can be aware of all application activities involving the kernel [6].

1) CONTAINER NETWORKING

Beyond that, eBPF can be used for creating an efficient and scalable K8s network provider, for providing security, achieving high-performance load balancing (K8s proxy replacement, north-south load-balancer), and multi-cluster and external workloads connections, such as, e.g., egress gateway, integration of metal and VMs, global services, service discovery, etc.

In Kubernetes, we use namespaces to isolate pods on a node, and we typically have a *network namespace* for each pod, meaning that the kernel runs a networking stack for each pod. This is very efficient in terms of CPU utilization, for instance, but the path for a packet that arrives at the Network Interface Card (NIC) of the host machine to reach an application in the user space, is relatively convoluted. For example, as shown in Figure 6 a), a packet that arrives at eth0 of the NIC traverses the kernel networking stack on the host; it is then routed across a virtual ethernet (*veth*) connection into the network namespace of the pod, and finally, the packet goes through the networking stack in the pod to reach the application or CNF, in the case of Telco cloud.

eBPF allows us to simplify the networking stack in the kernel, see Figure 6 b), and connect pods as endpoints [6]. As depicted in Figure 7, this is achieved using a hash table – eBPF service (svc) Map – that maps *Service (Cluster) IP* onto *Endpoint (Pod) IP* addresses. The table states: if you send a network packet to the Service IP (Cluster IP) using this IP address (10.107.72.75), then that packet will be delivered to this pod (Endpoint IP). This allows us to bypass the kernel networking stack on the host, as shown in Figure 6 b), and send that packet directly into the stack of the pod, for example by using the XDP hook, which runs on the NIC of the host machine.

In other words, *replacing the iptables rules with eBPF maps allows operators to transport data directly from inbound sockets to outbound sockets, which enables super-fast service load balancing with eBPF*.

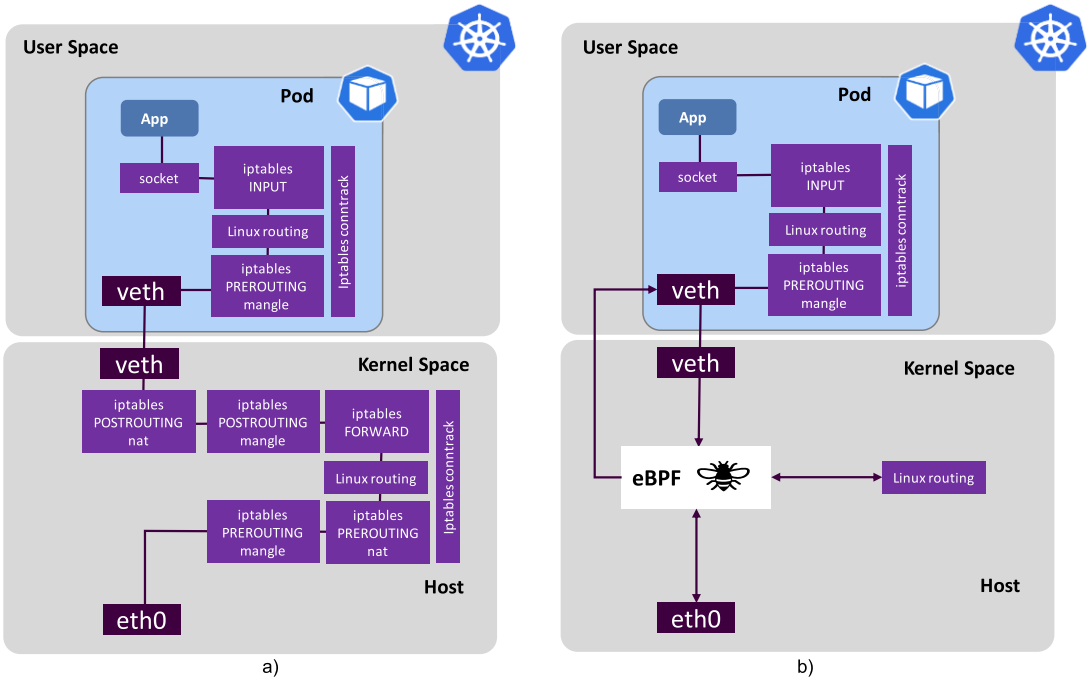


FIGURE 6. How eBPF advantages for pod networking [6], [15], [21].

Figure 7 illustrates how load balancing between four replicated pods on two worker nodes in a Kubernetes cluster is achieved by employing eBPF [16]. Replacing Kube-Proxy with a proper eBPF Agent, we can attain much better performance than using the basic routing capabilities of the host operating system (iptables DNAT).

For example, a containerized high-performance load balancer that distributes traffic using eBPF/XDP within the Linux kernel was presented in [17]. Experimental results demonstrated that the throughput performance of the proposed load balancer using eBPF was considerably better than that of iptables DNAT.

2) SERVICE MESH

A *Service Mesh* is a dedicated infrastructure layer that can be added to applications or CNF micro-services. This layer allows us to transparently add capabilities to applications, without requiring the addition of code. Service Mesh provides connectivity between applications at the service level, abstracts the underlying network, and offers features such as observability, security, and traffic management.

The evolution of the Service Mesh model is shown in Figure 8. The traditional approach was to use a Shared Library Model (Service Mesh Library). Then Service Mesh was achieved by implementing a cloud-native *Sidecar Model* (Service Mesh Sidecar). Now, with eBPF, many Service Mesh functionalities can be moved to the kernel (Kernel Model) and only complex L7 processing tasks need to be delegated to a Sidecar Container in the user space, such as an Envoy proxy (Service Mesh proxy), as described below.

The Sidecar Model allows us to place common code in the same container as the application or the CNF (or micro-service) and to run the same container in each pod so that each pod is instrumented in the same manner. That container is a *Sidecar Container*. The Sidecar Model has been widely used for logging, tracing, security tooling and service mesh [6].

Moreover, to be aware of what is happening in the pod, a Sidecar Container must run in the same *network namespace* as the application or CNF (or micro-service) in the case of Telco cloud. To deploy the sidecar within that pod, we need to either statically specify it into the pod Yaml manifest file or we can use an automatic mechanism such as the one offered by a Service Mesh implementation (see, e.g., Istio Service Mesh [18]) If, for whatever reason, any of the selected technique is not successfully employed, the pod will not be subject to the specified instrumentation and the desired visibility will not be achieved.

Since the eBPF program is loaded into the kernel, no pod configuration modifications are required. And once the eBPF program is attached to an event, the event will be triggered independent of the pod status (ready or not ready). Therefore, there is no need to restate or reconfigure the pod in question, as the event is triggered by the kernel, and the eBPF program is automatically executed. In addition, eBPF is aware of all activities carried out on that node (both expected and malicious activities), and based on this information, an eBPF security agent can see a malicious process running on the node and mitigate the related risks, e.g., enforcing some network security policies (as detailed in Section III-C), whereas a Sidecar Container cannot.

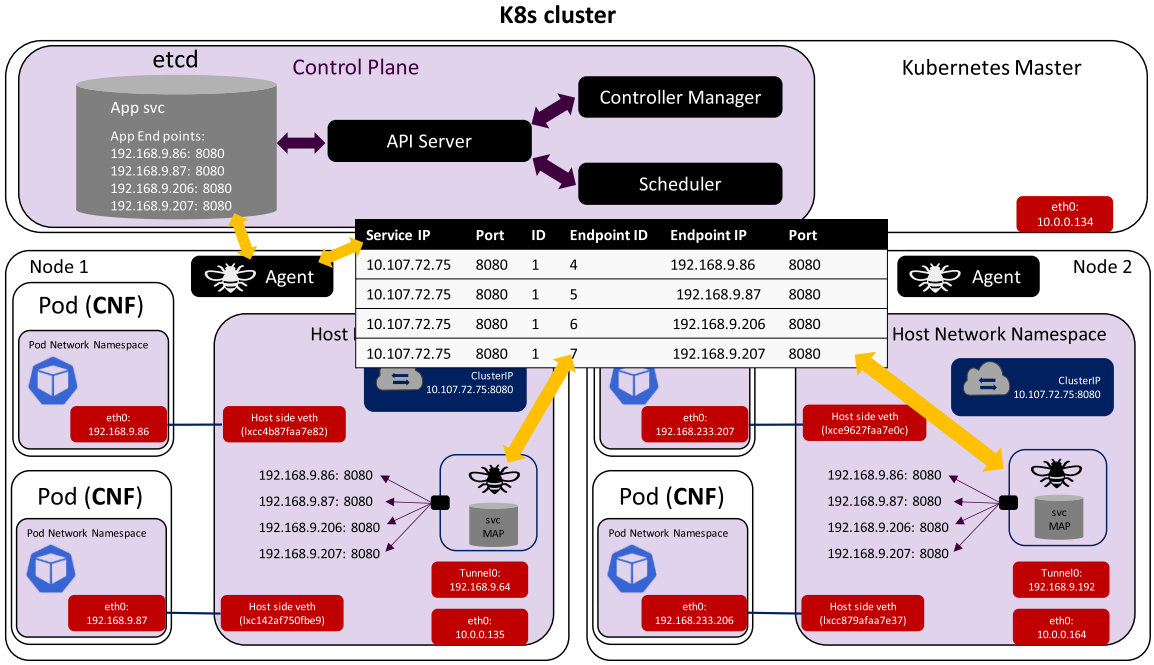


FIGURE 7. Example of load balancing using eBPF in a K8s cluster [16].

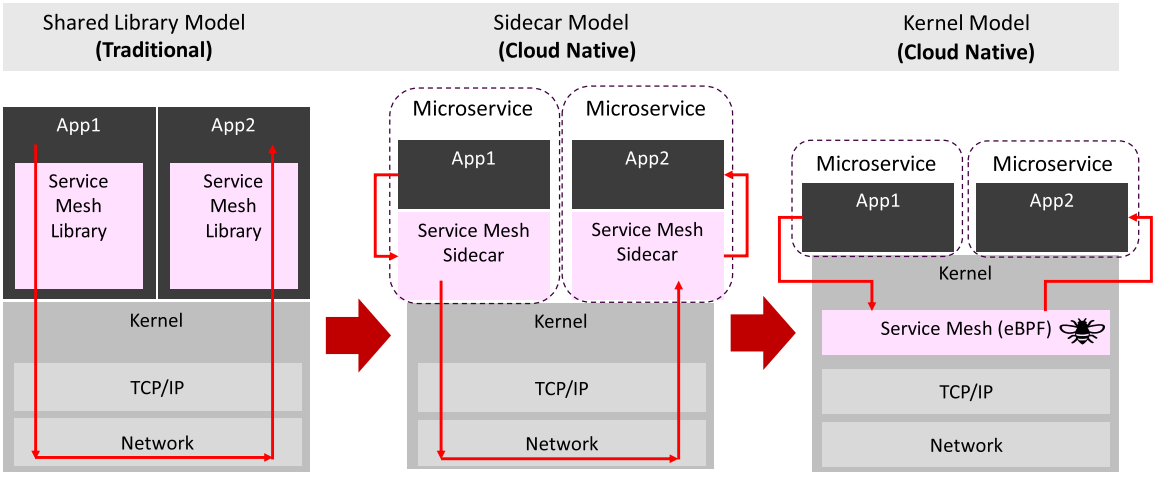


FIGURE 8. Service mesh models: shared library, sidecar model and kernel model (eBPF).

In addition, the Sidecar Model can waste many resources in the cluster because each pod must be configured with adequate memory and CPU quotas for the applications (or micro-services, in the case of Telco cloud) and Sidecar Container to perform. Besides that, we need duplicate copies of state configuration information within each pod because, by design, pods are isolated from one another, and pod-to-pod communication is limited to network messages or shared files. In contrast, eBPF maps are data structures that make data sharing between eBPF programs in the kernel, and eBPF tooling agents, in the user space extremely efficient.

Typically, service meshes utilize a *network proxy* to manage and process application-layer traffic (Layer 7, L7). However, other service meshes employ a Sidecar Model for such a proxy feature. In the latter case, as already explained above, a proxy is injected into each application pod, as depicted in Figure 9 a).

The advantage of using eBPF is that it enables the Service Mesh Proxy to be shared across multiple pods, as shown in Figure 9 b). Avoiding the Sidecar Model reduces the resources and complexity required to configure the proxy in each pod. Additionally, as already pointed out, we can achieve significantly more efficient networking, thereby reducing latency, which is critical for Telco to deliver carrier-grade communication services.

As shown in Figure 9 a), using the Sidecar Model, each network packet must traverse the networking stack multiple times to pass through the sidecar proxy. With eBPF, we can

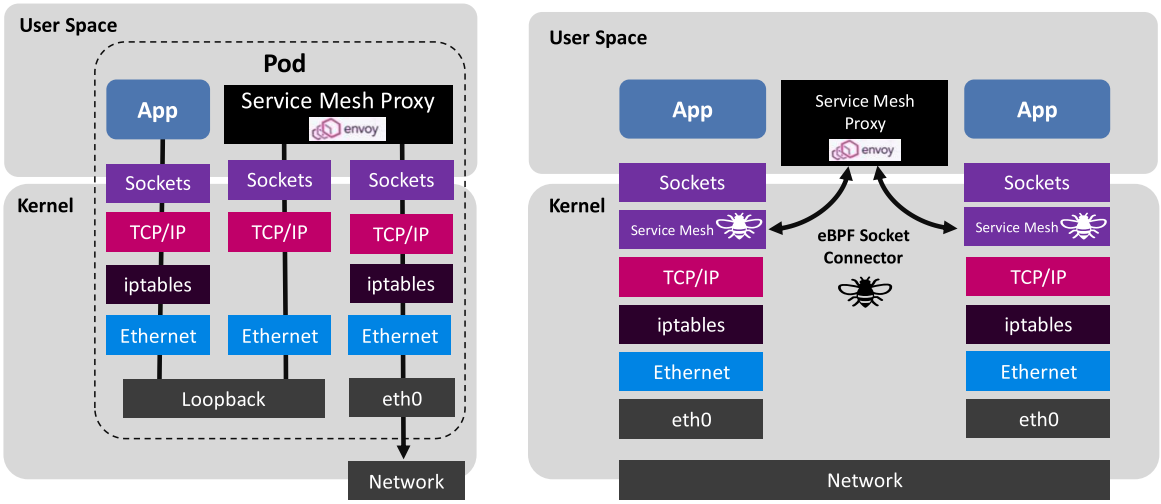


FIGURE 9. How eBPF may reduce the number of service mesh proxies to provide the same services [6], [15], [18], [21].

run the proxy outside of the pod and create a direct path between pods, with L7 traffic passing through the proxy only once and only when that is required, as illustrated in Figure 9 b) [6], [15].

Another example is the *mutual authentication between application workloads*. Traffic that flows between different nodes can be encrypted in the kernel using *IPsec* or *WireGuard* [19]. Those solutions are widely used and straightforward to implement. However, when next-gen mutual Transport Layer Security (TLS) is required – mTLS encryption, using X. 509 digital certificates as application workload identities, see Figure 10 a) and b) – you can use an Identity Management System (IMS) of your choice to manage the digital certificates, as shown in Figure 10 b), which represents each individual application workload, and eBPF to inject those certificates into the kernel, and use kernel level encryption for encrypting the traffic [6].

To summarize, the evolution of the Service Mesh model employing eBPF enhances resource utilization and maintainability by reducing the number of sidecar proxies and, in some cases, eliminating them.

Whenever feasible, eBPF native (no sidecar) allows *traffic management* (L3/L4 forwarding and load balancing, Canary, Topology Aware Routing, Multi-cluster), *security* (NetworkPolicy, mTLS), and *observability* (tracing, Open Telemetry and Metrics; HTTP, TLS, DNS, TCP, UDP, etc.), when it is not possible, such as in the case of *traffic management* (L7 load-balancing and Ingress), *resilience* (retries, L7 rate limiting), and upper layer *security* (TLS termination and origination), only a single sidecar proxy can be injected, as depicted in Figure 9 [6], [15].

