A methodology for using eBPF to efficiently monitor network behavior in Linux Kubernetes clusters

by

Timothy D. Zavarella

B.S. Electrical Engineering and Computer Science
Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author .....
Department of Electrical Engineering and Computer Science
May 6, 2022

Certified by .....
Tomas Palacios
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Aleks Ryabin
Senior Manager – Engineering, NetApp, Inc.
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

A methodology for using eBPF to efficiently monitor network behavior in Linux Kubernetes clusters

by

Timothy D. Zavarella

Submitted to the Department of Electrical Engineering and Computer Science
on May 6, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

With the rise of container orchestration systems, such as Kubernetes and microservice based application architectures there has been a corresponding growth in tools aimed at monitoring these systems. As monitoring approaches have evolved the implementation of instrumentation has shifted from the application level to the platform level. The extended Berkeley Packet Filter (eBPF) can enable high performance and low overhead collection for platform level monitoring. Existing commercial eBPF monitoring systems are often tightly integrated systems with large dependencies and little flexibility in integration into alternative monitoring systems. This thesis presents a methodology for developing modular self-contained eBPF monitoring systems which are portable across various kernel versions, Container Network Interface (CNI) plugins, and cluster configurations. The choice of stable hook points and the BPF CO-RE approach to development using the libbpf or Cilium/ebpf loaders is recommended in this methodology. A proof of concept monitor was developed which captures network traffic on a cluster using the stable Traffic Control direct-action hook point. Packet capture at pod virtual ethernet network interfaces was selected to allow for CNI independent correlation of packets to cluster workloads. The prototype developed provides a suitable platform for implementing additional monitoring functionality on top of and was integrated with an existing NetApp cloud monitoring system.

Thesis Supervisor: Tomas Palacios
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Aleks Ryabin
Title: Senior Manager – Engineering, NetApp, Inc.

Acknowledgments

I would like to thank everyone at NetApp for being incredible resources and extremely supportive throughout my time there. This research would not have been possible without them. Aleks, Francisco and Jimmy thank you for your advice and feedback throughout the research process. You provided indispensable guidance as I immersed myself in a field I had very little experience in, but which proved to be incredibly rewarding. Thank you to Anton, Pranav and everyone else at NetApp who also helped me along the way. Thank you, Professor Palacios, for advising this thesis.

Finally, I want to thank my partner for being a source of constant support and motivation.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

Chapter 1

Introduction

In recent years, the use of application containerization and container-orchestration systems has grown dramatically. Containerization provides a consistent deployment environment, security and isolation guarantees between containers and the host system, and allows for systems to rapidly scale up and down to meet current demand. To take the fullest advantages of these features, microservice based architectures are utilized instead of monolithic approaches. Microservice architectures organize applications as many independent, loosely coupled components. This enables services to be updated and scaled independently, unlike monolithic approaches where the entire application must be updated or scaled.

As services can interact in complex and unexpected ways there are a wide variety of tools aimed at monitoring these systems. Actions such as understanding interactions between services, the overall topology of services for applications running on the cluster, and identifying and resolving performance issues can all be difficult. Surveys and interviews with developers of microservice based applications consistently identify the complexity of monitoring as a particular pain point [40]. As microservice architectures spread application logic across loosely coupled components all potentially developed by different teams or third-party organizations, systematic monitoring solutions which do not require application support are attractive.

This research was performed at NetApp with the goal of developing a Kubernetes [25] network monitor capable of efficiently collecting network traffic with per-pod

granularity, and integrating with an existing Kubernetes monitoring product. An investigation of existing Kubernetes monitoring approaches and commercially available observability systems revealed several deficiencies. Distributed tracing approaches required manual instrumentation in the targeted application making deployment a development intensive process. Service meshes can provide observability to the TCP and application layer however they typically do not allow observability of lower network layers or protocols other than TCP. Service meshes also have overhead and latency impacts from their sidecar-based approach of capturing pod network traffic. Networking implementations such as Cilium [3] make use of the extended Berkeley Packet Filter (eBPF) [21] to implement components of the Kubernetes networking model in the kernel for improved efficiency. With access to various components of the networking stack, monitoring metrics can also be derived from this approach. These systems were not suitable for our use case as they are large and tightly integrated systems. Deploying these entire systems for capturing a small subset of network metrics would be cumbersome and unnecessarily resource intensive. Despite this, Cilium and other eBPF based monitoring solutions demonstrate the usefulness of eBPF for observing network behavior in Kubernetes clusters.

To address this need, a methodology for developing self-contained modular eBPF monitors for Kubernetes clusters was developed. eBPF toolkits and loaders are commonly used to develop eBPF systems more effectively and an evaluation of several common toolkits was performed. Toolkits and loaders were evaluated for their ability to meet the goals of minimizing dependencies and resource usage, and their portability support. The toolkit Cilium/ebpf was selected to implement a proof of concept monitor, for its high performance approach to supporting portability and ease of use.

This work is primarily focused on the Kubernetes container-orchestration system. Other popular orchestration systems such as Docker Swarm have structural similarities and utilize underlying containerization technologies in comparable manners [32]. As such the analysis of eBPF toolkits and loaders, and the methodology developed should be applicable for developing monitoring solutions for other orchestration systems.

1.1 Background

1.1.1 Kubernetes

Kubernetes is a container-orchestration system which aims to deliver reliable and consistent automation for tasks such as deploying, scaling, and updating containerized applications. Kubernetes systems, referred to as clusters, typically run across several potentially heterogeneous host machines called nodes. Pods are the atomic unit that Kubernetes manages, and a cluster contains multiple pods running on host nodes. Each pod contains one or more containers and is isolated from the host machine using namespaces and cgroups. Containers within the same pod are not isolated from each other and share namespaces. Containers performing ancillary functions to the main container in a pod are typically referred to as sidecars. This makes pods analogous to virtual machines and containers analogous to individual applications in traditional server architectures.

Kubernetes does not have a single networking system and instead allows for network plugins to implement their own custom networking. There are two main categories of network plugins. The first is the Kubenet plugin which is distributed with Kubernetes and provides a minimal feature set for a functioning Kubernetes cluster. The other category is network plugins which implement the Container Network Interface (CNI) Specification [11], referred to as CNI plugins. Kubernetes is agnostic to the specific network plugin used and CNI plugins can implement the networking model through dramatically different approaches and provide a variety of additional features.

For Kubernetes to interchangeably support different CNI plugins the Container Network Interface Specification defines the API CNI plugins must implement and some of the required functionality or behavior [10]. For this work the following are the notable features.

1. Clusters are assigned their own subnet masks and assign virtual cluster IP addresses to nodes and pods within this range.

2. Cluster IP addresses are not required to be routable on the host network.

However, the subnet assigned should be distinct from other subnets in use on the host network. Subnet collisions would prevent Kubernetes workloads from communicating with devices on that subnet.

3. Kubernetes follows, by default, an IP-per-pod model with each pod in a cluster receiving its own IP address. Pods can expose any ports required separately from any port bindings on the host machine or any other pod. Pods can also run on the host network; in which case they are assigned the host IP address and can bind to unbound host ports.

4. All containers on a pod exist in the same network namespace and share an IP and MAC address. Services are an abstraction which provides a stable address for resources to access ephemeral pods. Pods running on the host network share the host network namespace.

5. Pods must be able to communicate with all other pods in the cluster including pods on other nodes without network address translation (NAT). Pods running on the host network also share this requirement.

6. Processes running directly on host nodes must be able to communicate with all pods running on that node. It isn't required that processes are able to communicate with pods running on other nodes although it is often supported.

There are many different CNI plugins available tailored for various use cases. Many CNI plugins also allow for significant user configuration, including in some the option for entirely different data planes. This makes non-generalized monitoring solutions infeasible. Several popular CNI plugins and an overview of some of their added features are provided to demonstrate the level of variety.

1. AWS VPC CNI [31]. This is a CNI developed for Amazon's AWS which is highly performant and integrated into AWS VPC (virtual private cloud). Integration with VPC allows for using existing tools for monitoring, applying routing policies and security groups. Additionally, the CNI allocates pods IP addresses from the

VPC, so that packets between pods can be directly routed on the host network as opposed to proxying.

2. Calico [35]. Calico supports multiple networking stacks including an eBPF based stack optimized for Kubernetes networking. Calico also provides security features such as network policies and support for Wireguard encryption between pods.

3. Flannel [4]. This is a mature CNI plugin targeting easy installation and configuration. Flannel creates a VXLAN overlay network to implement pod to pod communications between pods on different nodes.

4. Cilium [3]. An eBPF based CNI plugin implementing monitoring, load balancing, and security policies with significant performance benefits for some cases. This plugin and its monitoring approach is discussed in greater detail in section 2.3.

Despite their increased complexity, containerization and container-orchestration systems such as Kubernetes provide several important advantages which have driven their adoption. Containerization provides isolation between containers and the host machine, and from other containers running on the same system. For example, a root privilege escalation on a vulnerable containerized application does not necessarily compromise other containers or the host machine. Additionally, containerized applications contain all of their dependencies so host systems can be updated independently of the workloads running on them. This also allows for simplification of the build and deployment process as a container can be run across many different Linux distributions without the need for distribution specific packaging and versions as is necessary for native packaging formats such as deb and rpm.

Containerization also provides significant performance advantages over virtualization, such as significantly reduced startup times. This allows for quicker provisioning of new instances of an application to scale to meet demand or more rapidly deploy updates. Containers share the host kernel as shown in Figure 1-1 which significantly reduces memory overhead when compared to virtual machines and can provide near host performance.

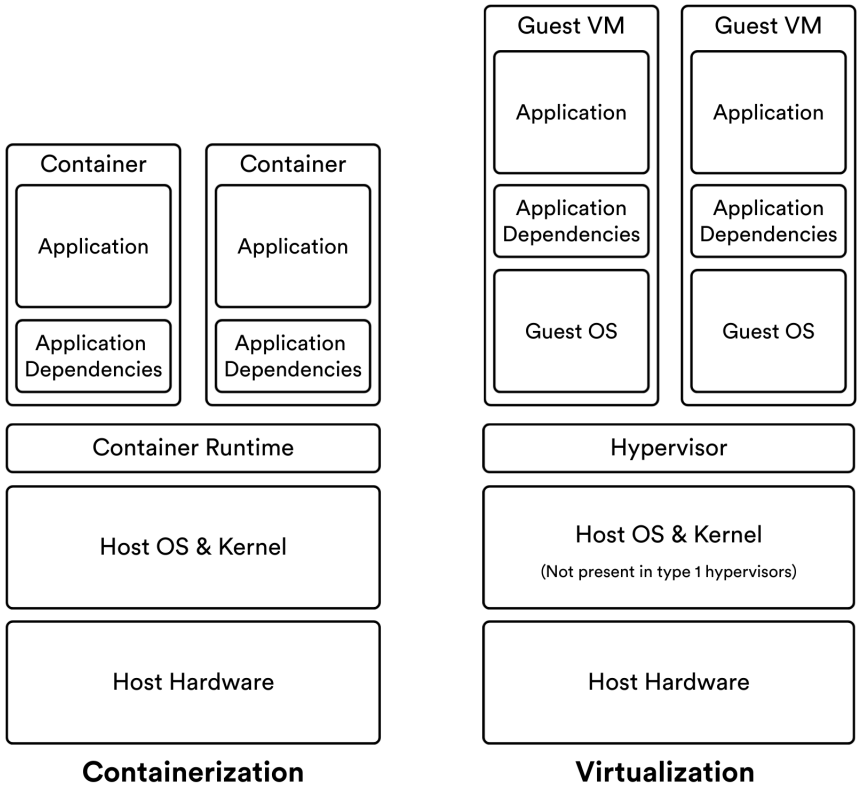

Figure 1-1: A comparison of applications running on containerized systems and virtualized systems

1.1.2 Overview of eBPF

The Berkeley Packet Filter (classic BPF or cBPF) is a Unix kernel technology that was introduced in 1992 that allowed for userspace programs to execute code in a limited in-kernel virtual machine at specific hook points in the network stack [37]. The primary use case was to enable configurable stateless packet filtering without the overhead associated with copying packets to userspace. Classic BPF was used prominently in tools such as tcpdump.

The development of the extended Berkeley Packet Filter (eBPF) [23] has unlocked many new use cases particularly for server and microservice environments. The addition of new hook points throughout the kernel and in userspace allows for the behavior of components throughout the system to be monitored or modified. Persistent

kernel data structures accessible in both userspace and eBPF programs called eBPF maps were introduced, enabling stateful eBPF programs. Finally, the in-kernel virtual machine was dramatically expanded by increasing the number of registers from two to ten, increasing register sizes to 64 bits, introducing a zero overhead calling convention, and increasing the maximum stack size among other improvements [22]. While not Turing complete (due to limits imposed by program verification which is discussed later) eBPF can be used for a wide variety of applications. Development of eBPF is still ongoing with additional hook points, eBPF map types and other features available for newer kernel versions.

The kernel provides an extensive set of kernel hook points or eBPF program types [24]. A subset of hooks which are often used in Kubernetes network monitoring is provided below.

1. eXpress Data Path (XDP): A hook point early in the networking stack where received packets have been minimally acted upon. Allows eBPF programs to capture and act upon packets before further processing by the network stack. An example use case is high performance filtering as packets can be dropped well before other approaches such as iptables.

2. Traffic control (TC) hooks: Allow attaching to the traffic control subsystem which is after the packet header has been parsed and a socket buffer structure has been created. Can capture both ingress and egress traffic unlike XDP. An example use case is packet capture for monitoring as it benefits from not re-implementing and duplicating work to process the raw packet.

3. Kernel Probes (kprobes): Allow for inserting breakpoints at instructions in the kernel. These can be used to monitor or modify kernel behavior at any point in the kernel.

4. Userspace Probes (uprobes): Similar to kprobes but which attach to userspace instructions. Useful for applications such as monitoring shared libraries.

An eBPF map, while somewhat misleadingly named, is a generic term for persistent

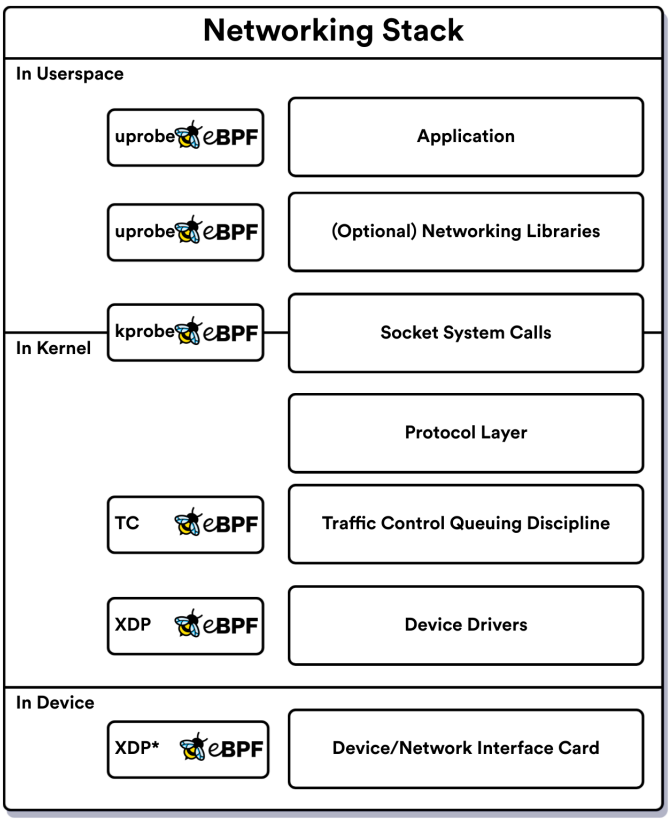

Figure 1-2: An overview of components in the Linux networking stack which are commonly modified or monitored with eBPF and hook points often used to attach to that subsystem. *XDP on device is not supported by all NICs.

kernel data structures which can be interacted with in eBPF and userspace programs through `bpf(BPF_MAP_*)` system calls. There are a number of different eBPF map types available such as hash (tables), arrays, queues, stacks, ring buffers and LRU caches. Additionally, there are eBPF maps which are used for storing specific kernel data such as cgroups, stack traces, socket references, and inodes. Maps are fixed size and can be used to store custom data structures.

This dramatic expansion of the eBPF feature set enables developers to monitor or change the behavior of most parts of the Linux kernel. As such, changes which could previously only be accomplished using custom built kernels, waiting for implementation in the mainline kernel, or through kernel modules can now be implemented with eBPF. In addition to allowing more rapid development and quicker adoption, implementing

kernel modifications using eBPF provides key stability guarantees. While bugs in custom kernel or kernel module implementations can lead to unstable kernels with kernel hangs or panics, eBPF programs undergo validation to establish program safety [29]. Validation is required at runtime and enforced automatically by the kernel significantly easing the process of developing custom kernel behavior. Validation is discussed in greater detail at the end of this section.

For clarity I will exclusively use the term eBPF program to refer specifically to the code that is executed in the kernel eBPF virtual machine. I will define eBPF systems as both the userspace control program and any eBPF programs it attaches to the kernel. So, an eBPF system for monitoring the performance of attached drives would consist of a userspace control program which compiles and attaches eBPF programs to each device on the system and consumes and acts upon the device data added to eBPF maps.

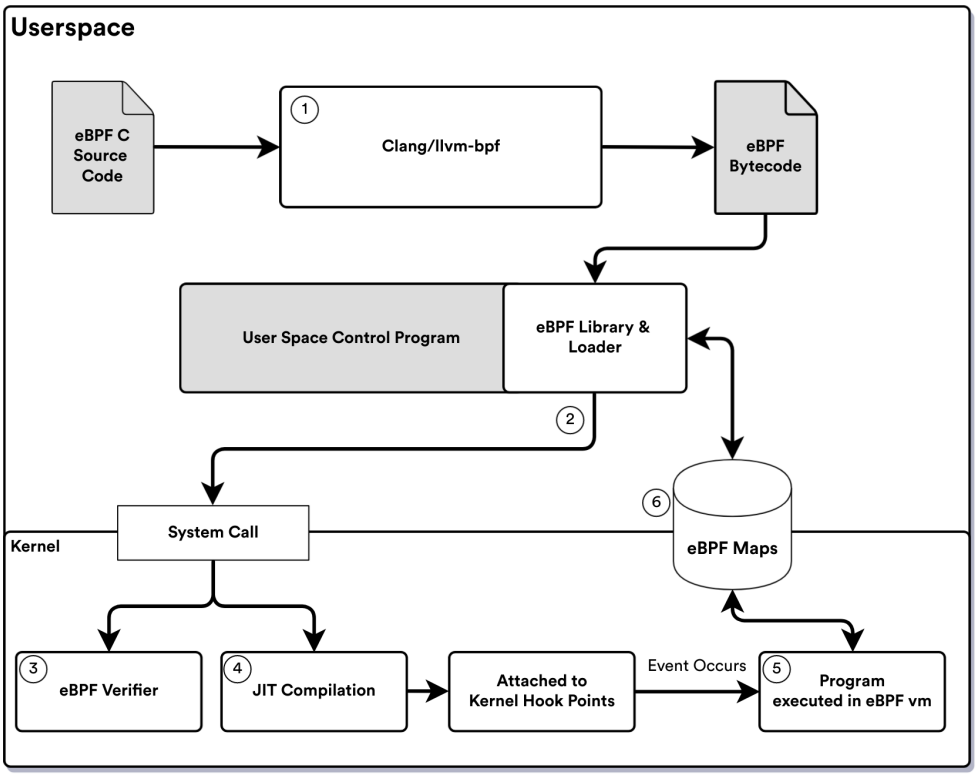

Figure 1-3: System diagram of a generic eBPF system. User developed components of the system are indicated by grey shading.