3) NETWORK SECURITY

Another important feature that eBPF brings to cloud-native environments, particularly valuable for Telco cloud, is *network security for microservices*.

Traditionally, containers runtimes (e.g., Docker) apply security policies and NAT rules per-container level by configuring iptables rules in the docker hosts. Using iptables, policies can only be enforced based on Layer 3 and Layer 4 parameters. Moreover, the container must re-construct the whole packet and forward that to the host to decide whether to drop or deliver it.

As described in detail in Section III-C, with eBPF, container security policies do not have the limitations of iptables, eBPF policies can be applied to the system call, before entering the stack or constructing the packet, and all calls may be intercepted and filtered on the spot, making the solution much faster. Also, eBPF allows us to apply security policies based on application-level verbs, such as rest *get/post/put/delete* or specific paths.

C. WHY eBPF NOW?

The market potential of eBPF is reflected in several significant companies changes in 2022, including the following examples:

- "*New Relic acquires Pixie Labs, a Next Generation of Machine Intelligence and Observability.*"[2]
- "*Datadog acquires Seekret and they are excited to leverage Seekret eBPF expertise to unlock new capabilities.*[3]"

Seekret uses powerful eBPF technology to auto-discover and visualize API assets, interconnections, and dependencies, enabling developers and product leaders to understand API behavior and usage patterns in complex, dynamic environments.

The industry comprehends eBPF, and a notable increase in the industry utilization and understanding of eBPF has created more buyers. For example, at a major Kubernetes conference, the discussion was on eBPF with one commentator stating that "*KubeCon was basically eBPF.*[4]"

[2]https://www.bloomberg.com/press-releases/2020-12-10/new-relic-signs-definitive-agreement- to-acquire-pixie-labs-a-next-generation-machine-intelligence-observability-solution-for

[3]https://www.datadoghq.com/blog/datadog-acquires-seekret/

[4]https://www.techtarget.com/searchitoperations/opinion/Looking-back-on-KubeCon-CloudNativeCon

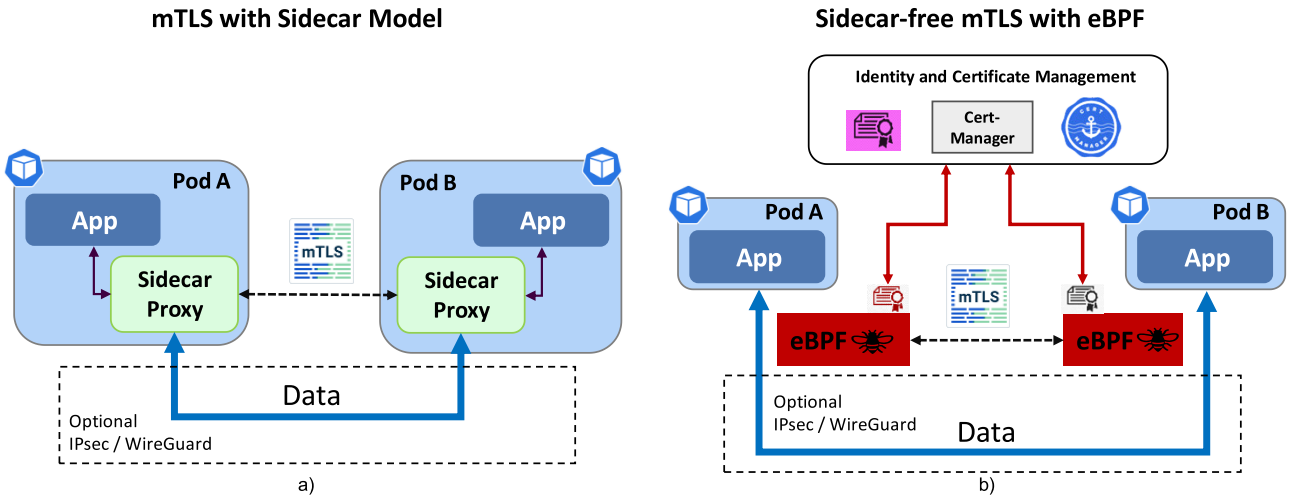


FIGURE 10. a) mTLS using Sidecar Proxy and b) Sidecar-free mTLS with eBPF [6], [15], [18], [21].

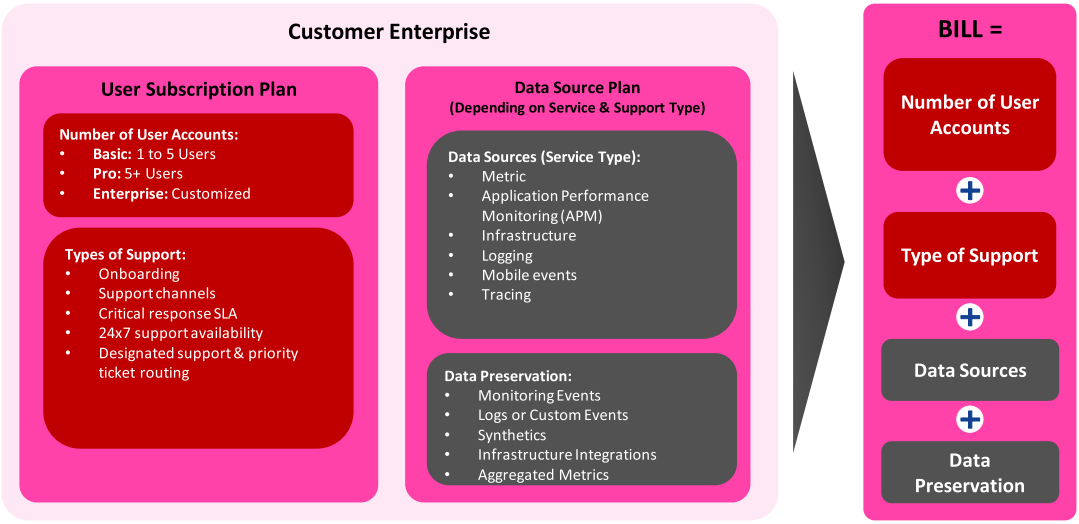


FIGURE 11. Example of charging and billing model for eBPF monitoring, observability, and security tooling [20].

The technology has reached its technical maturity. In fact, eBPF itself has become a markedly more general purpose, particularly due to eBPF for Windows. Also, a series of eBPF toolchains, such as *GCC* and *Aya* – a library that makes it possible to write eBPF programs entirely in *Rust* – have emerged.

D. HOW TO MONETIZE eBPF?

We may think of a marketplace where customers may access the software through subscription and use a product-based pricing and billing model based on a user subscription plan linked to specific products, including network observability, runtime security, energy, and a data source plan, depending on service and support type [20], as depicted in Figure 11.

The user subscription plan considers the number of user accounts and the type of support required. The data source plan is linked to a usage-based pricing model. The data reported to the eBPF agent are processed and then transformed into bytes (usage metrics) by means of transformation rules specific to the corresponding data source. If you are on the usage-based pricing plan, you will be charged for the number of bytes that exceed the free monthly allowance.

Another possibility is to charge based on resources, i.e., the number of deployed controllers (operators) and agents (daemonsets) and types of support (services), as shown in Figure 11.

E. eBPF LIMITATIONS AND COMPLEXITY

eBPF, like any other technology, has its own limits, and it is crucial to be aware of these limitations, and associated complications in developing different use cases.

eBPF can be a complex technology to work with, both in terms of writing eBPF programs and integrating them with other tools. Writing eBPF programs requires knowledge of eBPF language and its libraries, and a good understanding of the Linux kernel and its internals.

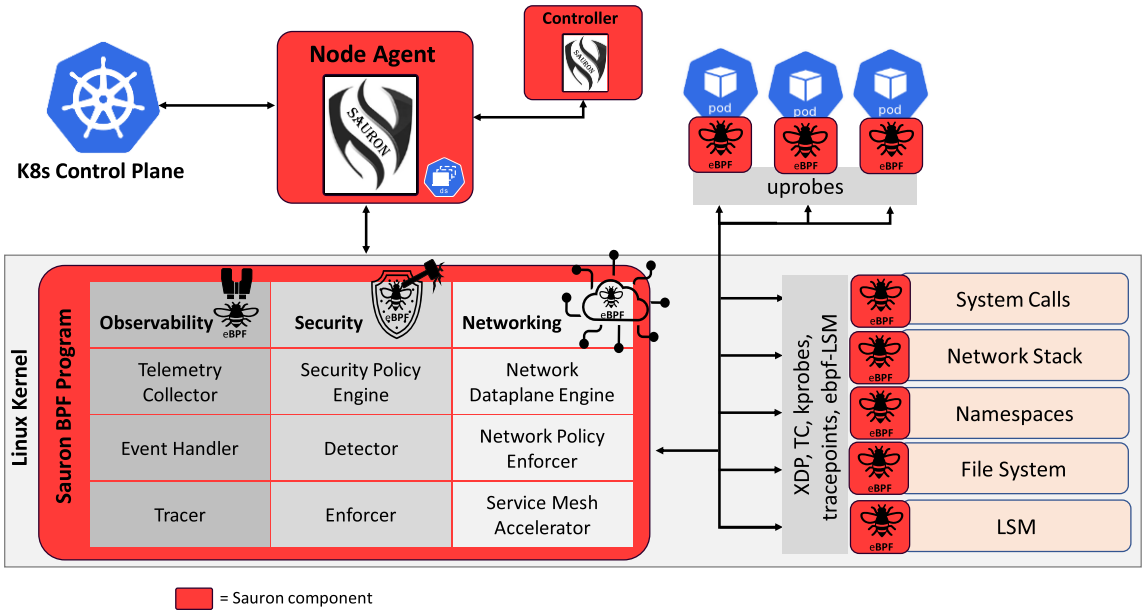


FIGURE 12. eBPF platform for telco and modules for observability, security and networking use cases.

Also, eBPF features availability is tightly coupled to the specific kernel version available on the host machine: the most advanced and valuable features could only be available in the most recent kernel versions, which in turn could not be available or installed in the actual production environment.

Besides, reaching kernel portability could be difficult. For example, it requires shipping the compilation suite (BCC approach) with the solution or BTF support (BPF Compile Once – Run Everywhere (CO-RE) approach).

Furthermore, since programmability is restricted, due to kernel safety reasons, solving some specific problems may require the rethinking of the development approach. The eBPF verifier is very complex, but still not complete: its versions limit the programmability a lot and generate many false positives. The developer must know where to find and use some ''hacks'', which complicate the code writing, to avoid some valid eBPF programs being rejected.

Another issue with eBPF is the *kprobe* interface of the Linux kernel. The *kprobe* interface allows eBPF programs to be attached to arbitrary hook points in the kernel, which can be used to monitor and control various aspects of the system. However, the choice of these hook points is critical, and it can greatly impact the effectiveness and efficiency of a host-based intrusion detection system (HIDS). For example, if the hook point is not in the right place, it might miss critical system events or introduce performance issues by consuming too many resources. Therefore, a profound knowledge of the Linux kernel is required to choose the right hook points to monitor the system effectively.

Moreover, eBPF can also generate a lot of data and events that need to be filtered, stored, and analyzed. This requires a thorough understanding of the system and data to extract the relevant information and take the right action.

Additionally, while eBPF is good at monitoring the kernel and system calls, its support for monitoring user space (enabled through *uprobes*) is less efficient than the kernel side counterpart and, indeed, introduces a non-negligible overhead. Many HIDS require monitoring and analyzing the user-space events and actions that cannot be handled by eBPF alone.

Beyond that, eBPF is effective at filtering and capturing system events, but it is not designed to perform complex correlations, analyses, or incident responses, which are frequently required in HIDS. Therefore, it is often used as a first line of defense that needs to be integrated with other systems and tools.

Lastly, a complex control plane is required to manage programs loading and unloading, attaching, and detaching, and configuration.

In the following sections, we present our eBPF platform for observability, and four case studies, to show how eBPF can be used for estimating energy consumption of cloud-native applications, derive performance counters and gauges for transport and 5G networks, and detect and respond to unauthorized access to API resources and files.

III. SAURON CONCEPT AND ARCHITECTURE

In this Section, we present our eBPF platform, named Sauron, and the corresponding eBPF modules supporting various use cases. The corresponding case studies and experimental results are discussed in Section IV.

The proposed eBPF platform for Telco comprises: a Node Agent (NA), i.e., a K8s *daemonset* deployed on each node, a Controller to handle the NAs deployed in the clusters, and three eBPF modules for observability, security, and networking use cases, as shown in Figure 12. Sauron does not

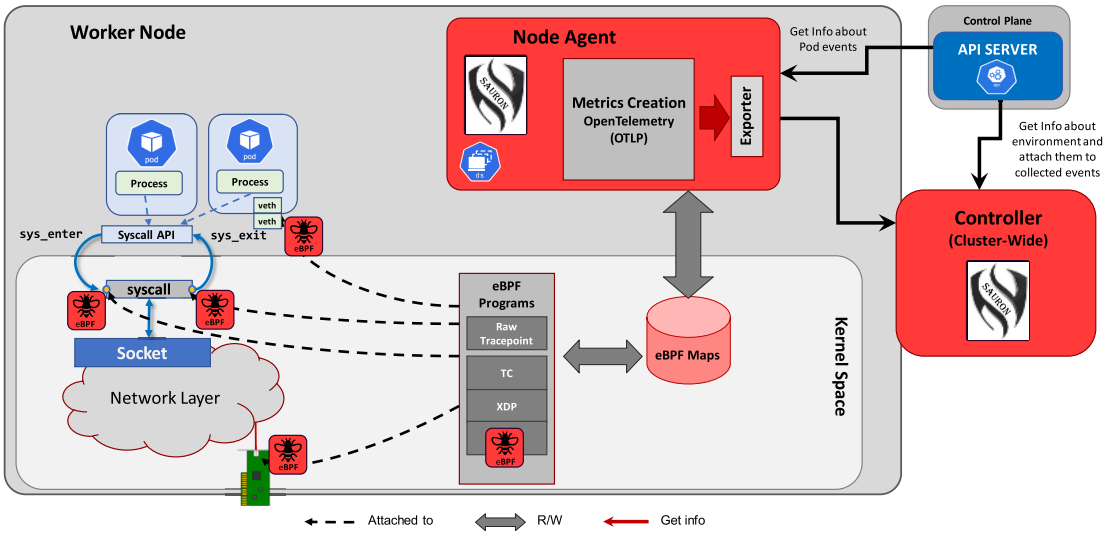


FIGURE 13. Sauron reference architecture for eBPF Observability.

require the installation of any specific K8s plug-in, such as Calico [19] or Cilium [21]. Examples of supported eBPF features are, but not limited to, the following:

- **Observability**: Telemetry collector, event handler and tracer.
- **Security**: Policy engine, detector, and policy enforcer.
- **Networking**: Data-plane engine, policy enforcer and service mesh accelerator.

Figure 13 shows how Sauron is implemented for network *observability*. We have one Controller for each cluster and one Node Agent (Sauron daemonset) on each node of the cluster. The Controller gets information about the cloud-native environment from the API Server and attaches metadata to the corresponding collected events. The Node Agent retrieves all relevant information about Pods (on the node the Agent runs) from the API Server; loads the eBPF probes into the kernel and attaches them to XDP hooks, system calls, and virtual interface(s) of each pod. Other hooks could be instrumented as well. As described in Section II, the eBPF Maps allow the communication between eBPF probes in the kernel space and Node Agent in the user space. More insights into our eBPF architecture, programs, interfaces, and supported protocols are provided in Section III-B.

In the following sections, we describe the eBPF reference architectures, platform building blocks, and interfaces for transport, energy, observability, and security tooling in detail. The related experimental validation is discussed in Section IV.

A. SAURON eBPF MODULE FOR TRANSPORT

The ability to analyze the provisioned network connections and performance of services is critical for telecom network management. Additionally, telecom networks are moving towards virtualization, and any network function is expected to be Virtualized (VNF) or Cloud Native (CNF) by design, meeting the requirements of CNCF [1]. This means the commercial off-the-shelf product (COTS) servers and network devices, such as, e.g., routers and switches, are the dominant factors affecting the network performance of a connectivity service provider (CSP) in terms of agility, resiliency, and high availability.

Moreover, the offering of hosting applications on the cloud has become a SaaS and IaaS business where the users of these applications consume the services via their portable devices, shifting the attention of network monitoring solutions to be cloud based. This progressive cloudification of networks and services is also making them more complex. In this scenario, *purely passive monitoring solutions are no longer sufficient to proactively detect connectivity issues*. And active probing has become a fundamental complementary monitoring technique to provide a complete hop-by-hop picture of the overall network performance and to detect devices or applications faults promptly [22].

In [23], eBPF was utilized for *passive measurement* of network latency to overcome the inefficiency of always-on passive monitoring to keep up with traffic as packet rates increase, especially on current multi-Gbps interfaces. The proposed eBPF-based solution that evolved Passive Ping (ePPing) delivered accurate RTT measurements and greatly improved on state-of-the-art software-based solutions, such as PPing, handling over 1 Mpps, or, correspondingly, more than 10 Gbps on a single core.

Conventional techniques for evaluating end-to-end (E2E) network behavior typically employ *active measurement protocols*, see RFC 792, RFC 4656, RFC 6038 in [24], which probe the network to gather information about its status, performance, and healthiness. These protocols rely on the generation of synthetic packets sent over the network to measure the time it takes to reach a specific destination and/or to receive a reply. Examples of active measurement protocols include:

1) Internet Control Message Protocol (ICMP)

TABLE 1. Summary of eBPF advantages and potentials [15].

#	As is	To be
1	eBPF speeds up development by decoupling from kernel releases	5G data planes with eBPF, IoT security enforcement with eBPF exist today
2	eBPF shifts data processing closer to the event source (per-socket hooks, pre-cgroup hooks, XDP, etc.), freeing up resources	APM and security monitoring platforms with eBPF at its core
3	eBPF allows shorter production feedback loops, decoupling from the kernel and allowing atomic program updates on the fly, and location-aware processing	XDP & eBPF L4LBs/gateways into the cloud, memcaches/accelerators, etc.
4	eBPF moves traffic with significantly lower latency by BPF host routing and BPF bandwidth manager	XDP and AF_XDP blends in naturally with Kubernetes, as opposed to DPDK
5	eBPF provides building blocks from the kernel, which are too complex for other kernel subsystems	Traffic engineering via eBPF and SRv6
6	eBPF (to some degree) fixes or mitigates kernel bugs on the fly	eBPF-based solutions are the preferred or default choice
7	eBPF enables low-overhead deep visibility and enforcement into the system, achieving significantly richer visibility, programmability, and ease-of-use than old-style perf	eBPF process scheduler (Google, Meta, Huawei) to customize, e.g., CFS for data centre workloads
8	eBPF decouples from legacy UAPI and allows efficient data processing	eBPF and IMA for file integrity protection/monitoring of system software and user applications
9	eBPF is an enabler to build policy enforcement features around stronger notions of identity	eBPF and XDP-like layer plus better observability for storage devices, e.g., around block layer and below
10	eBPF allows developers to extend the kernel but with a "safety-belt" on	Sustainable computing projects like Kepler exporting energy consumption data via eBPF for Kubernetes clusters

2) One-Way Active Measurement Protocol (OWAMP)
3) Two-Way Active Measurement Protocol (TWAMP)

Conventional tools (e.g., from vendors such as Juniper Networks, Accedian and others) based on those protocols for calculating the E2E latency of telecom networks can be useful for identifying connectivity and performance issues. However, they have some limitations, such as, but not limited to, the following:

- They are not always supported by all types of network devices.
- There is a necessity of being non-intrusive to the target network traffic, which mainly leads to a trade-off with the obtained accuracy.
- They do not always provide a complete picture of the latency experienced by all devices on the network.
- The calculated E2E measurements do not provide insights about the links with issues.
- They may not be suitable for measuring latency in real time (RT).

The following sections present how our eBPF solution supports latency measurements, overcoming the limitations introduced above, and its future developments.

1) eBPF SOLUTIONS FOR ACTIVE MEASUREMENTS

Transport network infrastructures are usually characterized by many network devices, which are typically organized in aggregation domains. Depending on the device location and/or type, the hardware characteristics of the devices can change, effectively making transport networks a set of heterogeneous nodes. However, despite these differences at the hardware level, nowadays, many operating systems on the network devices are mainly based on the Linux kernel, thus enabling a homogeneous platform at the software level. As a reference, Cisco IOS XR, IOS XE, and NX-OS families are all Linux-based. Other vendors such as Juniper Networks and Nokia have followed the same approach in some of their product series.

The Linux integration provides access to all the Linux utilities and tools. In this paper, we are leveraging the utilization of eBPF kernel technology. As mentioned in Section II, eBPF allows the running of custom programs at the kernel level without modifying the kernel source code or adding additional user space resources.

The eBPF programs can be attached to various hook points inside the kernel. Some of the hooks are strictly related to the network stack and capable of intercepting the traffic while it is being processed. For this specific use case, we leveraged the XDP hook, which allows eBPF programs to be executed at the earliest point of the packet processing path inside the network interface driver. In this context, the programs can inspect or modify the content of the packets, but they can also decide whether to redirect or drop them. Moreover, through eBPF helpers, they can perform more advanced actions such as *timestamping*.

These features make eBPF XDP programs suitable for many network applications. In this work, we focused on the problem of measuring the latency between network devices,

namely routers. We adopted an active approach for the generation of synthetic packets periodically. eBPF programs are not involved in this generation process; instead, they are responsible for the actual measurements of the latency experienced by these synthetic packets.

In contrast to other implementations, our solution does not require any special hardware support for clock synchronization or packet timestamping. These hardware capabilities are not always available on network devices, and in some cases, they are not accessible from custom applications. At the same time, our solution does not rely on any application-level mechanisms for measuring latency since all its logic is in the XDP program that runs at the kernel level. As a result, this solution represents a perfect combination of accurate measurement and minimum requirements. It achieves good quality measurements without any special requirement except for the Linux kernel version, which must include the support for XDP (it has been introduced in version 4.8).

As the resource utilization of these active measurement tools is a major concern, we would like to emphasize that the overhead of this solution is minimal in terms of CPU consumption. More details are provided in Section IV-A.

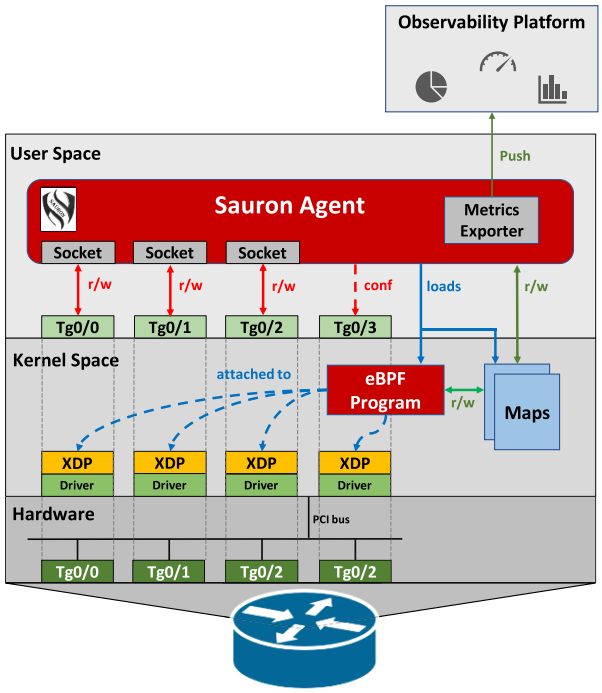


FIGURE 14. eBPF reference architecture for latency measurements.

2) eBPF ARCHITECTURE FOR TRANSPORT MONITORING

The proposed solution consists of two key components: a Sauron Agent and an eBPF program, as illustrated in Figure 14, where the Sauron Agent is deployed on a router.

Sauron Agent runs as a Linux native app on the network device. It provides an interface through which a user can configure the desired latency measurements; it collects and exports the metrics obtained, and it is responsible for the setup of the eBPF program. In particular, the agent loads the program in the kernel and attaches that to the XDP hook of all active data plane interfaces. Moreover, it creates eBPF Maps by passing some parameters to the program.

The Sauron Agent is also the component that generates, sends, and receives the synthetic packets that are used for measuring the latency. To do that, it simply leverages the traditional socket interface. Additionally, the generated packets can be configured to have specific characteristics, such as *DiffServ class* and *payload length*.

The eBPF program, on the other hand, processes the synthetic packets and extracts all the data needed for the latency measurements. In the XDP hooks, the program is triggered every time a packet arrives at a network interface. In this context, it can read and modify the packet, but it can also decide whether to redirect that to a different destination. The next section describes how these capabilities are utilized for latency measurements.

3) HOW eBPF IS USED FOR CALCULATING LATENCY

The process for measuring the latency is divided into three phases. The first is the generation phase, in which the Sauron Agent generates synthetic packets. By default, these packets are broadcast via all active interfaces in the router, where the Sauron Agent is deployed, to the neighbors directly connected to the router in question. To reach the remote peers – beyond the neighbors directly connected to the router where Sauron runs – their IP addresses need to be explicitly specified in the configuration file of Sauron.

When a synthetic packet reaches the interface of (any) peer device, the second phase of the process begins. This process is performed at the XDP level and encompasses the bouncing of the packet between the receiving device and the original sender for a fixed number of times. The program simply utilizes an XDP feature that allows us to redirect a packet through the same interface at which it was received. At each step, a *timestamp* of the time of occurrence of each action (send or receive) is added to the packet payload. In Figure 15, the dashed blue lines highlight the path the packet takes during the process. In this case, the router on the left is the initiator.

After the final iteration, the original sender breaks the loop, passes the packet to the networking stack, and reaches the Sauron Agent through the socket interface. The third and last part of the process takes place in the user space. The Sauron Agent extracts all the necessary information from the payload to make the latency measurements. Using the collected timestamps, it calculates both One-way and Two-way latency. Once these metrics have been calculated, they can be exported to a remote collector for processing and visualization.

4) WHAT IS NEXT?

Most of the "in-network" functions, such as routing, switching, load balancing, and Quality of Services (QoS) management, support the improvement of the performance and efficiency of the network using specialized hardware.

We believe that eBPF implementation on Telco networks will provide operators with the desired ability and flexibility to build programmable "in-network" monitoring functions and tooling that dynamically instrument the Linux kernel with minimal overhead.

B. SAURON MODULE FOR NETWORK OBSERVABILITY

The latest generation of mobile networks, namely 5G and future generations, have already adopted cloud-native architectures as standard, and cloud-native technologies will be the basis and foundation of network infrastructures. The migration of mobile communication networks to cloud-native architecture poses some challenges but at the same time offers substantial benefits, unquestionably. One of the key challenges is the instrumentation to achieve visibility or observability of the network performance near real time and at the session level, which is where eBPF can play a fundamental role.

1) eBPF SOLUTIONS FOR NETWORK OBSERVABILITY

In any network providing services to customers, including mobile networks such as 5G and beyond, a fundamental aspect and requirement is instrumentation enabling near real-time *performance monitoring*, *service assurance* and *subscriber experience monitoring*.

The key source of truth, providing the means of extracting required information and insights for achieving the required observability or visibility near real time, are the packets traversing the network (packet capture), or messages/information exchanged between different network functions. These messages or packets are typically collected at various standards-defined interfaces between the network functions [25]. Collecting information from these interfaces on both the control and user plane parts of the network is required with their complex correlation to achieve the required observability.

In traditional networks of previous generations deployed on physical proprietary hardware or even in VMs, various techniques were used to capture the required information. Most, if not all, of these are no longer adequate/possible or at best, significantly inefficient in a cloud-based architecture and, if applied, require a compromise in the architecture of the next generation of mobile networks, including 5G.

Traditional means of instrumenting and extracting the required information (e.g., packet capture) across different parts of the network consisted of utilizing various techniques ranging from *port mirroring*, *network packet brokers or physical tapping*, collection of *traces,* e.g., from the Radio Access Network (RAN) or Core Network. However, these means were often not consistently applied across various components of the network; they include, but not limited to, the following:

- Dependent on the network architecture and the capabilities provided by the vendors in their network elements, e.g., there is a significant difference between RAN cell traces provided by different vendors (almost all are limited to only control plane data).
- Intrusive, e.g., requiring the deployment of packet brokers and additional physical devices and components with various vendor-specific configurations (requiring support and expertise).
- Being limited by port spanning or traffic mirroring capabilities and capacity, which adds overhead and additional cost.
- Not easy to scale or deploy, monitoring new interfaces or network functions entails introducing new components or additional capacity and complex configuration, which is a lengthy process.
- Limited to monitoring only interfaces between network elements or functions.
- Not readily extensible and *programmable*.

Other means/methods of instrumentation, such as open tracing/open telemetry were introduced relatively recently to address some of these challenges [26]. However, they still have significant limitations and are typically invasive to the network and applications. A detailed description of which is beyond the scope of this paper, but a key limitation is varying implementations and the extent of what is supported.

In a cloud-based network architecture, such as the one employed in Rakuten Mobile Japan and being adopted by many other mobile network service providers, a different and novel approach is required that is cloud native, consistent throughout the network, easy to scale and deploy, and that is programmable. This is where eBPF comes into play.

In [27], the introduction of non-intrusive data collection for network monitoring at kernel level is proposed for Kubernetes cluster for an e-commerce platforms container services based on eBPF, supporting very high throughput and with minimal impact or overhead on the system (less than 1%). The solution presented in this paper, applied to a mobile telecommunications network running on a cloud platform using Kubernetes for the orchestration of containers, although similar, goes beyond the approach proposed in [27]. In our work, for network observability using eBPF programs running in the kernel, we propose a non-intrusive and *protocol-independent* data collection mechanism.

Furthermore, the ideas presented in this paper meet the requirement for fully adaptive and programmable network monitoring. Drawing from similar approaches as described in [28], we propose a highly programmable eBPF solution that allows for efficient, dynamic, and granular monitoring pipelines (i.e., data collection mechanisms) with minimal overhead or impact for Telco networks.