Figure 1-3 provides a system diagram for a typical eBPF system and the reference numbers are expanded upon below.

1. eBPF programs are typically written in C and compiled to eBPF bytecode using tools such as Clang or other frontends for llvm-bpf.

2. eBPF bytecode is loaded into the kernel using loaders which interact with the kernel through the `bpf()` system call. Some loaders will integrate steps 1 and 2 or perform changes to the bytecode before loading. This is done to provide program portability between different kernel versions.

3. The kernel runs the eBPF verifier on the loaded bytecode and approves or rejects it.

4. If approved, JIT compilation is performed to compile eBPF bytecode to native machine code.

5. When the kernel reaches a hook point the virtual machine is initialized, pointers to structures specified by the hook are passed and the program is executed.

6. eBPF maps are created by the userspace control program and can be accessed and modified by the eBPF program and other userspace processes.

Certain eBPF kernel hooks, especially kprobes, can require interacting with internal kernel data structures. As these data structures are not part of the kernel userspace API (uAPI) they are unstable and subject to change between kernel versions. For example, fields might be reordered, renamed, or change data types. As such eBPF programs and especially programs compiled to eBPF bytecode are not necessarily portable between different kernel versions.

eBPF validation is performed by static analysis of eBPF bytecode before JIT compilation to native machine code along with a limited number of runtime checks. The key properties validation attempts to establish for eBPF programs are memory safety and guaranteed termination [27]. For the memory safety property, the eBPF verifier ensures that the program only accesses memory locations within its allocated

regions. Programs with potentially out of bounds reads or writes to memory are rejected by the verifier. To prevent leaking other kernel internal state eBPF programs cannot access kernel data structures unless they are explicitly passed to the program by the kernel hook. Finally, eBPF programs cannot read uninitialized registers or stack locations. To validate program termination the verifier explores all reachable execution paths ensuring that they safely terminate without errors. To prevent hanging the verifier there is a maximum number of instructions and complexity limit. The complexity is defined as the total number of instructions checked by the verifier over all unpruned execution paths. Programs which reach this limit are rejected [38]. The complexity limit is not a significant constraint on eBPF development as most programs are relatively short and more recent kernel versions have significantly raised the limit. The eBPF verifier originally required loop unrolling but in recent kernel versions bounded loops are supported by the verifier.

The safety guarantees of validation do not prevent eBPF programs from producing unwanted behavior. A packet filter could still drop all packets which might be undesirable, however, the kernel will not crash while doing so.

The security model of eBPF is poorly defined. By performing verification in the kernel when loaded rather than at compile time the compiler is not included in the trusted computing base. This prevents buggy compilers or malicious bytecode from violating the safety guarantees of the validator. The majority of eBPF programs require root privileges or on newer kernels the CAP_BPF capability. However, there are a small subset of networking hooks which can be used by unprivileged users [20]. The verifier prevents programs from reading kernel memory not passed by the chosen hook but nearly any portion of kernel memory can be observed with the choice of the correct hook point. All eBPF map types require privileges to access, however there is no isolation of maps between processes. Any process with the required privileges can access and potentially modify any eBPF maps on the system. Due to the limited scope of eBPF safety verification, uncertain security model, and the extensive system access provided with eBPF it is not advised to execute untrusted eBPF programs.

1.1.3 Application of eBPF to Kubernetes

Kubernetes operates in a fundamentally different manner than traditional server applications that the Linux kernel was originally developed to support. As such there are a wide variety of Kubernetes functionalities which could be simplified, made more performant, or otherwise improved by modifying kernel behavior. While new functionalities used for containerized environments can, and are often, implemented in the kernel this can be a time consuming process with long lead times before mainstream adoption. eBPF allows for implementing new functionality to better support this application much more rapidly. It also allows for the creation of kernel functionality specialized for Kubernetes which would not be suitable for inclusion in the mainline kernel without kernel modules or developing custom kernel versions. eBPF programs can also be developed with techniques to make them portable to different kernel versions while custom kernels can require significant resources to maintain and keep up to date with the mainline kernel. This allows for rapid kernel development specialized for the Kubernetes environment.

eBPF can significantly improve the performance of many different tasks. For monitoring eBPF directly accesses data which otherwise might require expensive layers of redirection to inspect. eBPF based network implementations can prevent unnecessary work by the kernel required to support more general uses. One such application is in packet filtering where eBPF can act upon packets as soon as they reach the kernel, or even at the network device level and drop them if necessary before they are further processed. In a server environment these performance benefits can be significant when considered at scale.

The safety guarantees provided by eBPF also make it well suited for Kubernetes environments. eBPF verification provides end users of eBPF based systems a limited safety guarantee without the user needing to extensively understand the system's implementation details. Rejecting programs which could crash or hang the kernel also provides a guard rail for developing eBPF programs without the complexity of manually integrating static analysis within the build process.

Chapter 2

Existing monitoring landscape

There exists a multitude of approaches for monitoring Kubernetes clusters. I will provide a brief overview of several general approaches and evaluate their suitability for our use case of developing a monitor for existing clusters with a limited target feature set.

2.1 Distributed tracing

Distributed tracing is an approach to instrumenting services for profiling and monitoring microservice based applications. Similar to tracing in traditional applications, the goal is to create and collect low level information on the execution of a program such that requests and tasks can be followed throughout their entire lifetime within the program. This information can be used to monitor the health of an application and aid in the debugging process. For example, distributed tracing could be used to identify services which are acting as bottlenecks, suggesting to developers a target to optimize or scale up the deployment of.

In a microservice based cluster configuration the task of developing the infrastructure necessary to track a request along its dependencies on many different services can be difficult. APIs and tools such as OpenTelemetry [16] and Jaeger [14] allow for more easily creating sophisticated tracing of applications across microservice architectures. The primary disadvantage of distributed tracing approaches is that

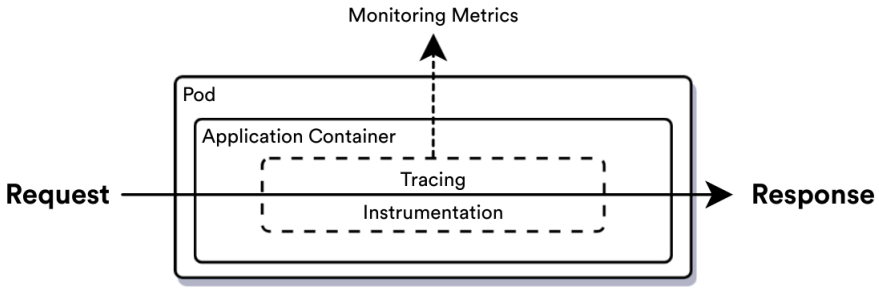

Figure 2-1: Observing an application using distributed tracing. Tracing instrumentation is manually added within the target application.

they require manual instrumentation of the targeted application's codebase. As such adding distributed tracing to existing microservice applications can be a lengthy and development-intensive process. In cases where it is not possible to instrument code such as in third party closed source binaries then shims can be developed to instrument entry and exit. However, shims cannot provide the same level of detail as first party instrumentation. An additional disadvantage of distributed tracing is that tracing frameworks are not fully standardized so switching to a new product or framework may require re-implementing instrumentation across the entire codebase which leads to lock-in.

Service meshes which will be discussed next can simplify adding tracing to microservices by essentially acting as a shim. However, this approach still requires minor codebase changes to forward the correct request contexts to the service mesh [13].

2.2 Service meshes

Service Meshes are a broad category of Kubernetes tools which typically aim to improve the observability, security, or reliability of Kubernetes systems. Unlike distributed tracing, service meshes operate at the platform level and are agnostic to the specific workloads running on the cluster. Service meshes can be deployed to clusters without application codebase changes, providing a significant usability benefit over distributed tracing.

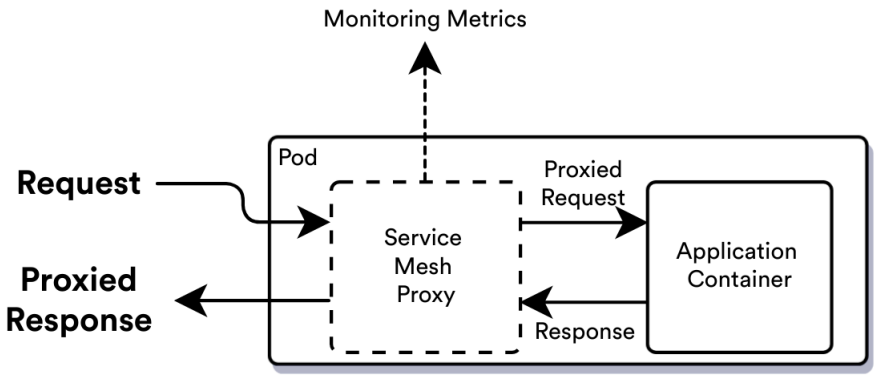

Figure 2-2: Observing a Kubernetes pod using service meshes. A sidecar captures and proxies network communications in order to observe application traffic.

Fundamentally service meshes are systems which introduce network proxies between workloads in a cluster to capture network data. Traditionally service mesh proxies are implemented as sidecars in which all connections for a pod are redirected through that pod's sidecar proxy. With access to all network data passing throughout a cluster, they can perform tasks such as load balancing, encrypting communications, or monitoring network activity. For network communication between services that are not encrypted at the application layer, service mesh proxies can capture unencrypted communications and provide application level observability. As there are many different service meshes available, a brief overview of two popular service meshes, Linkerd [15] and Istio [12], and their monitoring features is provided.

Linkerd is aimed at being lightweight and high performance. It utilizes a special purpose proxy linkerd2-proxy and has a restrained feature set compared to other service mesh offerings. While Linkerd does not provide a system for users to create custom metrics, it does provide a set of monitoring features by default. A subset of which is provided below.