In short, our eBPF solution complements and expands the scope of the works published in [27] and [28] to modern Telco infrastructures running on cloud-native platforms. Our focus is on the application of the eBPF technology, for observability/monitoring, security, and energy, specifically to the latest generation of mobile communication networks deployed on

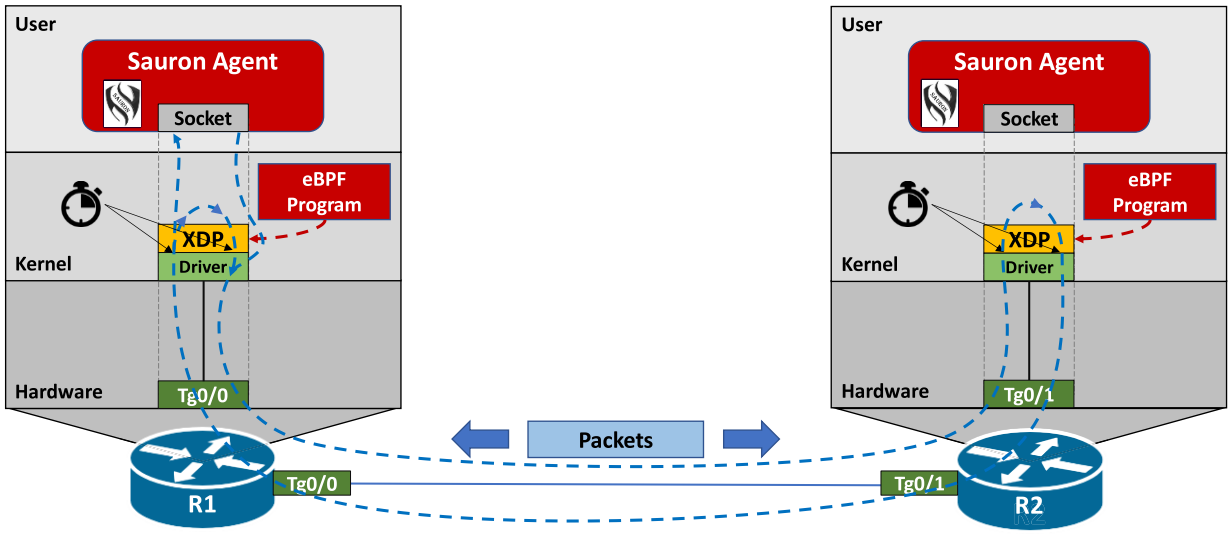


FIGURE 15. eBPF reference architecture for latency calculation.

a cloud-native environment utilizing container and container orchestration platforms.

2) eBPF ARCHITECTURE FOR NETWORK OBSERVABILITY

Figure 16 depicts a high-level generic logical diagram of a 5G Core Network (5GC) instrumentation that utilizes eBPF programs controlled by a daemonset agent (Sauron Agent). The diagram is derived from the eBPF platform reference architecture shown in Figure 13. In the figure, the generic pods effectively represent the 5G cloud-native network functions or related network micro-services, depending on the containerization.

In the high-level logical diagram illustrated in Figure 16, the daemonset (ds) agent, i.e., the Sauron Agent, is deployed on each node in the cluster. Sauron detects containers and pods of the 5GC network by querying the API Server. Also, the agent configures the necessary eBPF programs, collects the messages using a Ring buffer or throughout eBPF Maps, parses, and extracts information from the protocol messages. The agent can be configured to collect information from all parts of the network (radio access and core domains) with minimal overhead and process them, thus removing vendor-specific cell trace configurations and avoiding their inherent limitations and dependencies.

Also, the eBPF solution enables simple traffic forwarding to an existing collector or probe component of a sophisticated service assurance and/or subscriber experience tooling, which might be already in use, and to perform complex correlation of data for call or session tracing and analysis, packet analysis, and for generating metrics and key performance indicators (KPIs).

An agent configured to simply forward traffic is depicted in Figure 17, where 1) A *libpcap* probe is installed on the interface. 2) Packets pass through the BPF filter installed on the interface (e.g., *src* 10.0.0.1 or *dst* 10.0.0.1 and not *src* net CLUSTER_CIDR); 3) Matching packets flow to the agent; 4) The agent encodes incoming packets in PCAPNG format and 5) sends them to the processing probe.

Furthermore, it is important to note that the described solution is programmable and can be configured not only for continuous observability but also on demand to monitor different parts of the network or provide different levels of observability as required and, in that way, optimal for performance and minimal overhead. Furthermore, such configurations could be automated using AI/ML algorithms or models that detect changes in traffic patterns, anomalies, and utilize other network data sources, such as PM/FM/CM etc.

The proposed eBPF solution can provide an additional level of detail that is not supported by any traditional solution. That is, 5G network functions deployed in a cloud environment will typically follow a microservices-based architecture. The proposed solution could be configured to provide details of the communication or information exchanged between microservices within the network function, providing another level of context in network observability.

Finally, the solution has no dependency on those (vendors) providing the network functions deployed in a cloud platform. It can automatically scale up and down with the network, ensuring optimal and energy-efficient use of resources for observability, end to end.

C. SAURON eBPF MODULE SECURITY

As Kubernetes gains momentum, organizations adapt how their security teams detect and respond to threats. In contrast to conventional environments, the workloads and resource requirements in a containerized (virtualized) environment are constantly changing, posing new challenges for security teams and runtime security monitoring solutions. Since it can be difficult to detect and respond to threats in a timely manner when infrastructure is in a constant state of change, a new generation of tools has evolved, where eBPF plays a significant role as an enabling technology.

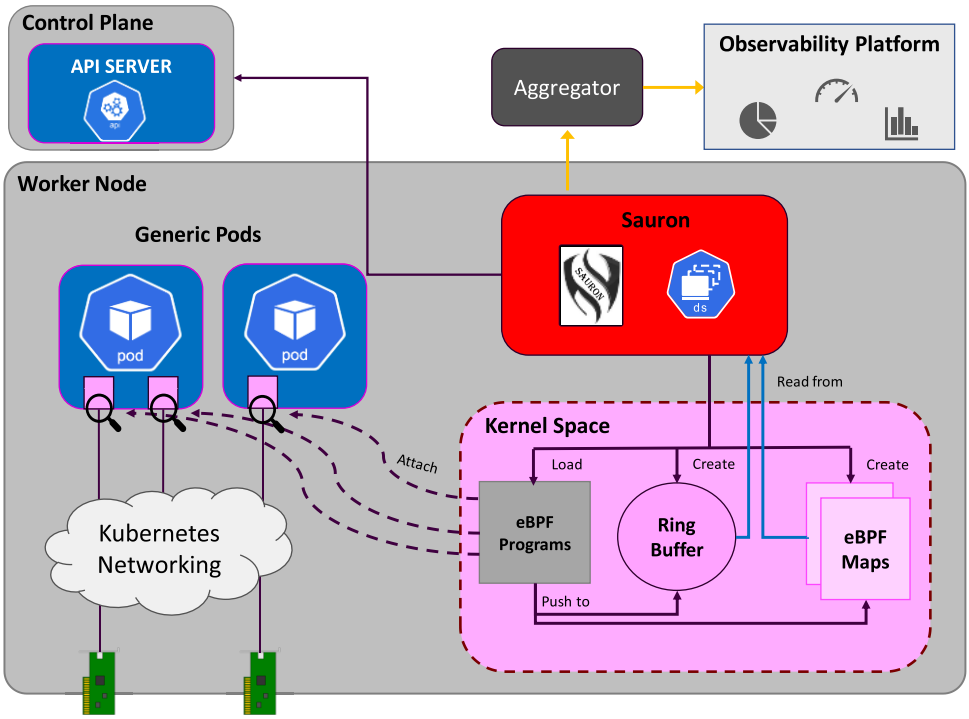


FIGURE 16. Sauron reference architecture for 5G network functions (generic pods in the figure).

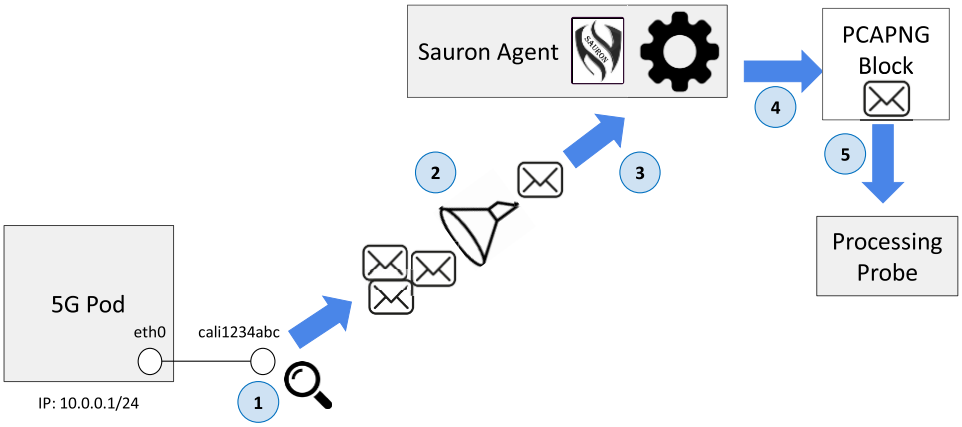


FIGURE 17. Agent configured to simply capture and forward traffic.

1) eBPF SOLUTIONS FOR NETWORK SECURITY

By providing visibility into low-level system and network events, eBPF is a powerful technology that can be used to enhance security in cloud-native environments. There are many open-source tools that leverage eBPF to detect and prevent malicious activity, which can be grouped into two broad categories: *network security* and *application runtime security*.

Monitoring and inspecting network traffic at various layers of the stack, such as the kernel, eBPF offloading on the network interface, and application layers, is the primary focus of eBPF-based network security tools. These tools can be used to detect and prevent various types of network-based attacks, such as distributed denial of service (DDoS), port scanning, and other types of reconnaissance. Cilium [21] and Polycube [29] are examples of eBPF-based network security tools.

Application runtime security tools utilizing eBPF typically focus on monitoring and inspecting the behavior of host-running applications and processes. These tools can detect and prevent a variety of application-level threats, including privilege escalation, code injection, and other forms of malware. Falco [30] and Tetragon [31] are examples of application runtime security tools that employ eBPF.

2) eBPF ARCHITECTURE FOR NETWORK SECURITY

Despite the limitations introduced in Section II-E, eBPF is a potent technology that can be used to enhance security in various scenarios, and it continues to evolve and acquire new capabilities. In addition, its real-time nature and capability

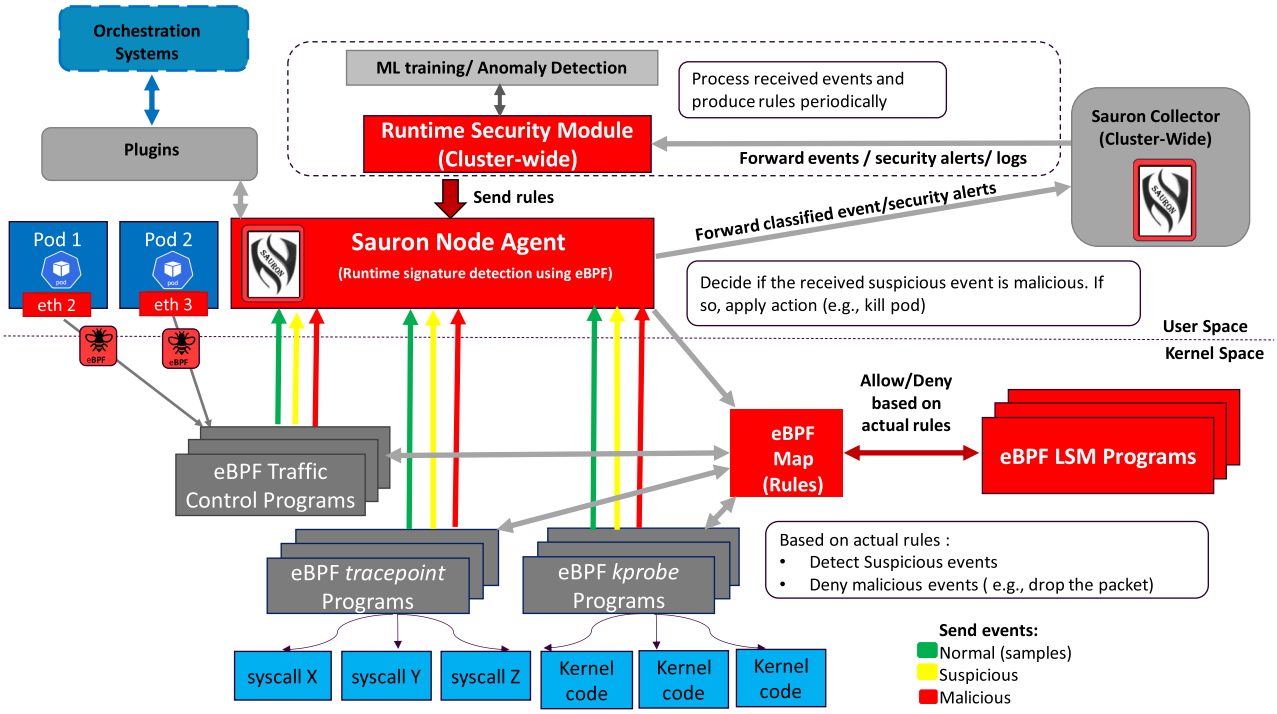


FIGURE 18. Sauron reference architecture for eBPF runtime security monitoring.

to intercept, filter and redirect traffic and monitor and report events such as *syscalls* on-the-fly, make it especially suitable for network security monitoring and enforcement, runtime security and incident response.

Figure 18 illustrates our eBPF reference architecture for network security. The main aim of the proposed solution is to provide *eBPF runtime security monitoring* to detect and mitigate malicious activities in cloud-native environments.

The solution comprises an agent named Sauron deployed on each node of the cluster. The agent is responsible for loading eBPF programs into the kernel and attaching them to specific hooks. Different types of eBPF programs are used for providing various capabilities: among these, the most relevant are *TC*, eBPF *tracepoints*, *kprobes* and eBPF *LSM*.

TC programs are used to implement network security (an alternative would have been the use of sk_skb programs): the primary role is to inspect packets flowing in and out pod boundaries (traversing virtual ethernet interfaces) or between containers in the same pod, detecting policy violation and enforcing configured rules on the unwanted traffic. Unwanted traffic could be:

1) Traffic directed to or coming from subjects (entities) that are not allowed to communicate with the object pod.
2) Malicious traffic (malformed packets) directed to or coming from sources that are allowed to communicate with the object pod but not authorized to perform certain tasks.
3) Lastly, malicious traffic that insists on a disallowed communication path.

eBPF tracepoints programs are attached to static markers defined by kernel developers in the kernel code. The more recent the Linux kernel version is, the more efficient these kinds of eBPF programs (e.g., eBPF raw *tracepoints* and eBPF btf-enabled raw *tracepoints*) are. Moreover, static markers implementing the hook points guarantee the Application Binary Interface (ABI) for *tracepoints* to be more stable than the one available for *kprobes*: this could be leveraged to implement a more stable security solution. *Tracepoints* can be used to track events related to multiple subsystems, including *sched*, *netlink* and *system call*, even though Figure 18 shows only the system call subsystem. The system call subsystem allows installing eBPF programs to be triggered on entering or exiting specific system call handlers. System call boundaries are useful for detecting unwanted filesystem interactions: by attaching eBPF programs on *open*, *close*, *write*, *read*, *dup*, *fcntl*, and so on, it is possible to track file access and modification by any process, optionally running in a container inside a pod.

Since *tracepoint* static markers are not available for all events of interest, *kprobe* can be used to complete the monitoring surface. Even though *kprobes* are not as effective as *tracepoints* and do not provide a stable interface, they are essential for achieving monitoring completeness; moreover, they allow the implementation of an additional layer of defense beyond what *tracepoints* provide. Lastly, by tapping deeper into the system calls implementation, it is possible to circumvent the well-known Time-Of-Check Time-Of-Use (TOCTOU) vulnerability of *tracepoints*.

eBPF LSM programs add an additional layer of security to newer kernels. Besides allowing the detection of malicious

behaviors, they allow the denial of access and permissions directly in the kernel, enabling faster response to the one that can be provided via *tracepoints* and *kprobe* programs.

The user space agent provides all rules for detecting malicious behavior and enforcing policies to eBPF programs via eBPF maps. eBPF programs interact with the agent via the eBPF perf buffer or the eBPF ring buffer (depending on which feature is available in the kernel installed on the node). Programs send three kinds of data to user space: samples of normal events, suspicious events, and malicious events.

- *Normal/malicious events* are events recognized as permitted/denied directly by eBPF programs based on the accessible information (rules configured into eBPF maps and event-surrounded context). Malicious events and (sampled) normal events are sent to user space to enable data collection, which in turn enables AI/ML model construction. Moreover, malicious events are sent to user space to enforce related policies that cannot be enforced directly in the kernel (e.g., *pod killing*), or that cannot be enforced by the specific detecting eBPF program type (e.g., network message dropping in *tracepoint* programs)
- *Suspicious events* are events that cannot be directly classified into eBPF programs due to the lack of information; therefore, they must be sent to the user space for further analysis.

The *user space agent* classifies suspicious events as either benign or malicious events. Then, all events collected by the agent (including the suspicious events once they have been classified) are sent to a cluster-wide events collector.

The *collector* stores the received events and makes them accessible to a cluster-wide runtime security module, which has two responsibilities: building AI/ML model by leveraging the collected events and providing each node agent with enforcing rules inferred from the model.

By providing a plugin-based interface to third-party systems, the system can be integrated with them.

D. SAURON eBPF MODULE FOR ENERGY

Energy management in Telco networks and cloud computing environments is motivated by climate protection, resource conservation and cost savings. In such networks, it is important to calculate the energy consumption per each element in the network to manage the energy consumption.

The most important hardware elements with respect to power consumption are the CPU and GPU. The power consumption of a CPU comprises three main factors, namely dynamic power consumption, short-circuit power consumption, and power loss due to transistor leakage currents.

The dynamic power consumption is due to the charging and discharging of the capacitors within the logic gates that shape a CPU. It is linearly proportional to the CPU frequency, as well as the switched load capacity, and to the square of the CPU voltage.

The short-circuit power consumption is caused by a direct path between the source and ground which is accidentally created for a very short time when the transistors inside the logic gates are changing their states. Modelling the short-circuit power consumption is a complex task, and it needs to be done per each logic gate.

Leakage currents are the currents flowing between semiconductor regions of the transistor that were doped differently. These currents are a function of the physical properties of the transistors, such as their size and state. Although their individual amount is small, the total amount of power dissipation due to transistor leakage currents is an increasing function of temperature and a decreasing function of transistor sizes.

Another essential element to consider is the GPU. The GPU power dissipation might be critical for 5G systems because some network functions run on the GPU. Indeed, a GPU can enhance the performance of 5G signal processing, which is more advanced than in previous generations. The main factors that determine the power consumption of GPU include GPU clock speed, GPU voltage, GPU architecture, cooling solution and GPU power limits.

1) eBPF SOLUTIONS FOR ENERGY MONITORING

Direct measurement of dynamic power consumption, short-circuit power consumption, and transistor leakage currents power loss is not possible in a scalable manner, and it is only feasible to measure energy consumption at node-level granularity. Therefore, finding a scalable and economical alternative is very important for continuously monitoring energy consumption in a large network.

One possible solution is to approximate the capacity and execution time using the number of CPU instructions and CPU cycles, respectively. Using these approximations, one can characterize the power consumption of processes and consequently the power consumption of containers and pods. This is where eBPF enters the picture as a tool with minimal overhead for probing the CPU performance counters and Linux kernel *tracepoints*.

As already pointed out in Section II, KEPLER is one such tool that adopts this approach [14]. It is designed for cloud computing environments with Kubernetes as their container orchestration system. KEPLER uses eBPF programs to collect system and process information (such as *process id*, *cgroup id*) and performance counters from the kernel on the CPU, GPU and RAM (such as *cpu cycles*, *cpu time*, *cpu instructions*, *cache misses*), and then feeds them, along with other stats from *cgroup*, *sysfs* and other hardware counters, into ML models to estimate the instantaneous energy consumption of each process. KEPLER also uses Running Average Power Limit (RAPL) counters to read energy consumption and because of that KEPLER works only partially in a Virtual Environment due to the lack of RAPL counters. To overcome this drawback, KEPLER estimates energy consumption based on the pre-trained ML models.

By exporting this information as Prometheus metrics and aggregating them, KEPLER provides the energy consumption per container, namespace, pod, and node in a Kubernetes cluster. This level of granularity enables the operator to determine the energy consumption per CNF. KEPLER architecture is depicted in Figure 19.

Other tools that utilize the output of KEPLER are Power Efficiency Aware Kubernetes Scheduler (PEAKS) [32] and Container Level Energy-efficient VPA Recommender (CLEVER) [33]. PEAKS uses metrics exported by KEPLER to schedule Pods to achieve optimal performance per Watt, and CLEVER employs metrics exported by KEPLER to recommend Vertical Pod Autoscaler (VPA) resource profiles to improve energy efficiency by running workloads. Tools such as PEAKS and CLEVER do not utilize the eBPF by themselves but build the intelligence on top of KEPLER measurements. They are general-purpose tools for Kubernetes clusters, and further customization is required for developing any optimization instrument for Telco networks that employ eBPF.

Other examples of energy optimization tools for Kubernetes clusters that do not necessarily employ eBPF are, but not limited to, the following:

1) Cloud Carbon Footprint is a tool to estimate energy use and carbon emissions from public cloud usage [34].
2) Kube-green is a Kubernetes add-on that automatically shuts down (some of) the resources when they are not needed [35].
3) Kube-downscaler scales down or ''pauses'' Kubernetes workloads during non-work hours [36].

It is worth mentioning that CNCF TAG Environmental Sustainability [37] launched in May 2022, is a Technical Advisory Group (TAG) that defines environmental sustainability factors for the cloud-native landscape, incubate and advocate open-source projects to observe and measure cloud-native infrastructure carbon footprint, optimize, and eventually reduce carbon footprint and promote cloud-native infrastructure to combat environmental challenges.

The final aim of any cloud-native energy management tool is to provide *energy-aware pod scheduling and node tuning capabilities*. eBPF can serve as a reliable and lightweight tool for observing some of the metrics in the Linux kernel. However, the rest of the intelligence for managing the observed data and decision-making should be built independently and often needs to be customized to that environment or application.

2) MONITORING ENERGY IN OTHER NETWORKS

WiFi networks are another arena where the eBPF has been leveraged for monitoring energy consumption. In [38] and [39], the authors proposed a framework that utilizes eBPF to enable the access points to track the energy consumption and duty-cycling pattern of their associated stations without any external energy measurement tools. They achieved this by relying on the fact that stations need to inform their associated access point (AP) whenever they change their power mode. Therefore, monitoring the driver-pertaining data structures allows for tracking stations' duty cycle patterns.

The architecture of the framework proposed in [38] is depicted in Figure 20. The right side of the figure presents the networking stack, and the left side shows the Network State Monitor (NSM) module that relies on eBPF programs to interact with the networking stack. The dotted lines with arrows denote the collection of monitored data. The solid lines with arrows represent the path taken by data packets (wired-to-wireless switching data path).

IV. EXPERIMENTAL VALIDATION

This section describes four case studies and validates the technical feasibility of the related eBPF modules for transport, observability (including energy) and runtime security presented in Section III.

A. CASE 1 – eBPF FOR TRANSPORT NETWORK LAYER PERFORMANCE MONITORING

We validated our eBPF-based solution by deploying it on real routers using two different testbeds. All the routers that we used were from the Cisco NCS 540 family.

The first experiment was conducted using a simple network configuration with two routers and two VMs, as depicted in Figure 21.

The agent was deployed only on the routers, while the VMs were used to generate traffic and increase the traffic volume to put the routers under pressure. All traffic flowing from one VM to the other was forced to pass through the two routers using only one link between them.

The performed tests focused on measuring the latency perceived by the two routers as the traffic load on the link in between was changing. Results showed that our solution correctly detects the latency variation in real time, demonstrating that it is possible to keep the status of the links always under control.

Figure 22 shows the output metrics of the latency measurements calculated by the agent. It provides the current value for the One-way and Two-way latency as well as the mean value and the jitter. In addition, each measurement is associated with some metadata, such as the timestamp, the peer's name, and the interface at which it is reachable, and additional information related to the characteristics of the packet, such as the Differentiated Service Code Point (DSCP) value and the length of the payload in use.

For generalization, we implemented our solution for calculating the E2E latency on eight routers, as illustrated in Figure 23. This configuration allowed us to validate our solution in a more complex scenario, where we could measure the latency between network devices that were not directly connected, and the route between them was not fixed.

As shown in Figure 23, our agent considers two routers in communication as ''Local Peers'' if they are adjacent and as ''Remote Peers'' if they do not share the same link, i.e., there are multiple hops between them. We want to emphasize

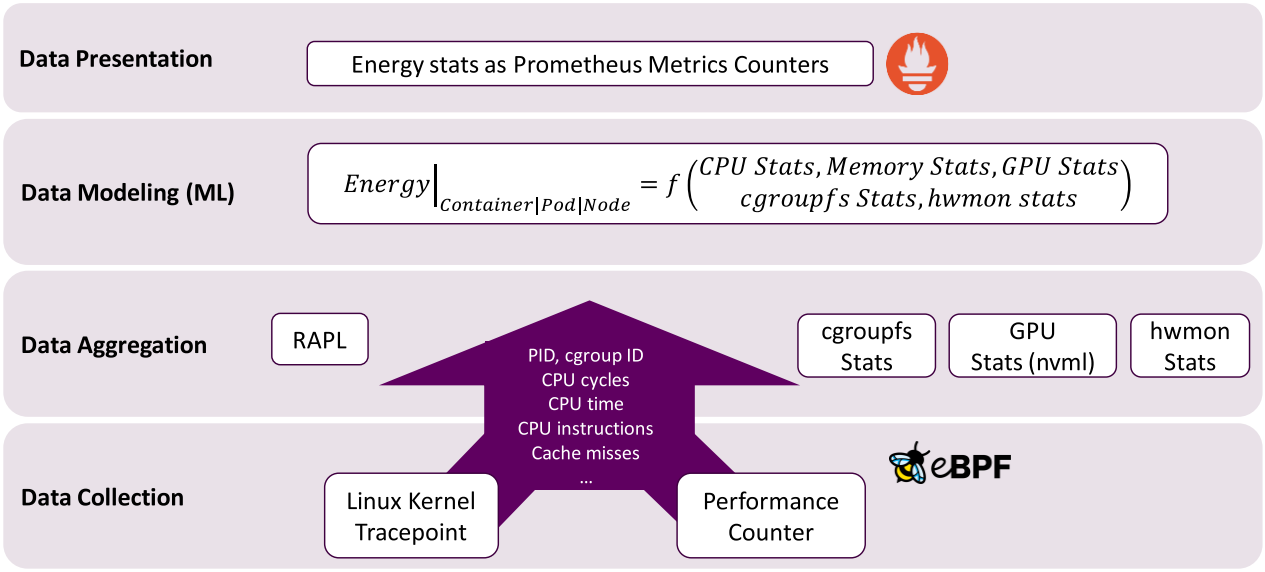


FIGURE 19. KEPLER reference architecture.

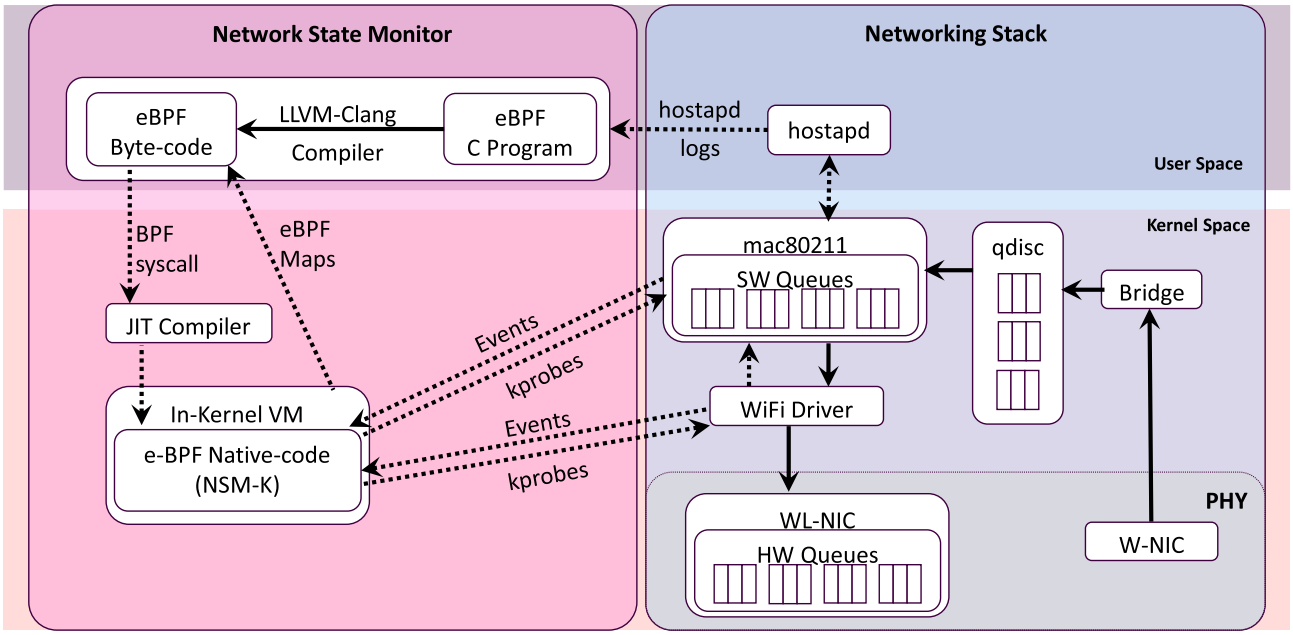


FIGURE 20. The architecture of FLIP framework proposed in [38].

that, by default, our solution measures latency for local peers. To calculate it for remote peers, the user needs to explicitly configure them in Sauron by providing their hostname or address, as explained in Section III-A.

Figure 24 shows the results attained (latency measurements) for two remote peers (R1 and R8). The outputs are like for ''Local Peers'', but for ''Remote Peers'', the agent displays the route (path) of the packet chosen from the sender (R1) to reach the remote peer (R8). This helps to display the measurement path, considering that multiple routes are supported.

Moreover, we assessed the resource consumption of the proposed solution in terms of CPU utilization. The NCS 540 routers were powered by an IntelI PentI(R) CPU D1519

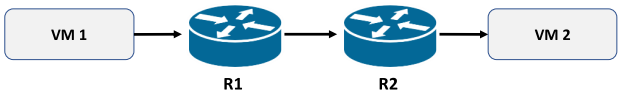


FIGURE 21. Measurement setup for testing eBPF for transport network layer performance monitoring.

@ 1.50GHz CPU. We measured the CPU usage as a function of time between measurements (period) and the number of peer entities. As shown in Figure 25, the CPU footprint is very low, especially when the time interval between two measurements is higher than 0.1s, and it increases exponentially as the measurements are made more frequently. This tradeoff must be considered in the actual implementation (production environment).

```
Timestamp: 2023/01/20 20:12:52
Local Peer Metrics: R1@Tg0_0_0_1 → R2
Traffic Class: 32 (DSCP)
Payload Len: 100 (byte)
Num Measures: 6
Lost Packets: 0
TW Latency: Current 0.053 ms Mean 0.063 ms Jitter 0.007 ms
OW Latency: Current 0.026 ms Mean 0.031 ms Jitter 0.003 ms

Timestamp: 2023/01/20 20:12:53
Local Peer Metrics: R1@Tg0_0_0_1 → R2
Traffic Class: 32 (DSCP)
Payload Len:  100 (byte)
Num Measures: 7
Lost Packets: 0
TW Latency: Current 0.059 ms Mean 0.062 ms Jitter 0.006 ms
OW Latency: Current 0.030 ms Mean 0.031 ms Jitter 0.003 ms
```

FIGURE 22. Examples of latency, jitter, and packet loss measurements between local peer entities.