1. Application level metrics. Request volumes, HTTP or gRPC response codes and success rates (based on response codes).

2. Transport level metrics. The number of TCP connections opened or closed, the

total number of bytes sent or received over TCP, and the length of time TCP connections have been open.

3. Protocol level metrics. The total number of requests, responses, and the response latency.

4. Kubernetes workload labeling. The previous metrics are labelled with the Kubernetes workloads associated with them.

Istio has a more expansive feature set than Linkerd and uses the general-purpose proxy Envoy. Istio provides an API for implementing custom metrics with access to all standard Envoy attributes, which allows users to better tailor monitoring with Istio to their specific applications. By default, Istio includes monitoring for similar metrics as Linkerd in addition to metrics on the Envoy proxy health such as uptime and memory usage. Istio typically has higher performance overheads to provide these features.

The primary drawback to service meshes is that they are heavy solutions, especially if the primary features of interest are monitoring. Every pod on a cluster requires its own dedicated sidecar proxy which in service meshes such as Istio are fully-fledged general-purpose proxies. This results in higher latency as all network traffic on the cluster must be redirected through two proxies. This also leads to higher resource utilization, particularly memory usage as there is an individual proxy running for every pod on the cluster. An analysis by Andrew Wei found that for a microservice app running on minikube under a simulated load Istio increased memory usage by approximately 20% when compared to the same load without a service mesh [42]. While this can vary dramatically based on the cluster configuration and deployed workloads it serves to demonstrate the resource overhead of service meshes.

There are projects attempting to implement service meshes without sidecars to improve performance. One project to note is Cilium's service mesh (currently in beta) [9] which uses eBPF to perform some features of service meshes directly in kernel and for unsupported features falls back to a proxy-per-node model as opposed to the proxy-per-pod model of sidecars. Using eBPF, the Cilium service mesh captures

packets at the socket level and either acts upon them or directly passes them to the per-node proxy. This avoids the need for packets to fully traverse the network stack before being redirected to a sidecar proxy for monitoring only to traverse the network stack a second time. In a latency test by Cilium of an eBPF based monitoring approach compared to a traditional proxy-based approach it was found that the eBPF based approach had performance near the unmonitored control. The proxy-based approach had approximately four times larger latencies [30]. This demonstrates that eBPF based monitoring approaches can provide significant performance gains.

Another important caveat is that since service meshes typically use TCP proxies they do not support non-TCP based protocols. While other protocols such as UDP are less commonly used in Kubernetes clusters they are supported natively. Service meshes' inability to operate on such traffic would prevent their application for some use cases.

2.3 eBPF based monitoring systems

As discussed eBPF can be used to perform highly efficient monitoring of Kubernetes clusters. Three monitoring projects to note are Cilium Hubble [8], Weave Scope [41], and Pixie [34] which all heavily utilize eBPF.

Cilium Hubble is a monitoring tool integrated with the Cilium CNI plugin. Cilium attaches eBPF programs to various points in the Linux networking stack to implement the Kubernetes networking model and additional features such as network policy enforcement. Cilium provides high performance XDP based packet filtering. Additionally, Cilium attaches to the socket layer to provide features such as transparently injecting proxies or allowing pods to directly connect to pods on other hosts avoiding performing expensive network encapsulation operations for each packet. Network policy is enforced by embedding the source workload identity in all cross-node packets allowing ingress filters to securely identify packet sources and apply security policies. By integrating itself into many layers of the networking stack for core functionality it is also easily able to collect data on the behavior of these systems. The data generated

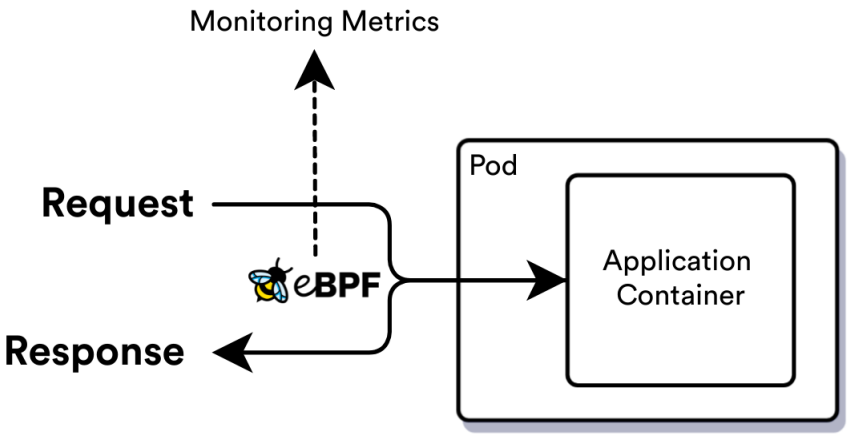

Figure 2-3: Observing a Kubernetes pod using eBPF. The eBPF program captures pod traffic as it traverses the host network stack.

by the Cilium CNI plugin is utilized by Hubble to produce metrics such as the number of HTTP requests and DNS queries, and the rate of dropped packets. Hubble also creates visualizations of service dependency graphs, which allows users to more easily understand the dataflows of their application.

Cilium Hubble depends on the entire Cilium project. As Cilium is primarily a networking plugin not a monitoring solution, for many users, the observability features are an ancillary benefit of deploying Cilium and often not a sufficient reason to deploy Cilium alone. As discussed, Cilium implements security features and a drastically different networking implementation. This would also likely serve as a barrier to utilization if customers' existing cluster configurations are incompatible with these products or they deem the performance overheads and installation complexity too great for the monitoring features alone.

Finally, since Cilium utilizes a number of different eBPF features the minimum supported kernel version for the entire project is higher than might be required for some metrics alone. A more modular approach would also allow users with less recent kernel versions to use a subset of supported eBPF monitors.

Pixie and Weave Scope are both more monitoring focused systems. However, like Cilium, they are also tightly integrated systems where monitoring for specific

components cannot easily be disentangled from the overall system.

Weave Scope is a real-time cluster monitoring and management system. eBPF programs are attached to pod TCP sockets to capture the source and destination of current traffic. This is used in the Scope UI to generate real time cluster topologies. Historical data is not saved limiting the use cases of the collected metrics and making the tool primarily focused on real time management. A plugin system can be used to generate custom metrics from the captured HTTP traffic. The Scope UI is either self-hosted or cloud based, and the data collectors cannot easily export to external services or be used without the UI. This makes Weave Scope unsuitable for our goal of a monitor which can be integrated into existing monitoring systems.

Pixie more extensively instruments the target systems compared to Weave Scope and generates historical data. eBPF kprobes are attached to networking system calls to capture network traffic. eBPF performance events which capture stack traces are created and triggered by a set timer. This is used to generate flame graphs of running workloads. Pixie also uses uprobes to provide observability to the state of running workloads at user specified instructions. Finally, Pixie can deploy user created bpftrace probes to clusters and aggregate the collected data. This provides a wider and more extensible feature set than Weave Scope and can be exported to external tools such as Grafana.

Cilium and other commercial eBPF based monitoring solutions demonstrate the usefulness of an eBPF based approach for implementing monitoring in Kubernetes clusters. Our use case was to utilize eBPF to implement a network monitor capable of functioning on most kernel version and cluster configurations, and integrating into existing cluster monitoring product offerings. Existing solutions were not suitable for this application as monitoring for specific metrics was not done in a modular or self-contained manner. As such to use any metrics provided by these solutions customers would be required to deploy the entire projects to their clusters.

In addition to being tightly integrated systems both Weave Scope and Pixie make extensive use of the BPF Compiler Collection (BCC). While this toolkit will be discussed in greater detail in section 4.1 it is not ideal for Kubernetes environments and

has higher resource overheads than alternative approaches. To address the concerns identified with existing eBPF monitoring systems, we developed a methodology for creating self-contained eBPF monitors.

Chapter 3

Goals and requirements

The goal of this work was to investigate approaches to monitoring Kubernetes clusters using eBPF and to evaluate supporting eBPF toolkits and technologies for their suitability for this purpose. This evaluation informed the development of a methodology for monitoring the network connections between pods in Kubernetes clusters using eBPF in a self-contained manner. Using this methodology, a proof of concept network monitor was implemented. The core functionality of this prototype was collecting network packets from Kubernetes pods, correlating them to specific Kubernetes resources, and integrating these metrics into an existing product.

The target audience of this work is developers and administrators of Kubernetes clusters who could use this approach to develop their own targeted monitoring solutions for components of interest in their Kubernetes clusters. The approach is aimed at developing long lived, lightweight data collectors for "set and forget" style uses that require little maintenance or configuration once developed.

The proof of concept of our method specifically targeted using eBPF for collecting metadata on the number and size of packets sent at the granularity level of specific pods. In addition to troubleshooting scenarios, a system for collecting this network information could also be useful as the basis for deeper analysis of cluster behavior. For example, the information collected by this system could be used to analyze the interactions between different microservice pods to determine which workloads should be collocated on the same nodes to improve performance.

The following goals guided the evaluation of eBPF toolkits and technologies and development of a monitoring methodology.

1. Minimize resource usage. Of particular interest were memory usage and startup time as CPU resource usage at steady state should be similar between different eBPF approaches. The overall disk size when containerized was less of a concern although still considered.

2. Minimize Dependencies. The approach should be self-contained to the degree possible and should not require other specific monitoring solutions or dependencies not typically installed on production clusters.

3. Portability Support. A wide range of kernel versions should be supported in a general fashion (to the extent possible as kernel eBPF support is a limiting factor) so as not to require significant development work to support specific kernel versions. Features unique to specific CNI implementations should be avoided. Additionally, this solution should be easily deployable to existing Kubernetes clusters with little to no configuration required.

When evaluating the existing commercial solutions with respect to these design goals they were found unsuitable for this application. Distributed tracing approaches, while powerful and extremely targeted, require development time to instrument existing codebases or to create shims which fails the portability goal. Service meshes provide extremely rich feature sets and would be resource heavy. This fails the goal of minimizing resource usage. Additionally, customers might have existing service meshes or not desire to install a service mesh which also fails the portability goal. Finally existing eBPF monitoring solutions while high performance have many of the same concerns as service meshes for not being targeted and introducing large dependencies. A custom eBPF monitoring system is required for supporting our use case within these constraints.

3.1 Use cases

eBPF monitoring was investigated specifically through the lens of network monitoring as the metrics generated could be used in multiple interesting ways.

A potential application of this work could be identifying and debugging performance degradation in running systems. Take the scenario of a production Kubernetes cluster without any existing network monitoring which suddenly experiences performance degradation. An investigation of the problem reveals that a service, for example a database, utilized by numerous other workloads was experiencing a dramatically increased number of requests. Without any network observability it might be extremely difficult to identify the misbehaving workload responsible for the increased requests. A brute force approach might involve manually shutting down workloads and watching for a decline in requests. In this scenario a self-contained network monitor created using our approach could be deployed to the impacted cluster in minutes without impacting existing workloads or potentially obscuring the offending behavior. By analyzing the source of captured traffic to the impacted service, the culprit could be quickly identified.

The network metrics collected by the prototype monitor could also be used as the basis for performing security audits. A topical example is the recent Log4j vulnerability which allowed for remote code execution attacks on vulnerable machines [26]. An audit of network activity might reveal if an attack has taken place. Such an audit might use the collected network data to identify unexpected or suspicious network activity with external resources, or it might reveal a pod in a cluster communicating with workloads it normally would not access. All of which might indicate a potential breach, and could be investigated further. Automated audits could create alerts for changes in network behavior.

Finally, this approach could also be useful for collecting information on the usage of workloads. After deprecating a service, a network monitor could be deployed targeting nodes providing that service to quantify how often it is used. Once usage has stopped or is sufficiently low enough the service can be removed safely with less fear

of impacting other workloads. Usage information could also enable billing chargeback without needing to implement any functionality in the actual application.

Chapter 4

eBPF loaders and other tooling

eBPF toolkits and loaders simplify the development process and provide key functionality for creating eBPF programs able to run on real-world systems. As such a significant component of this work involved evaluating and selecting a suitable toolkit or loader. Most eBPF systems include tightly coupled userspace programs which compile the eBPF program to bytecode (if necessary), load the program into the kernel, attach the program to the desired hook, and process data returned by the eBPF program. eBPF toolkits and loaders provide this functionality along with portability between kernel versions. Portability is of special importance for this work as the approach must work on Kubernetes clusters of heterogeneous machines.

Several eBPF toolkits and loaders were evaluated using our design goals with a specific focus on approaches to implementing eBPF portability.

4.1 BPF Compiler Collection

The BPF Compiler Collection (BCC) [1] is a popular well-established toolkit. The core of the BCC toolkit is a compiler, loader, and an API for interacting with kernel eBPF features. BCC has one of the smallest barriers to entry for writing eBPF systems of the major toolkits. This is due to reasons both inherent to BCC and circumstantial. BCC provides abstractions to eBPF programs for developers to write programs in a manner much closer to traditional C programs. In many cases the BCC loader can

also automatically support eBPF program portability. The toolkit also includes an extensive set of premade tools and example usages. These tools can be incorporated into larger eBPF systems without requiring significant knowledge of eBPF or kernel programming. Finally, BCC was released years earlier than other toolkits and loaders analyzed and a significant amount of literature and tutorials have been produced discussing its usage.

BCC provides a scripting language frontend (Python or Lua) for the userspace control program with eBPF C source code embedded as a string within. BCC has a simple, well documented API for interacting with eBPF programs. This allows developers to implement business logic in a higher-level language while abstracting away many of the friction points of eBPF programs.

To provide portability BCC performs just-in-time (JIT) compilation of embedded eBPF programs. As with most approaches, compilation is performed by the llvm-bpf backend. JIT compilation on the target machine has significant advantages for creating portable eBPF systems. BCC eBPF programmers can utilize C preprocessor directives to tailor the eBPF program to the target machine. When handling network packets, preprocessor directives could be used to handle byte order correctly. Newly introduced features which are not supported by all targeted kernel versions can be included only if the machine is running a supported kernel version. BCC also makes use of code rewriting before compilation. eBPF programs access fields within kernel data structures by using the `bpf_probe_read()` function but BCC supports structure accesses with the standard `'->'` by rewriting the embedded source code before compilation.

The BCC toolkit and loader provides several advantages for monitoring Kubernetes environments. It would be infeasible to manually write, compile, and distribute monitors for each kernel version in the targeted clusters. BCC's JIT compilation is an automated approach to that process, simplifying deployment. The extensive documentation would lower the difficulty of developing and maintaining one off monitors. The proof of concept application could potentially leverage existing BCC tools to perform the actual collection of network data. For example, the existing tools for TCP tracing included in BCC might be incorporated. This would allow

for creating a robust tool which supports a variety of kernel versions with minimal expertise in eBPF. BCC supports a wide variety of kernel hooks and eBPF programs. Future iterations of a monitor could include eBPF programs which collect data from drastically different parts of the kernel using the experience gained during this work without needing to use an alternative toolkit.

The use of JIT compilation to support portability also has significant drawbacks which should be considered. A BCC based monitoring system would require Clang/LLVM to be installed on the target machine or packaged with the eBPF system. In a Kubernetes environment, if there were many different modular eBPF monitors installed on a cluster then there would be multiple redundant copies of Clang/LLVM on each node.

Another dependency which might cause problems is the requirement for kernel headers, which are not common on production machines. There would also be increased complexity to pass the installed headers to the containerized monitor. For example, to provide headers for compilation Pixie bundles a subset of kernel headers with the system. At runtime if headers are not installed on the machine the best matching of the bundled headers are selected. To ensure the dynamically selected headers are compatible with the running system the developers of Pixie manually curate the subset to include all kernel versions which modify the targeted data structures. This is a brittle approach and requires constant maintenance as new kernel versions are released.

JIT compilation can also be resource intensive and slow to start. This could impact the time to start new nodes or pods on a cluster. Finally due to both the scripting language userspace control program and JIT compilation, systems developed using BCC typically have higher memory usage than alternative approaches.

A BCC based Kubernetes monitoring approach would be most suitable for applications where ease of development is a primary concern, higher resource overhead is acceptable, and there is access to the target machines to install dependencies. One such use case might be a one-off monitor to debug a specific problem. As it will be used for a specific task and will not be necessarily used afterwards, being able to

quickly develop and deploy the system likely outweighs any performance concerns. BCC achieves the support design goal due to the robust portability provided by code rewriting and JIT compilation. However, the large memory and disk footprints and slow startup speeds do not achieve the goal of minimizing resource overheads. A BCC based approach also does not meet the goal of minimizing dependencies due to the requirement for kernel headers. As such BCC is not appropriate for this methodology and application of a long running monitor.

4.2 Kubernetes eBPF deployment tools

There are several tools which simplify the process of scheduling eBPF programs on nodes of a Kubernetes cluster. Two popular tools to note are Inspektor Gadget [28] and kubectl-trace [5]. These tools are useful for debugging applications but do not appear to be targeted towards developing long running eBPF systems on top of. They do not allow for custom userspace control programs which complicates the process of aggregating the data collected by the monitors and forwarding it to external services. These tools also have limited feature sets. For example, Inspektor Gadget only allows for only a small number of supported eBPF programs, called gadgets, to be deployed on the cluster. Additionally, both tools mentioned make use of BCC to implement functionality and do not meet the goals of minimizing dependencies and performance overhead for similar reasons. For these reasons we did not evaluate Kubernetes eBPF deployment tools further.

4.3 BPF CO-RE & libbpf

BPF Compile Once Run Everywhere (CO-RE) [39] is a modern technique for developing portable eBPF programs. While not a specific toolkit or loader the approach will be described here in conjunction with the loader libbpf [6]. Libbpf is generally considered the reference implementation for BPF CO-RE loaders. BPF CO-RE makes use of new kernel and compiler features to create portable eBPF bytecode as opposed to

individually compiling for target machines. This enables BPF CO-RE systems to achieve significantly better performance and lower memory footprints when compared to analogous eBPF systems implemented using BCC. There is an ongoing effort to convert existing tools provided by BCC to BPF CO-RE. This has reduced the memory footprint of some tools by nine times [43]. BPF CO-RE programs do not generally require special support from the kernel as binary portability is implemented by the userspace loader transparently to the kernel.

The core technologies used in developing BPF CO-RE programs are kernel BPF Type Format (BTF) [36] information, LLVM reloactions, and the libbpf loader. BTF information is generated by the kernel and includes information on most kernel functions and data structures replacing the build requirement of kernel headers. BPF CO-RE loaders use BTF information to modify eBPF binaries directly to run on the target system.

One of the core problems for eBPF program portability is that programs can access unstable kernel data structures. BPF CO-RE solves this problem with CO-RE-relocatable accesses which translate intended accesses to their actual underlying layout in memory. The LLVM-bpf compiler backend produces a binary ELF file with BTF relocations capturing information on the intended field access. At runtime libbpf takes the compiled eBPF ELF object and BTF information provided by the running kernel and using the LLVM relocations modifies the memory accesses to match the running kernel's data structures.

This approach is not able to provide relocation for all cases and ways in which kernel data structures might change between versions. Fields which are renamed or removed entirely between kernel versions are not relocated. Fields might also change in use or meaning such as changing the units between versions without changing name. CO-RE cannot identify these cases and programs accessing these fields would not be portable. In these cases, the eBPF developer needs to identify and address these changes manually. Libbpf provides features to aid in this task such as accessors and comparators to the running kernel version, access to userspace provided configuration variables, and macros to check for field existence and type. These approaches provide

analogous functionality to BCC's use of preprocessor directives without requiring JIT compilation.