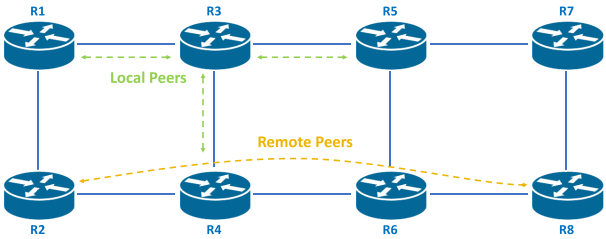


FIGURE 23. Extended measurement setup for testing latency, jitter, and packet loss between remote peer entities.

```
Timestamp:  2023/01/20 19:56:57
Remote Peer Metrics: R2@Tg0_0_0_0 → R8
Traffic Class: 0 (DSCP)
Payload Len: 100 (byte)
Num Measures: 7
Lost Packets: 0
TW Latency: Current 0.094 ms   Mean  0.105 ms   Jitter  0.065 ms
OW Latency: Current 0.050 ms   Mean  0.052 ms   Jitter  0.032 ms

Timestamp:  2023/01/20 19:56:57
Remote Peer Path: R2 → R8
Traffic Class:   0 (DSCP)
Distance:        3 (Hops)
R2 → fc00:1000:2::1 → fc00:1000:12::1 → fc00:1000:14::1
         (R4)               (R6)               (R8)
```

FIGURE 24. An example of latency measurements for remote peers.

B. CASE 2 – eBPF FOR PERFORMANCE MONITORING OF 5G PROTOCOLS

Our observability solution concept was validated in a 5G testbed deployed in a Kubernetes cluster running on the Google Cloud Platform (GCP). The 5G testbed is shown in Figure 26.

The three components of the testbed were:

1) 5GC functions, from Free5GC [40].
2) User Equipment (UE) & gNodeB (GNB), from UERANSIM [41].
3) Sauron Agent running on each worker node in the cluster.

The protocol stack supported by our open-source testbed is illustrated in Figure 27. The experiments were limited to observing the control plane of the 5G communication.

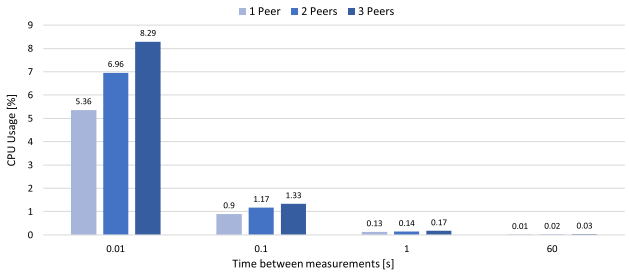


FIGURE 25. CPU utilization as a function of time between measurements (period) and number of peer entities.

In the first part of the validation, we focused on calculating *gauges* and *counters* to assess the performance of the next-generation application protocol (NGAP) and non-access stratum (NAS) messages in real time. In the second part, we tested the ability of Sauron to capture (using *libpcap*), filter (using BPF) and forward data in PCG.

1) NGAP AND NAS PERFORMANCE MONITORING

NGAP and NAS are layer five protocols, with complex message structures (composed of different optional parts). NAS messages are encapsulated into NGAP messages, and NGAP relies on the stream control transmission protocol (SCTP) of the 5G transport network layer. To extract information from NGAP messages, we need to parse SCTP packets containing multiple data chunks that can be fragmented, as shown in Figure 28.

XDP and *TC* hooks act at the packet level, not the message level, making them unsuitable for handling NGAP messages. Hence, to have direct access to messages, we instrumented *trace point* hooks (available in Linux kernel version 4.7 or later releases) by attaching eBPF programs to *BPF_PROG_TYPE_TRACEPOINT*.

Examples of NGAP and NAS metrics, derived using the approach presented above, are:

- NGAP Initial Context Setup Counters.
- NGAP Setup Success Ratio.
- NGAP Setup Time (ms).
- NGAP Procedure Duration (ms).
- NGAP Failed Procedures Cause Code
- PDU Session Establishment Attempts.

Some of the performance indicators displayed by Sauron in real time are shown in Figure 28, namely the NGAP procedure duration on average (0.073 ms) and the PDU session establishment request counter (1).

2) PACKET CAPTURING AND FORWARDING

For the experimental validation, we used the same 5G testbed on GCP as for the previous use case (see Figure 26 and Figure 27), and the reference architecture for packet capturing and forwarding is illustrated in Figure 17. The traffic generated by UERANSIM (see Figure 26), using a *libpcap* probe attached to the 5G pod (eth0), as shown in Figure 17, filtered using BPF programs, forwarded to the Sauron Agent, where it

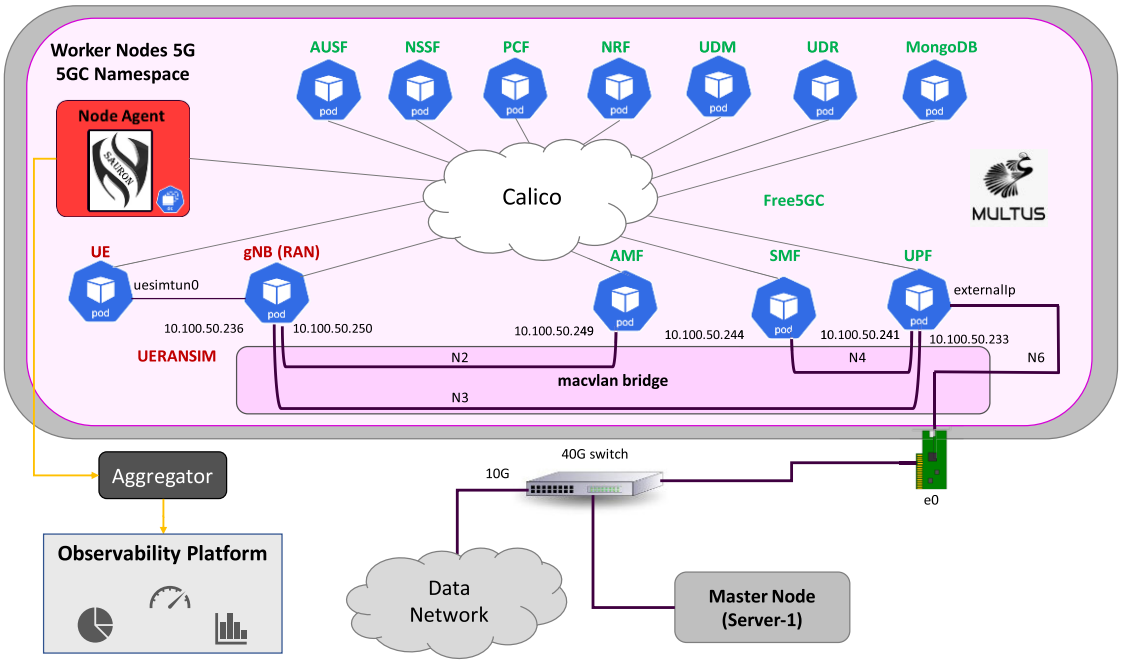


FIGURE 26. UE-RAN Sim (UE, gNB), Free5GC (AMF, SMF, UPF, AUSF, NSSF, PCF, NRF, UDM) and Sauron Agent running in a Kubernetes cluster.

is encoded in PCAPNG format, and then passed to an external processing probe for further processing.

The encoded packets as PCAPNG were thereafter replayed using Wireshark and tested at Rakuten Mobile Inc. for further validation using their processing probes (service assurance and subscriber experience solutions). Some examples of the attained files results for NGAP (between RAN and Core) and HTTP2 (between 5GC virtual network functions) are depicted in Figure 29 a) and b).

C. CASE 3 – eBPF FOR CLOUD-NATIVE PLATFORM RESILIENCE

In this section, we present the results attained with Sauron for monitoring the integrity of files, detecting and preventing unwanted traffic between two applications, and detecting and preventing unwanted access to filesystems utilizing eBPF.

The reference architecture for the case studies discussed in the following was described in Section III-A and shown in Figure 18. Tests were run using a few selected files and applications deployed on a node in a Kubernetes cluster running on GCP.

1) FILE INTEGRITY MONITORING UTILIZING eBPF

File integrity monitoring (FIM) is a security technique used to detect changes to critical files, folders, or system settings on a computer or network. eBPF is not typically used as the primary tool for file integrity monitoring as it is designed to intercept and filter system calls, network packets and events on the kernel level rather than monitoring files and directories on the file system level. However, it is possible to use eBPF to monitor system calls and events that can indicate unauthorized changes to the file system. For example, by attaching eBPF programs to relevant system calls, you can monitor file access and detect changes to the files. Additionally, eBPF can monitor network activity and detect any abnormal network traffic that could indicate an attack, such as the exfiltration of files or a malicious process communicating to a command-and-control server.

An effective architecture for implementing FIM using eBPF can be designed as follows:

1) eBPF Probes: Attach eBPF programs to system calls relevant to file access, such as *open*, *close*, *read*, *write*, and *unlink*. These eBPF programs can monitor file access and detect any unauthorized changes to the files.
2) Collection and Parsing of Data: The eBPF programs can collect information about file access, such as *file paths*, *process IDs*, and container-related information such as *cgroup IDs* and *timestamps*. This data can be parsed and stored in a data store, such as a database, for further analysis.
3) Comparison: Compare the current state of the files with a previously recorded "baseline" state, which is the state of the files when they were last known to be in a safe state. Any changes that occur outside of the baseline state can be flagged as suspicious.
4) Correlation and Analysis: Correlate the data collected by the eBPF programs with other security data such as *logs*, *network traffic*, or *system events* to identify any suspicious behavior or anomalies. This step can use machine learning techniques, or any other type of analysis, to detect patterns of attack.

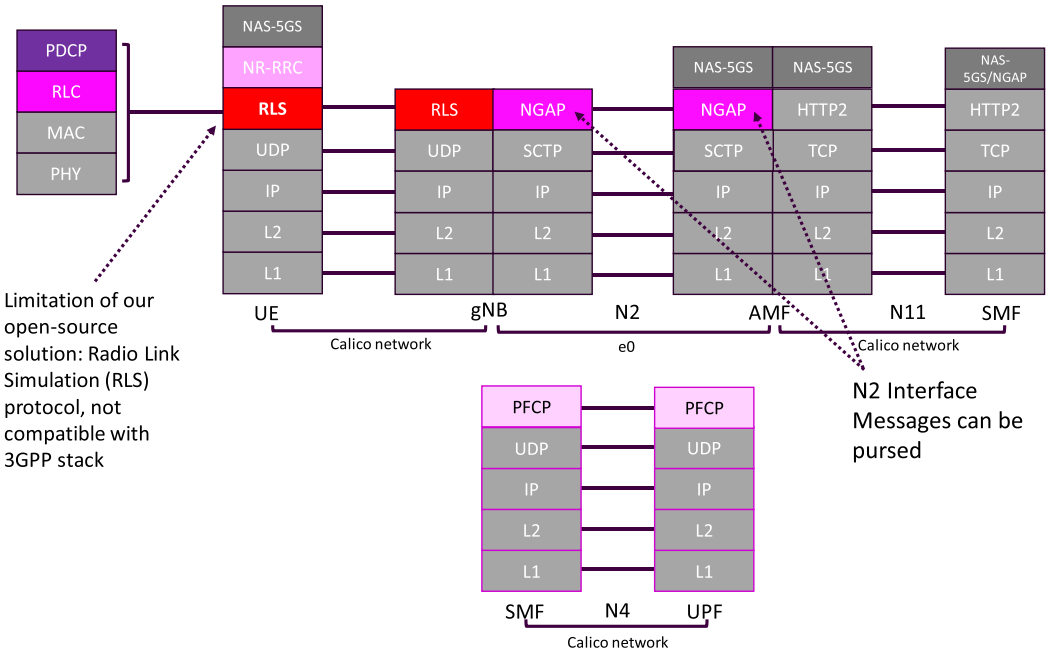


FIGURE 27. Control plane protocol stack supported by the UERANSIM-Free5GC testbed.

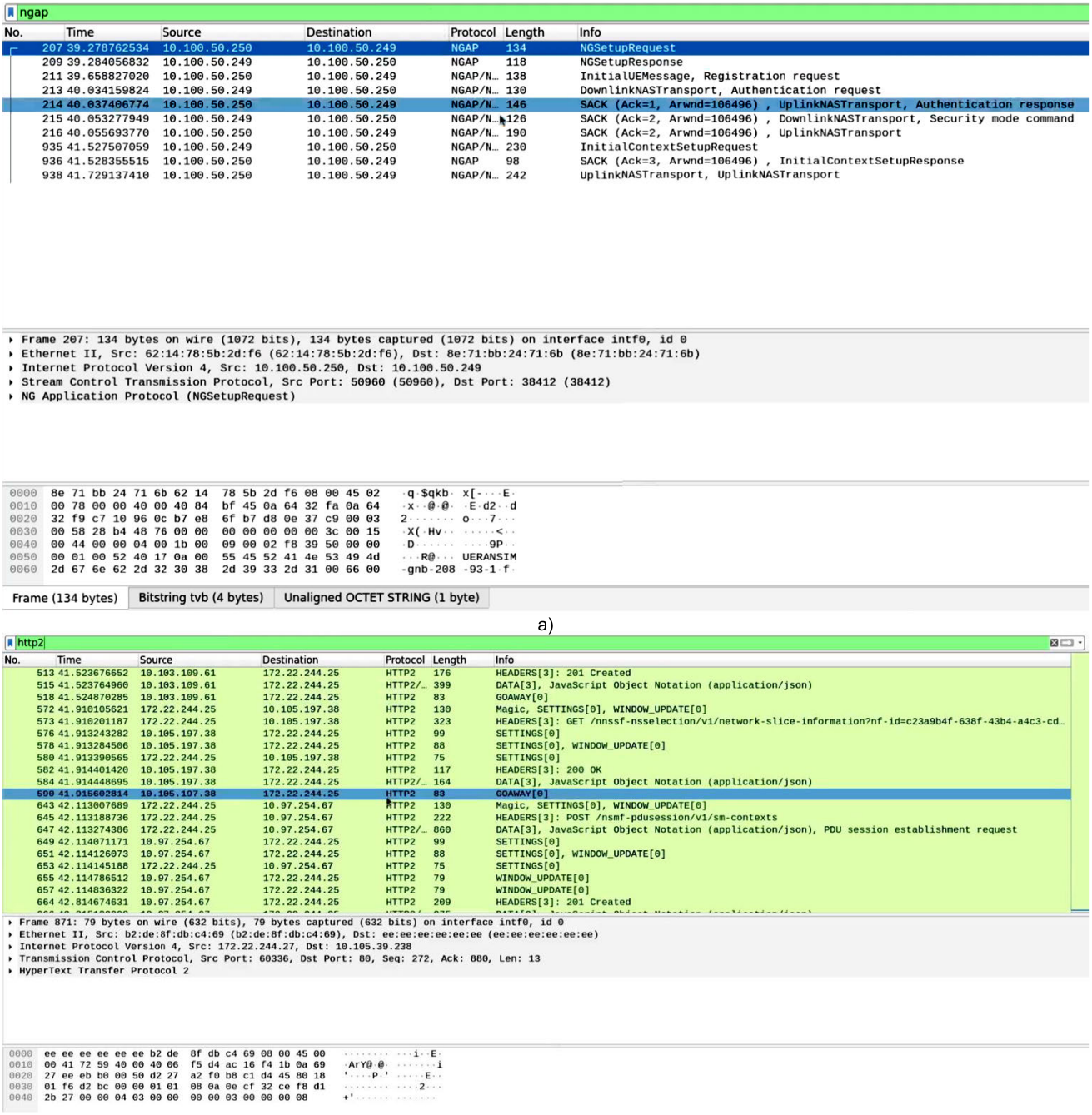
FIGURE 28. Examples of 5G gauges and counters derived by the Sauron Agent (daemonset) in real time.