The BPF CO-RE approach requires the kernel to provide BTF type information but otherwise does not require more modern kernel versions than BCC based approaches for most cases. BTF support was added to kernel version 4.18 and at the time of writing there are only two remaining long term support kernel versions which do not support BTF information [18].

BPF CO-RE is particularly useful for kernel tracing applications as they often require interacting with unstable kernel data structures. For some other applications access to unstable kernel data structures is not necessary. For instance, monitoring components of the userspace API does not require CO-RE for portability as the API is stable. In some eBPF program types, particularly in networking, the kernel provides stable views of the underlying unstable data structures which also avoids the need for CO-RE-relocatable reads. While portability might not necessarily be required, libbpf introduces minimal overheads and would still provide a useful API for interacting with and loading eBPF programs.

The primary disadvantage of the CO-RE technique is that it is a relatively new approach relying on modern technologies. There is less literature available on developing eBPF programs with the CO-RE approach when compared to the more established BCC toolkit. There are several eBPF loaders other than libbpf which implement varying levels of CO-RE support and it can be difficult to evaluate their feature sets in comparison to the libbpf standard. Some eBPF program types, such as networking programs, were introduced well before BTF support. This would prevent using a CO-RE approach if versions earlier than 4.18 are required to be supported.

Libbpf and BPF CO-RE are well suited for Kubernetes environments. The approach has no external runtime dependencies and implements eBPF portability with minimal performance overheads. For applications which require monitoring unstable components of the kernel, and the target machines are at least kernel version 4.18 a BPF CO-RE approach is highly recommended.

4.4 Cilium/ebpf

The final toolkit evaluated was Cilium/ebpf [2] which is a pure Go library. The library provides a loader and functionality for compiling and embedding eBPF programs within userspace Go code. Implementing the toolkit in pure Go allows for the userspace control program to be written in Go without the performance impact of calling cgo code. This is particularly helpful in Kubernetes environments as interacting with the Kubernetes API and external data collectors is often simpler in Go than in C as is required by libbpf. The toolkit attempts to minimize the manual boilerplate code required for interacting with eBPF programs by autogenerating Go interfaces to the eBPF programs and maps used with the control program. By embedding the eBPF program directly into the userspace control program it avoids manually loading the program from disk at runtime.

The Cilium/ebpf loader supports BTF information and CO-RE-relocations and can be used for BPF CO-RE programs. To note however, documentation on Cilium/ebpf CO-RE compatibility is limited and the entire feature set available in libbpf may not be available. Similarly, a drawback to Cilium/ebpf's pure Go approach is that a significant amount of re-implementation is required and there is not yet support for all eBPF kernel hooks supported by libbpf or BCC. If hooks or features supported by libbpf and not Cilium/ebpf are necessary and developing the userspace control program in Go is still desired, libraries which provide a Go wrapper for libbpf using cgo such as libbpfgo could be employed instead.

As with libbpf and the BPF CO-RE technique in general, Cilium/ebpf meets the design goals and is well suited for developing long running Kubernetes monitors with. Cilium/ebpf is an excellent choice for eBPF systems which require interacting with external systems which could be simplified with the use of higher-level languages and for which sophisticated libbpf features are not required.

Toolkit	Resource usage	Dependencies	Portability	Notes
BCC			X	Ease of Use
K8 eBPF Deployment Tools			X	Good for uses such as debugging
Libbpf	X	X	X	More complex usage, good for monitoring unstable components
Cilium/eBPF	X	X	X	Limited support for hook points, Go userspace, easier integration with K8s

Table 4.1: Table eBPF toolkits and loaders and evaluation for our goals.

4.5 Our approach

The choice of supporting eBPF toolkits and loaders used to develop the prototype network monitor was highly informed by our specific application of collecting network traffic. The network subsystem is a particularly mature component of the kernel with stable eBPF abstractions. This can avoid many of the portability difficulties present for eBPF programs which interact with unstable components of the kernel and does not require BPF CO-RE-relocations.

An eBPF hook in the Traffic Control (TC) subsystem, specifically TC-BPF direct-action, was selected for packet capture. TC-BPF direct-action collects packets from the system using TC queuing disciplines. This choice allows for collecting data from all ingress and egress packets on a network interface from a single eBPF program. This avoids the need to develop eBPF programs for each targeted protocol as would be necessary for protocol socket listener approaches. The Linux kernel also provides a stable mirror `__sk_buff` of the socket buffer to TC-BPF direct-action programs. Transparently to the eBPF developer the kernel rewrites accesses from `__sk_buff` to the underlying potentially unstable `sk_buff` data structure. New fields will only be added to the end of this structure and existing fields are stable so working with `__sk_buff` does not require compiling for specific kernel versions or using portability approaches such as BCC or CO-RE compatible loaders. Additionally, TC-BPF direct-

action can perform packet filtering so the work performed for this traffic monitor could be easily extended for filtering in the future.

As discussed above, BCC and Kubernetes eBPF deployment tools were not suitable for this application. Of the two remaining loaders analyzed, libbpf and Cilium/ebpf, we chose to use Cilium/ebpf. Cilium/ebpf supports the TC-BPF direct-action hook. The ability to write the userspace control program in Go enabled easy interfacing with existing data pipelines and interacting with host Kubernetes clusters using the first party Kubernetes client-go library. Additionally, Cilium/ebpf's support of embedding eBPF programs within the userspace control program to produce a single binary allows for easy distribution. Libbpf would also have been suitable for this application. However as CO-RE portability and libbpf specific features were not required, the added complexity of interacting with the libbpf API was unnecessary.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Implementation details of a prototype network monitor

5.1 Overview

The eBPF system follows a multiple-producers single-consumer model. Figure 5-1 depicts a system diagram of the monitor. eBPF programs are attached to network interfaces for every pod by the userspace control program. These programs capture packets from the Traffic Control subsystem. Upon capturing a packet the socket buffer is parsed to extract the source and destination IP addresses, ports, and packet size. This data is populated in an eBPF map and the eBPF program returns control to the kernel and regular network processing resumes. This data is then aggregated, enriched, and sent to external monitoring systems by the userspace control program.

The Cilium/ebpf library was used for interacting with eBPF objects in the control program. The actual loading of the eBPF program was done using Vish Abrams' "netlink library for go" [7]. This library was used to create TC queuing disciplines (qdiscs) on target network interfaces which the embedded eBPF program attaches to. The Cilium/ebpf utility `bpf2go` was used to compile the eBPF system to a single self-contained statically linked binary. This binary can be run on target systems directly but was containerized for ease of use in Kubernetes environments. A DaemonSet was created to deploy the monitor to target clusters.

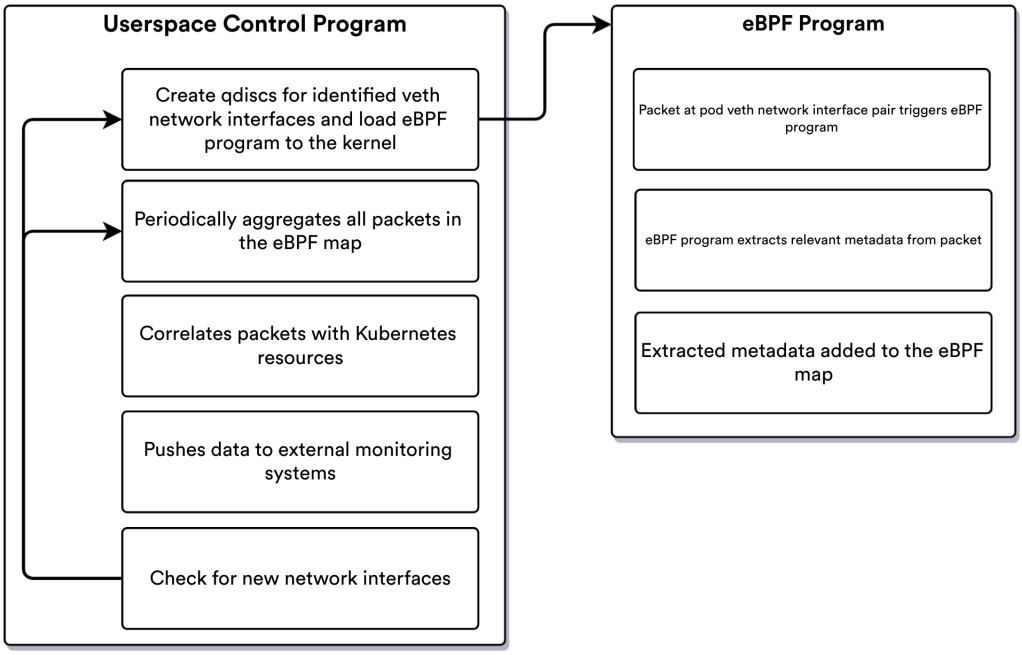

Figure 5-1: Overview of the prototype eBPF monitoring system.

5.2 Kubernetes networking

Selecting a suitable network monitoring point was an important implementation design choice and required a more detailed understanding of the Kubernetes networking stack in practice than the abstracted model discussed in section 1.1.1. Every pod (excluding pods running on the host network) exists in its own network namespace with a unique virtual IP address (cluster IP). To provide networking to the pod namespace most CNI plugins create virtual ethernet (veth) pairs for each pod. One interface is added to the pod namespace and the other interface is added to the host namespace acting as a tunnel between the namespaces. CNI plugins then direct traffic on a pod's host veth interface to other pods in a variety of different manners. Virtual network bridges are often used to connect all pod veth interfaces on a node to enable communications between different pods on the same node. This is depicted in figure 5-2.

While most CNI plugins follow the pod veth pair approach it does not necessarily hold for all cases. The Cilium service mesh is one example where not all pod traffic is routed through a pod veth pair. In this case traffic which must be proxied is captured

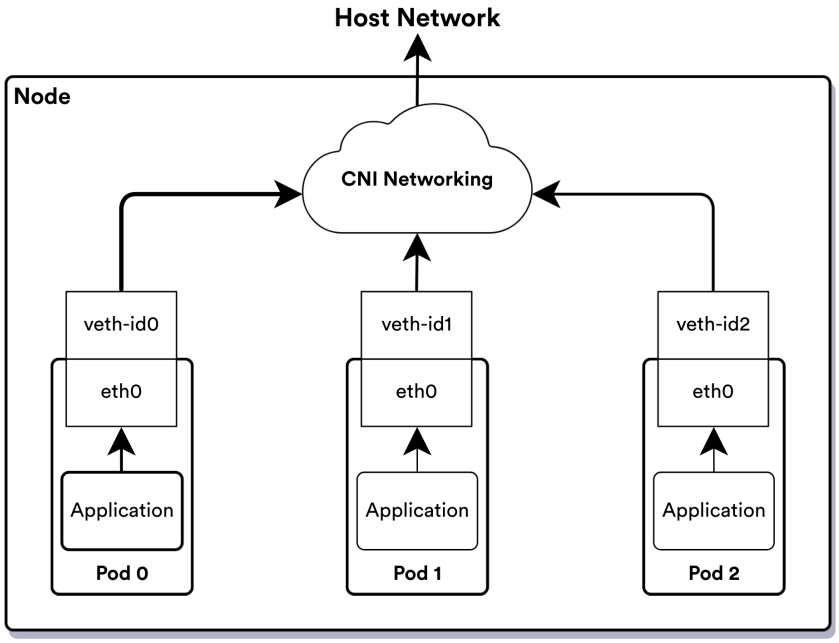

Figure 5-2: A system diagram of egress traffic in a node with a generic CNI plugin which uses pod veth pairs.

at the socket level by the service mesh and directly attached to the socket listeners on the per-node proxy. This captures traffic before it traverses the pod veth pair.

CNI plugins vary far more in approaches to enabling communication between pods on different clusters. If supported by the underlying host network many CNI plugins can be configured to directly route Kubernetes packets on the host network. If this is not possible an overlay network is created. IP in IP tunneling, VXLAN and Wireguard network virtualization are common approaches. These approaches encapsulate cross-node communications making it more difficult to inspect and correlate this traffic with specific Kubernetes workloads if captured on the host network interface.

As one of our goals was to support a wide variety of cluster configurations it was necessary to support both direct routing and overlay networks. This presented a tradeoff between the ease of capturing packets and the ability to correlate those packets with Kubernetes workloads. The naive approach would be to attach to the network interface of the host node. However, for overlay networks the source and destination addresses would be the actual IP addresses of the nodes hosting the workloads, not the

virtual cluster IP addresses. While it is likely still possible to correlate these packets with specific Kubernetes workloads it significantly increases the complexity of the monitor and solutions might have to be created for each supported overlay approach or CNI. Additionally, this would only capture packets sent between pods on different nodes. Packets between pods on the same host node would traverse the container network bridge and would not reach the network ethernet interface.

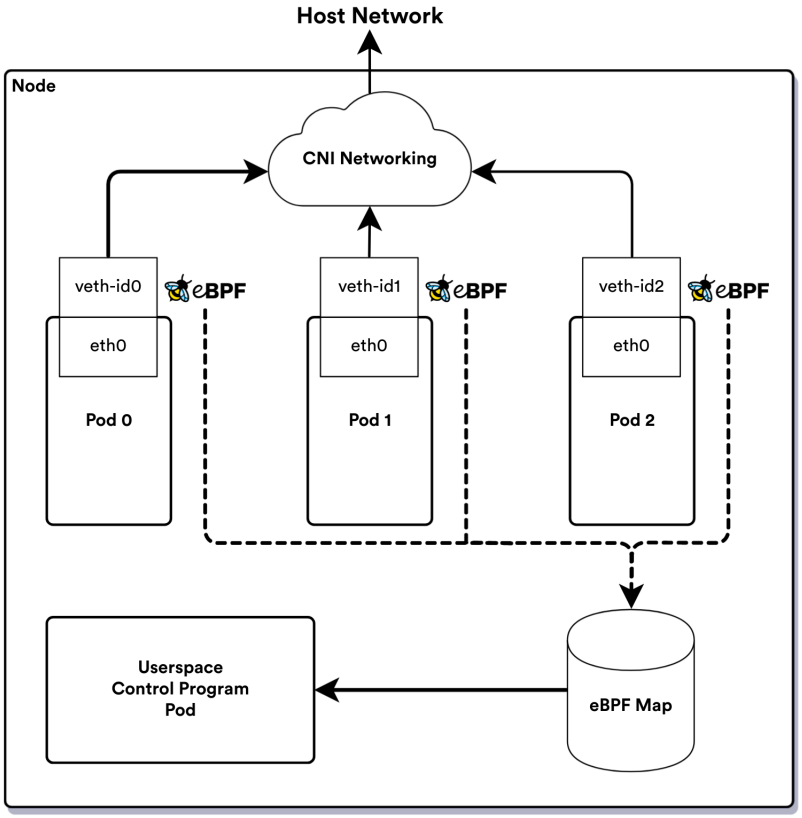

Figure 5-3: Instrumenting a Kubernetes node with eBPF pod veth interface packet monitors.

Pod veth pairs provide a convenient point to attach the eBPF data collectors. They are present in and behave similarly for most CNI plugins. Packets captured at this point have not yet been encapsulated for overlay networks and still contain virtual cluster IP addresses. From a pod veth interface it is possible to identify the pod it is connected to which also aids in correlating captured packets. Veth pairs often follow a consistent naming scheme making it easy to programmatically identify

pod veth interfaces in the host network. For these reasons, the prototype monitor attached eBPF programs to all pod veth interfaces in the host network namespace to capture traffic. The node depicted in figure 5-2 is shown in figure 5-3 with eBPF veth interface monitoring.

5.3 Userspace control program

The userspace control program followed a simple control loop shown in figure 5-1 in the overview.

1. There is a configurable polling period where the monitor will wait for the eBPF programs to fill the eBPF map.

2. All captured packet information is pulled from the queue and a sum of the total number of packets and data transferred between each connection is computed.

3. The sources and destinations are correlated to known Kubernetes workloads by the IP address and port if possible.

4. This summary of network connections is then pushed to external monitoring pipelines.

5. The host network namespace is scanned for new pod veth interfaces. If any are found TC qdiscs are created and the eBPF program is attached to the new interfaces.

Pod veth interfaces are identified by periodically scanning the available host network interfaces for new interfaces which match the naming pattern for pod veth interfaces. For instance, with the Calico CNI plugin all veth interfaces are named `cali[id]` where id is a unique hex string. This pattern is currently supplied to the control program through command line flags. To avoid manual specification by users the patterns for most major CNI plugins could be collected and included in the control program to automate this process. When pods are unscheduled the pod veth pair and namespace

are destroyed which removes our queuing discipline and eBPF program automatically. This approach attaches and removes the eBPF program automatically as the cluster changes and pods are created and destroyed.

Packets are correlated to Kubernetes workloads according to the source and destination IP addresses and the port numbers (for services). Workload addresses and ports are collected by querying Kubernetes for all pods and services. This is parsed to create a workload lookup table. Some Kubernetes DNS implementations support reverse-dns queries which would simplify this processes, however it is not broadly supported. The collected data is then labelled with the identified workloads. To avoid repeated calls to the Kubernetes API it is recommended to cache workload addresses when collected. The time to live for cached information should be user customizable as more dynamic clusters with workloads which are more frequently created and destroyed would require a shorter caching duration than clusters with longer lived workloads. For the lookup table approach the cache should not be refreshed on misses as misses can be frequent and refreshing a single value is equivalent to refreshing the entire table. As Kubernetes does not immediately reallocate IP addresses for reuse after pods are destroyed a reasonable caching duration would not cause packets to be misattributed to destroyed pods. However, too long of a caching duration could prevent packets from being correlated with any workload. This is a minor concern as caching durations could be adjusted to the specific cluster if a considerable number of packets are uncorrelated to Kubernetes workloads.

When aggregating the packet count and size there are a few cases to consider as opposed to simply summing the quantities. As packets are captured on the veth pair of every pod there is double counting of packets between pods. This information could be used to detect packet loss in a network even in connectionless protocols such as UDP. If traffic collected by one network monitor is not collected by the network monitor on the destination node it could be considered lost. When aggregating any traffic between two Kubernetes pods there should be deduplication to avoid double counting. A sensible deduplication technique would be to use the minimum of the two counts and include a packet loss metric of the difference. Traffic between a pod

and an external resource does not require deduplication as it is only captured by the source node's monitor and can be aggregated directly.

The case of pods operating in the host network namespace is more complicated as they do not have veth pairs and use the host network interface directly. If collecting packet information from pods operating on the host network is also desired this can be done by attaching the eBPF program to the host network interface. However, it is difficult to differentiate packets from processes on the host machine from Kubernetes packets as they share the host IP address. If a pod is assigned specific ports, then this could be used to correlate traffic from the pod separately from processes on the host. There are also additional double counting issues as traffic between two pods on different nodes will be counted both on the pod veth interface and on the host network interface. As the traffic might be proxied or encapsulated, it is difficult to correlate with specific Kubernetes workloads to address this overcounting problem. An inelegant solution is to simply exclude all uncorrelatable packets captured from the host interface.

This monitoring platform is extensible and can be easily used to implement additional metrics. Approaches to generating latency metrics were also investigated. In this case latency was defined as the time between a message being sent to a service and a response being received from that service. One simple implementation involves timestamping packets when captured. When aggregating if a packet from `destination->source` had a corresponding `source->destination` packet within the polling period and it is sufficiently fresh (no more than 2 seconds old) then it is considered a response to the first packet and the latency is calculated. This approach is naive, and a protocol aware approach might perform better in some cases. For instance, the TCP sequence number could be extracted and used to ensure that packets are considered in the correct sequence order, not simply in the received order.