5) Alerting: Send alerts to the incident response team when suspicious or malicious activity is detected. This can include information such as the file that was accessed, the process that performed the access, and the time of the access.
6) Response: Depending on the level of the detected attack, the incident response team can take actions such as blocking the malicious process, quarantining the affected files, or shutting down the affected systems.

While eBPF can monitor file access and detect unauthorized changes, this method can only detect changes that happen after the eBPF probe is attached. Therefore, it is important to use eBPF in conjunction with other file integrity monitoring solutions to ensure that all changes to the file system are detected. Also, it is essential to note that FIM is not a single point of security; it should be integrated with other security tools, such as antivirus, firewall and intrusion detection systems, to provide a comprehensive security solution that could detect and respond to potential threats.

Figure 30 and Figure 31present the logs of our FIM proof of concept (PoC) using eBPF.

Figure 30 illustrates how an attacker was detected after writing "Hi!" into the monitored file /etc/file1 using the shell standard output redirection. The FD_INSTALL event is logged when a process allocates a new file descriptor for a specific file. In this example, the bash process, whose PID was detected as 6436, allocated the file descriptors 3 and 1 to the /etc/file1 file. The next CLOSE event indicates that file descriptor 3 was closed. Lastly, the WRITE event allowed us to detect that the process wrote 4 characters into the file



a)

b)

FIGURE 29. Example of packet captured at 5GC pod interface and filtered based on protocol type, e.g., a) NGAP between RAN and Core; and b) HTTP2 between 5GC virtual network functions, for validation purposes.

/etc/file1 through the file descriptor 1. (“Hi!” is composed of 3 characters plus the invisible ‘\0’ termination character.) It is worth noticing that file descriptor 1 is not closed by the bash process since, by convention, it represents the process’s standard output.

Figure 31 shows a similar attack pattern. Here the attacker was detected while modifying the content of the monitored file through the *nano utility*. The nano utility opens the file and reads its content to show it on the screen. In this experiment, those two events were detected and communicated to the user through the first three log lines (FD_INSTALL, READ(4) and READ(0) events). The nano utility saves the modification to the file in a second temporary file during the editing phase. When the attacker saved the changes, the original file was reopened, and the file was modified and then closed; these events were reported in the

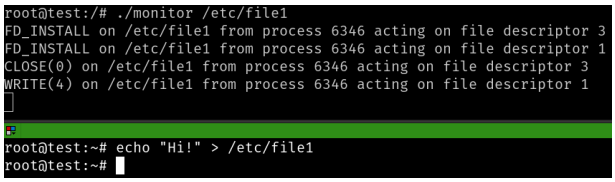


FIGURE 30. Sauron daemonset agent detecting the modification in monitored file /etc/file1 utilising the shell standard output redirection.

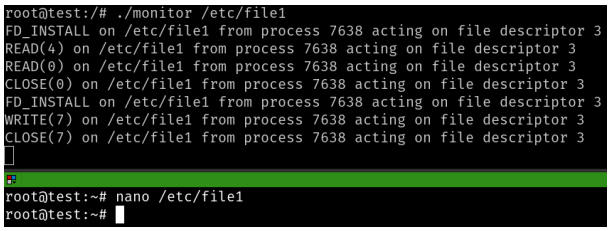


FIGURE 31. Sauron daemonset agent detecting the modification in monitored file /etc/file1 employing nano utility.

last 3 log lines (FD_INSTALL, WRITE(7) and CLOSE(7) events).

In both examples, only events related to files whose path starts with */etc/file1* are monitored. However, in our implementation, a different folder path, as */etc*, etc., can also be specified.

2) DETECTING AND PREVENTING UNWANTED TRAFFIC EMPLOYING eBPF

eBPF can also detect and prevent unwanted traffic by attaching eBPF programs to various hook points in the Linux kernel. Here is an example of how this can be done:

1) First, eBPF programs are written to intercept and filter network traffic based on *IP addresses*, *ports*, or *other header fields* or information related to generating processes. This can be done by attaching eBPF programs to the appropriate hooks in the Linux kernel, such as *XDP*, *TC*, *sk_skb* and *cgroup_skb*.
2) The eBPF programs are configured with rules to match the unwanted traffic, such as *IP addresses*, *ports* or *other parameters* that identify the flows to be blocked. This can be done by using eBPF maps, which are data structures in the kernel that can be dynamically configured from the user space with the desired rules and accessed from eBPF programs to enforce them.
3) Once the eBPF programs are in place, they begin intercepting and filtering network traffic in real time, based on the rules configured.
4) If needed, eBPF can also be used to log the events of unwanted traffic, which can be useful for forensic analysis or troubleshooting.

Figure 32 a) and b) give an example of a declarative way to express network policies (the style follows the Kubernetes NetworkPolicy API).

In Figure 32 a), the first SecurityPolicy is applied to all pods having the label "run" equal to "curl". In the test environment, only one *curl* pod, having the address 172.22.244.35, was deployed with this label. The policy only allowed: 1) Incoming traffic from 172.22.80.22 to port 7000, on which the *curl* pod was listening, and 2) Outgoing traffic to target port 80 with IP address 172.22.80.22, using TCP. The address 172.22.80.22 is the IPv4 address of a *nginx* pod deployed in the same cluster.

In Figure 32 b), the *nginx* pod is labelled with "app" equal to "nginx" and is targeted by the second SecurityPolicy. This policy only allows: 1) incoming traffic to port 80 from 172.22.244.35; and 2) outgoing traffic to target port 7000 with IP address 172.22.244.35, using TCP. The address 172.22.244.35 is the Ipv4 address of the *curl*pod deployed on the same test environment.

Figure 32 c) shows how Sauron was able to detect the traffic and enforce the policies. Using curl 172.22.80.22:39395 from the *curl* pod, where the destination port 39395 was selected randomly, the outgoing *http* request was blocked at the *curl* egress interface, as the random target port 39395 (*nginx* port) was not equal to the target port (80), specified in the egress policy applied to the "curl"d.

3) DETECTING AND PREVENTING UNAUTHORIZED ACCESS TO FILESYSTEMS UTILIZING eBPF

eBPF can be used to detect and prevent unwanted access to filesystems by attaching eBPF programs to the appropriate hooks in the Linux kernel, such as the Virtual File System (VFS) hooks. Here is an illustration of how this can be achieved:

1) The eBPF program is written to intercept file system events, such as *file creation*, *modification*, and *deletion*. The eBPF program is then attached to the Linux kernel VFS hooks to capture file system events in real time.
2) The eBPF program uses eBPF maps to store a list of files and directories that should be protected from unwanted access, along with their permissions and access control rules.
3) When a file system event occurs, the eBPF program compares the file or directory being accessed with the list of protected files and directories stored in the eBPF map.
4) If the file or directory being accessed is on the list of protected files and directories, the eBPF program compares the requested access permissions with the access control rules stored in the eBPF map.
5) If the requested access does not match the access control rules, the eBPF program can trigger an alert and deny access to the file or directory.
6) Logging the events can also be done by the eBPF program. This can be useful for forensic analysis or troubleshooting.

Figure 33 shows the log of a PoC implementation of a system able to detect unwanted access to the filesys-

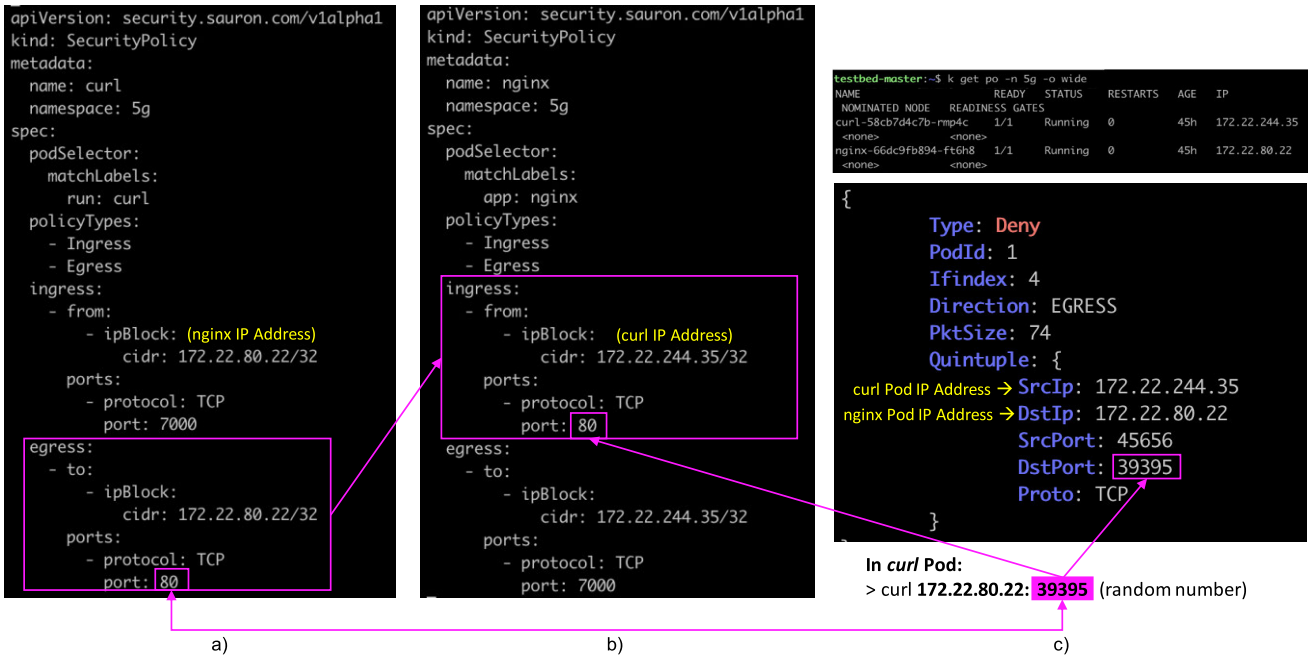


FIGURE 32. a) SecurityPolicy applied on curl (client) pod; b) SecurityPolicy applied on nginx (server) pod; c) Example of detecting the unwanted traffic and enforcing policies between nginx and curl pods.

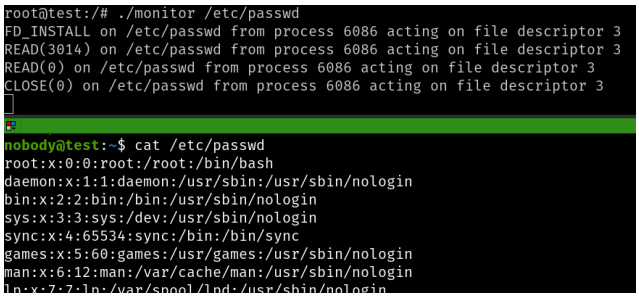


FIGURE 33. Sauron daemonset agent detecting the unwanted access to filesystems.

tem. Here, an attacker was trying to read the content of the monitored */etc/passwd* file through the *cat* utility. The FD_INSTALL event indicated that a process (the *cat* process), whose PID was detected to be 6086, had allocated the file descriptor 3 to the */etc/passwd* file. The following three events (READ (3014), READ (0) and CLOSE(0)) indicated that the process read the entire file content and then closed it.

D. CASE 4 – eBPF FOR ESTIMATING ENERGY CONSUMPTION

As mentioned in Section III-D, KEPLER utilizes eBPF probes. To put KEPLER to the test, we have created a testbed in a K8s cluster consisting of four worker nodes and one master. The cluster was created in a virtualized environment to understand how reliable KEPLER is in such an environment and whether we can add other usable data sources that would work in a similar context. Obviously, in a virtualized environment, KEPLER will not be able to collect any data from sources such as *PERF hardware* counters or *RAPL* counters.

In our tests, we focused on 5G virtual functions, using the open-source software presented in Section III-B, Figure 26, and a sample microservices application, i.e., Google Online Boutique [42].

Regarding the 5G network, we aimed to investigate KEPLER's ability to track the real-time energy consumption of CNFs in a K8s cluster that runs both the Core and RAN part of the 5G network, as in reality, the energy consumption of individual components varies in time.

Initially, we analyzed the case where only the CNFs of the 5G-Core, along with two gNBs (of the RAN) were running with no UEs. Then, the UEs were gradually added and removed from one gNB to the other, using suitable scripts to simulate the case where a certain number of UEs connect and disconnect from a serving cell, e.g., in the case of handovers. That allowed us to observe the resources that were required to perform some procedures involving, for example, the access and mobility function (AMF) in the core network or protocols in the gNBs.

The data that KEPLER managed to obtain – using a *tracepoint* type eBPF program and, as mentioned in Section III-D, the ML model to estimate the energy consumption per container or pod – were then displayed using a Grafana dashboard, as shown in Figure 34. (The actual measurement unit was joules, and the result was an average value of a set of measurements made over an interval of 10 minutes. Then these measurements were scaled to the unit of kWh per day.

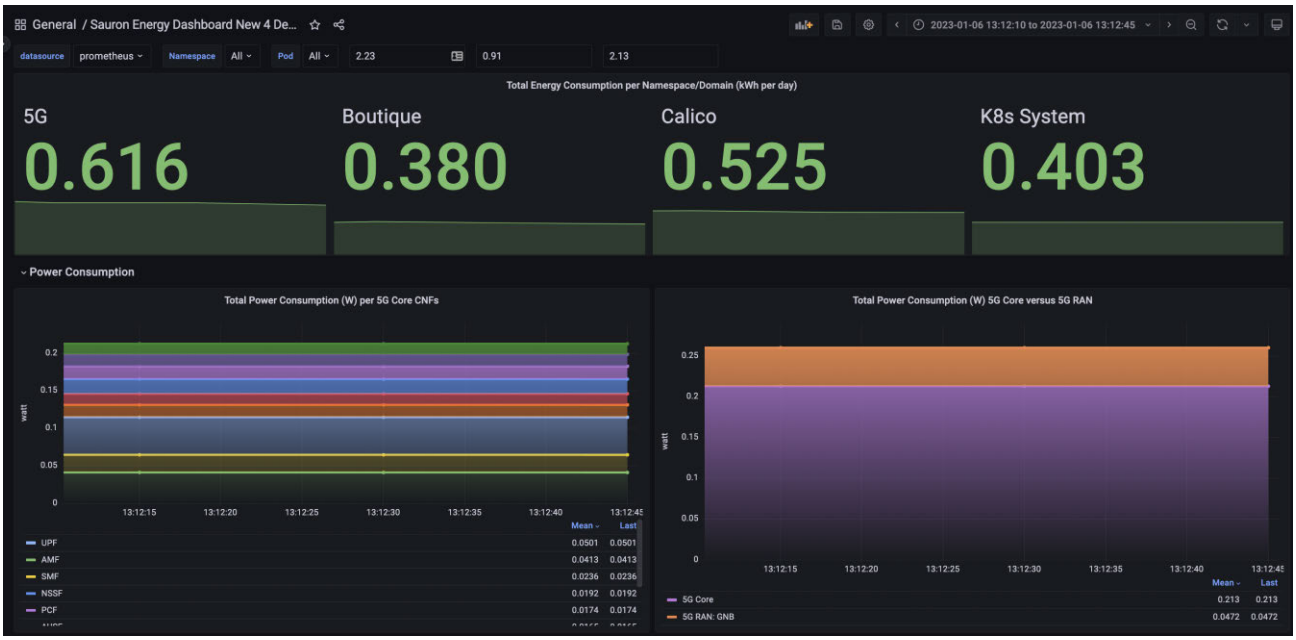


FIGURE 34. Example of eBPF energy dashboard - example of demo results.

That is the reason why the numbers on the dashboard are so small.)

In addition, based on those data, an operator could identify which nodes of the cluster needed resources and derive how much the various 5G-Core CNFs consumed with a granularity of their choice, e.g., per container, pod, node or per CNF as illustrated in Figure 34.

V. CONCLUSION

eBPF is to the Linux and Microsoft Windows kernel what JavaScript is to the web browser. Although some challenges do exist – for example, eBPF development is not easy, eBPF is fast-paced and hard to keep up with, implementation details may vary by kernel version, and there is no easy packaging/deployment solution – this fundamental enabling technology is leading to a major wave of innovation in the kernel space, bringing immediate benefits in a cloud-native environment, especially for Telco networks.

In production, many of the Linux kernel building blocks are decades old, and eBPF is creating the necessary cloud-native abstractions and new building blocks required for dynamically programming the kernel in a safe, performant, and scalable way. This is primarily due to its dynamic programmability, reliability, and ability to achieve great kernel instrumentation and workload visibility with minimal disruption.

The fact that we can inspect packets gives us extremely performant *observability* tools that can be mapped to other features, such as Telco and Kubernetes metadata, and provide in-depth *security* forensics from the extracted information.

We can utilize the ability of eBPF to drop or modify packets based on *network policies* applied to various hooks in the kernel or apply *security policies* based on application-level verbs or specific paths and perform encryption with eBPF.

Additionally, since we can send packets and change the destination for a packet, eBPF allows us to create powerful *network functionalities*, such as load balancing, routing, and service mesh, with minimal utilization of envoy proxies.

eBPF enables the *next generation of service mesh* because we do not necessarily need to instrument pods with sidecars and can improve Telco cloud performance without any app, network micro-service, or configuration changes.

We will see more and more observability, security, and networking solutions based on eBPF, and both legacy vendors and emerging eBPF-native players will heavily rely on eBPF as an integral part of their infrastructure stack.

As a result, eBPF is expected to be one of the largest and fastest-growing markets in infrastructure software, with organizations willing to allocate a two-digit figure of their infrastructure spend to it, proportionally.

Rakuten Mobile has built a fully virtualized, end-to-end, cloud-native mobile network, and Open RAN coverage has been deployed with 300,000 cells in Japan. We operate the world's first fully cloud-native mobile network, and our revolutionary mobile network deployment provides customers with a high-quality mobile service, powered by eBPF.

Among other benefits, eBPF efficiently extends the cloud-native capabilities of our network. This more modular eBPF architecture enables developers to safely and reliably deploy software frequently, and platform engineering teams can provision, observe, and secure scalable, dynamic, available, and high-performance environments, allowing

developers to focus on coding high-quality applications and microservices.

Our platform, named *Sauron* in this work, is an example of what we have achieved with eBPF for Telco systems.

ACKNOWLEDGMENT

The authors would like to acknowledge Tareq Amin and Sharad Sriwastawa of Rakuten Mobile and Rakuten Symphony for providing them with the opportunity to work on this important subject and for their invaluable contributions.

REFERENCES

[1] *Cloud Native Computing Foundation*. Accessed: Feb. 2, 2023. [Online]. Available: https://www.cncf.io/
[2] *Kubernetes*. Accessed: Feb. 2, 2023. [Online]. Available: https://kubernetes.io/
[3] *Open RAN Alliance*. Accessed: Feb. 2, 2023. [Online]. Available: https://www.o-ran.org/
[4] D. Soldani, "6G fundamentals: Vision and enabling technologies: From 5G to 6G trustworthy and resilient systems," *J. Telecommun. Digit. Economy*, vol. 9, no. 3, pp. 58–86, Aug. 2021, doi: 10.18080/jtde.v9n3.418.
[5] (2023). *ITU-R Working Party 5D (WP 5D), IMT Towards 2030 and Beyond (IMT-2030), Vision Paper*. Accessed: Feb. 2, 2023. [Online]. Available: https://www.itu.int/en/ITU-R/study-groups/rsg5/rwp5d/Pages/default.aspx
[6] L. Rise, *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security*. Sebastopol, CA, USA: O'Reilly Media, 2023, p. 218. [Online]. Available: https://isovalent.com/learning-ebpf/
[7] *Libbpf Library*. Accessed: Feb. 2, 2023. [Online]. Available: https://github.com/libbpf/libbpf
[8] *BCC Tutorial*. Accessed: Feb. 2, 2023. [Online]. Available: https://bcc/docs/tutorial_bcc_python_developer.md
[9] *eBPF-io. Documentation*. Accessed: Feb. 2, 2023. [Online]. Available: https://ebpf.io/
[10] B. Gregg, *BPF Performance Tools*. Reading, MA, USA: Addison-Wesley, 2020, p. 841.
[11] H. Sharaf, I. Ahmad, and T. Dimitriou, "Extended Berkeley packet filter: An application perspective," *IEEE Access*, vol. 10, pp. 126370–126393, 2022, doi: 10.1109/ACCESS.2022.3226269.
[12] *Gartner Says Cloud Will be the Centerpiece of New Digital Economy*, Gartner, Stamford, CT, USA, Nov. 2021.
[13] D. Soldani, H. Bour, and S. Jafarizadeh, "Security and observability for cloud-native platforms," Cybersecurity Magazine, Sep. 2022. [Online]. Available: https://cybersecurity-magazine.com/security-and-observability-for-cloud-native-platforms-part-1/
[14] *KEPLER*. Accessed: Feb. 1, 2023. [Online]. Available: https://sustainable-computing.io/
[15] Isovalent. *Documentation and Blogs*. Accessed: Feb. 2, 2023. [Online]. Available: https://isovalent.com/
[16] *The Learning Channel*. Accessed: Feb. 2, 2023. [Online]. Available: https://www.youtube.com/@TheLearningChannel-Tech/featured
[17] J.-B. Lee, T.-H. Yoo, E.-H. Lee, B.-H. Hwang, S.-W. Ahn, and C.-H. Cho, "High-performance software load balancer for cloud-native architecture," *IEEE Access*, vol. 9, pp. 123704–123716, 2021, doi: 10.1109/ACCESS.2021.3108801.
[18] Istio. *The Istio Service Mesh*. Accessed: Feb. 2, 2023. [Online]. Available: https://istio.io/latest/about/service-mesh/
[19] Calico. *Documentation and Blogs*. Accessed: Feb. 2, 2023. [Online]. Available: https://www.tigera.io/project-calico/
[20] New Relic. *Simple, Transparent Pricing. Only Pay for What You Use*. Accessed: Feb. 2, 2023. [Online]. Available: https://newrelic.com/pricing
[21] Cilium. *eBPF-based networking, security, and observability. Documentation and Blogs*. Accessed: Feb. 2, 2023. [Online]. Available: https://cilium.io/
[22] ThousandEyes. (2023). *The Evolution of Network Monitoring From Data Center to Cloud*. [Online]. Available: https://www.thousandeyes.com/resources/evolution-network-monitoring
[23] S. Sundberg, A. Brunstrom, S. Ferlin-Reiter, T. Høiland-Jørgensen, and J. D. Brouer, "Efficient continuous latency monitoring with eBPF," in *Passive and Active Measurement* (Lecture Notes in Computer Science), vol. 13882, A. Brunstrom, M. Flores, and M. Fiore, Eds. Cham, Switzerland: Springer, 2023, pp. 191–208, doi: 10.1007/978-3-031-28486-1_9.
[24] *RFC Series*. Accessed: Feb. 3, 2023. [Online]. Available: https://www.rfc-editor.org/
[25] *5G System, Technical Realization of Service Based Architecture; Stage 3*, document 3GPP TS 29.500, Version 18.0.0, 2023. [Online]. Available: https://www.3gpp.org/specifications-technologies/specifications-by-series
[26] OpenTelemetry. *A Collection of Tools, APIs, and SDKs*. Accessed: Feb. 2, 2023. [Online]. Available: https://opentelemetry.io/
[27] C. Liu, Z. Cai, B. Wang, Z. Tang, and J. Liu, "A protocol-independent container network observability analysis system based on eBPF," in *Proc. IEEE 26th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Hong Kong, Dec. 2020, pp. 697–702. [Online]. Available: https://ieeexplore.ieee.org/document/9359159
[28] S. Magnani, F. Risso, and D. Siracusa, "A control plane enabling automated and fully adaptive network traffic monitoring with eBPF," *IEEE Access*, vol. 10, pp. 90778–90791, 2022, doi: 10.1109/ACCESS.2022.3202644.
[29] *Polycube's Documentation*. Accessed: Feb. 2, 2023. [Online]. Available: https://polycube-network.readthedocs.io
[30] Sysdig. *Sysdig and Falco now Powered by eBPF*. Accessed: Feb. 2, 2023. [Online]. Available: https://sysdig.com/blog/sysdig-and-falco-now-powered-by-ebpf/
[31] Tetragon. *eBPF-based Security Observability & Runtime Enforcement*. Accessed: Feb. 2, 2023. [Online]. Available: https://isovalent.com/blog/post/2022-05-16-tetragon/
[32] *Sustainable-Computing-io. Peaks*. Accessed: Feb. 1, 2023. [Online]. Available: https://github.com/sustainable-computing-io/peaks
[33] *Sustainable-Computing-io. Clever*. Accessed: Feb. 1, 2023. [Online]. Available: https://github.com/sustainable-computing-io/clever
[34] *Cloud Carbon Footprint*. Accessed: Feb. 1, 2023. [Online]. Available: https://github.com/cloud-carbon-footprint/cloud-carbon-footprint
[35] *Kube Green*. Accessed: Feb. 1, 2023. [Online]. Available: https://github.com/kube-green/kube-green
[36] *Kube Downscaler*. Accessed: Feb. 1, 2023. [Online]. Available: https://codeberg.org/hjacobs/kube-downscaler
[37] CNCF. *Tag ENV Sustainability*. Accessed: Feb. 1, 2023. [Online]. Available: https://github.com/cncf/tag-env-sustainability
[38] J. Sheth, V. Ramanna, and B. Dezfouli, "FLIP: A framework for leveraging eBPF to augment WiFi access points and investigate network performance," in *Proc. 19th ACM Int. Symp. Mobility Manage. Wireless Access*, Nov. 2021, pp. 117–125.
[39] J. Sheth, "Enhancing the quality of service and energy efficiency of WiFi-based IoT networks," Ph.D. thesis, Dept. Eng., Santa Clara Univ., Santa Clara, CA, USA, 2022. [Online]. Available: https://scholarcommons.scu.edu/eng_phd_theses/40
[40] *Free5GC*. Accessed: Feb. 2, 2023. [Online]. Available: https://github.com/free5gc/free5gc
[41] *UERANSIM*. Accessed: Feb. 2, 2023. [Online]. Available: https://github.com/aligungr/UERANSIM
[42] *GCP Microservices*. Accessed: Feb. 1, 2023. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

DAVID SOLDANI (Senior Member, IEEE) received the M.Sc. degree (magna cum laude) in engineering from the University of Florence, Italy, in 1994, and the D.Sc. degree in technology (Hons.) from the Helsinki University of Technology, Finland, in 2006. Throughout his career, he has held prestigious academic positions, including a Visiting Professor with the University of Surrey, U.K., in 2014, an Industry Professor with The University of Technology Sydney (UTS), Australia, in 2016, and an Adjunct Professor with the University of New South Wales (UNSW), in 2018. In his professional roles, he has served in significant leadership positions. He has held the positions of the Chief Information and Security Officer (CISO), e2e, Global, as well as SVP innovation and advanced research with Rakuten. Previously, he held the position of Chief Technology and Cyber Security Officer (CTSO) with the ASIA Pacific Region, Huawei. He was the Head of 5G Technology, e2e, Global, with Nokia, the Head of the Central Research Institute (CRI), and the VP of strategic research and innovation in Europe with Huawei European Research Center. He is currently with Rakuten Mobile Inc. He can be reached at: https://www.linkedin.com/in/dr-david-soldani/

PETRIT NAHI received the Ph.D. degree in electronic engineering from the Queen Mary University of London. His research explored the use of artificial intelligence to implement agent-based control and optimization of network coverage in combination with the use of novel advanced antenna techniques. He has over 20 years of experience in the mobile communications industry and has held various product and technology leadership roles. He joined Rakuten Mobile Inc., in 2019, as the Head of Data Science and the Chief Data Officer. He was a Chief Scientist of RAN with NetScout Systems Inc. He has worked on the development of various network planning and optimization products, network performance analytics, service assurance, and other applications throughout his career. He is currently a Senior Member of Technical Staff with the CTO Office, Rakuten Mobile Inc., responsible for driving strategic partnerships, innovation, and the adoption and use of artificial intelligence to derive insights on network performance throughout its life cycle and enable automation of engineering and operation to achieve network autonomy.

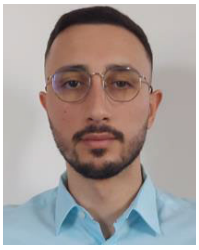
HAMI BOUR (Member, IEEE) has been a Research Scientist with Rakuten Mobile Inc., since January 2021. She has spent the last 20 years as a Senior Network Security Expert, a Cyber Security Professional, an Ethical Hacker, and an Information Security Analyst Team Lead in several industries, including telecommunications corporations and banking systems, working with senior executives and CEOs worldwide. Her extensive knowledge of multi-layered security architecture, incident response, compliance management, cloud-native security, as well as software-defined networking security, which was the focus of her Ph.D. research, has made her the go-to expert in the field.

SABER JAFARIZADEH (Member, IEEE) received the Ph.D. degree from the Department of Electrical and Information Engineering, University of Sydney, in 2015. He is currently a Senior Research Scientist with the Innovation and Advanced Research Department (the Research and Development Organization, Rakuten Mobile Inc.). His research interests include wireless ad-hoc networks and distributed control, with an emphasis on distributed optimization and epidemic processes.

MOHAMMED F. SOLIMAN received the B.Sc. and M.Sc. degrees in computer science from Suez Canal University, Egypt, in 2000 and 2006, respectively, and the Ph.D. degree in computer engineering from Yokohama National University, Japan, in 2013. He was a Postdoctoral Researcher with Yokohama National University and The University of Electro-Communications and a Visiting Professor with the Tokyo Institute of Technology. He has been a Research Scientist with Rakuten Mobile Inc., Japan, since February 2019. He has proven leadership skills working with multiple teams across the globe in various domains, including industry (since 2019), entrepreneurship (2000–2010), academia (2000–2018), and volunteering environment (2014–2021). He is a peer reviewer of multiple journals. His research interests include cloud-native networks performance optimization, and cross-layer optimization for Telco, BAN, and native networks.

LEONARDO DI GIOVANNA received the M.Sc. degree in computer engineering from Politecnico di Torino, Italy, in 2022. He is currently collaborating as a Research Fellow with Politecnico di Torino. He is an Italian Computer Engineer. His research interests include cloud-native technologies, networking, and kernel design and implementation aspects.

FRANCESCO MONACO received the M.Sc. degree in computer engineering from Politecnico di Torino, Italy, in 2022. He is currently collaborating as a Research Assistant with Politecnico di Torino. His research interests include high-performance networking, programmable data planes, and cloud-native technologies.



GIUSEPPE OGNIBENE received the M.Sc. degree in computer networks engineering from Politecnico di Torino, in 2021. He is currently a Research Fellow with the DAUIN Department, Politecnico di Torino. His main research interests include high-performance networking, cloud-native computing, and everything related to the open-source world.

FULVIO RISSO (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer engineering from Politecnico di Torino, Italy, in 2000. He is currently an Associate Professor with Politecnico di Torino, and responsible for the Network and Multimedia Laboratory, Department of Control and Computer Engineering. He started many open-source projects, including WinPcap (the de-facto standard library for network analysis tools under the Windows platform), NetBee, Liqo, CrownLabs, and Polycube, which represent a breakthrough in their respective fields. He has coauthored more than 100 scientific publications. His research interests include high-speed and flexible network processing, edge/fog computing, software-defined networks, and network function virtualization.