5.4 eBPF program

A section from an example eBPF TC monitor which could be attached by our userspace control program is shown in Listing 5.1. The entire program is attached in the Appendix Listing A.1. This program collects the packet size of connections captured by the eBPF program and populates it to an eBPF map. A unique connection was defined by the protocol, source address and port, and the destination address and port.

```
1  SEC("classifier")
2  int sniff_action(struct __sk_buff *skb) {
3    __u32 data_len =  skb->data_end - skb->data;
4    void *data_end = (void *)(long) skb->data_end;
5    void *data = (void *)(long) skb->data;
6
7    struct ethhdr *ethh = data;
8    __u64 ethhdr_off = sizeof(*ethh);
9
10   if (data + ethhdr_off > data_end) {
11     return TC_ACT_OK;
12   }
13
14   if (__bpf_ntohs(ethh->h_proto) == ETH_P_IP) {
15     struct iphdr *iph = data + ethhdr_off;
16
17     if (iph + sizeof(*iph) > data_end) {
18       return TC_ACT_OK;
19     }
20     struct cnxn_data msg;
21     memset(&msg, 0, sizeof(struct cnxn_data));
22     msg.ethproto = ethh->h_proto;
23     msg.ipproto = iph->protocol;
24     msg.size = data_len;
25     msg.saddr = iph->saddr;
26     msg.daddr = iph->daddr;
27
28     if (iph->protocol == IPPROTO_TCP) {
```

```
29        struct tcphdr *tcph = data + ethhdr_off + iphdr_off;
30        if (tcph + sizeof(*tcph) > data_end) {
31          return TC_ACT_OK;
32        }
33        msg.sport = tcph->source;
34        msg.dport = tcph->dest;
35      } else if (iph->protocol == IPPROTO_UDP) {
36        struct udphdr *udph = data + ethhdr_off + iphdr_off;
37        if (udph + sizeof(*udph) > data_end) {
38          return TC_ACT_OK;
39        }
40        msg.sport = udph->source;
41        msg.dport = udph->dest;
42      } else {
43        msg.sport = 0;
44        msg.dport = 0;
45      }
46      bpf_map_push_elem(&queue, &msg, BPF_EXIST);
47    }
48    else {
49      bpf_printk("IPv6 Packet");
50    }
51
52    return TC_ACT_OK;
53  }
```

Listing 5.1: Subsection of a TC direct-action packet monitor.

In this example only IPv4 packets are processed after capture. From the IPv4 packet, the protocol is identified and if the protocol is TCP or UDP the source and destination ports are extracted. The majority of this eBPF program consists simply of populating a structure with the desired information from the network packet and passing it to our userspace control program using an eBPF map.

There are several features of interacting with eBPF maps using structures which are important to note. When a structure is passed to userspace, `memset` must be used in order to pass validation. This prevents eBPF programs from leaking

uninitialized memory on the stack to the userspace control program. Structures passed to maps should include the compiler attribute `packed`. This ensures that the compiler consistently aligns the fields of the data structure so that the userspace control program can parse the raw bytes from the map. To optimize memory accesses an additional padding field should be manually included to properly align the data structure to four-byte memory accesses.

The example monitor also includes several checks to ensure that pointer arithmetic does not result in out-of-bounds memory accesses. This is required for the eBPF verifier to ensure memory safety as memory safety is validated through static analysis rather than runtime checks.

This program could be extended to extract additional information from the captured packet. A copy of the `__sk_buff` data structure showing fields accessible within TC direct-action programs is available in Appendix Listing A.2.

If the underlying data is plaintext, deep packet inspection could also be performed within the eBPF program. By performing this operation in the kernel the performance penalty of copying the entirety of the packet to userspace memory is avoided and only the relevant extracted information, if any, is passed to userspace. Potential approaches to monitoring encrypted workloads and capturing plaintext to perform deep packet inspection is discussed in section 6.1.

The queue eBPF map type was chosen to avoid potential race conditions and concurrency bugs. The userspace monitor loads multiple eBPF programs which act as producers to the same eBPF map. In the case of aggregating on the kernel eBPF side if two eBPF programs were to capture packets with the same primary keys (which were considered the protocols, and source and destination addresses/ports) there is a risk of a race condition. If one program reads the count while another program is in the process of incrementing, there could be undercounting. This approach does incur a performance penalty as additional information is passed to userspace that could be avoided by aggregating in the eBPF program. Additionally, as eBPF maps are fixed size and cannot grow once created, on large nodes there is the possibility of overwhelming the consumer and losing packets, which should be considered for this

approach.

An alternative approach would use a hash table eBPF map. The keys would be a data structure containing the primary keys as defined above and the values would be the cumulative packet count and size captured. This would be more memory efficient as the large key value would only need to be added to the eBPF map once per polling cycle. There are multiple ways this approach could be made thread safe. Each eBPF program could be made to only modify its own entries. This could be accomplished by allocating one eBPF map per instance of the eBPF program loaded in the kernel. The userspace control program would then aggregate from multiple maps. Alternatively, each eBPF program could be given a unique identifier which would be included within the primary key data structure so that multiple programs could share the same map. The most memory conscious approach would be to utilize eBPF spinlocks to modify the eBPF map in a thread safe manner. One significant disadvantage of this approach is that eBPF spinlocks were introduced in a recent kernel version which would limit the supported kernel versions or would require implementing a custom spinlock for older kernel versions. Due to the increased complexity of these approaches the prototype utilized queue eBPF maps.

5.5 Integration with external monitoring

As the monitor on each node only has connection data for traffic captured on that pod external databases or monitoring systems should be employed to effectively interact with the captured data. As the collected data is a time series and the monitor is long-lived, popular pull-based aggregation systems such as Prometheus [17] can be used. Traditional push-based databases can also be easily integrated. The approach described in section 5.3 is to push data at the end of every polling cycle. If more frequent reporting is desired the polling time could be decreased with little overhead. To increase the reporting frequency, data should be pushed on multiples of the polling cycle as opposed to increasing the polling time. This is to avoid filling the memory of the fixed size eBPF map and losing data on captured traffic.

Figure 5-4: Cloud Insights display consisting of a table of the total bytes and packet counts of captured traffic, and a time series of the average number of packets captured by source and destination pairs. Data collected from a mostly idle Kubernetes cluster.

The proof of concept was configured to push data to the REST API of the NetApp Cloud Insights tool [33]. Cloud Insights can be used to create visualizations and alerting for the collected network traffic. Figures 5-4 and 5-5 show a Cloud Insights display of networking metrics collected by the proof of concept.

5.6 Packaging and deployment

The final component of the monitoring solution was packaging it to be deployed on Kubernetes clusters. Our system, when compiled, produces a single statically linked binary which was containerized in an empty container. To collect data from every pod on the cluster it requires running the eBPF system on every node in the cluster. The system can then attach the eBPF program to the network interface of every pod running on that node. This is easily accomplished using DaemonSets as shown in figure 5-6.

As the monitor is loading eBPF code into the kernel and accessing the host network

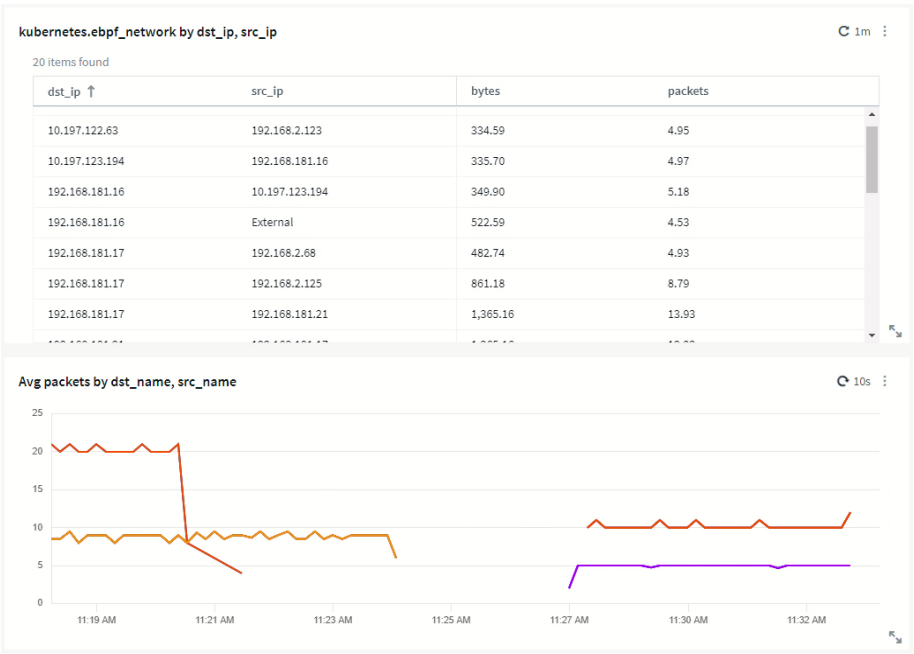

Figure 5-5: A view of filtering the time series in 5-4 for a targeted simulated load which was killed and restarted on new pods during the observation period.

namespace it needs the following permissions.

1. The pod must be run as root or have the `CAP_BPF` capability.

2. To access the veth pairs of other pods, the monitor pod must also be run in the host network namespace using the `hostNetwork:true` specification.

In this approach there is a clear separation of concerns between the actual eBPF packet collection system and any aggregating, monitoring and alerting systems which could be built on top of the system. As discussed, the prototype was integrated with a cloud infrastructure monitoring tool. This is not a necessary component of the design, and it could be quickly configured to push to various other existing monitoring tools or databases. This is an important distinction between this solution and many other commercially available eBPF based Kubernetes monitoring solutions. This prototype is self-contained and does not require installing systems which also collect additional unwanted or redundant data and can be used directly with existing data collection systems.

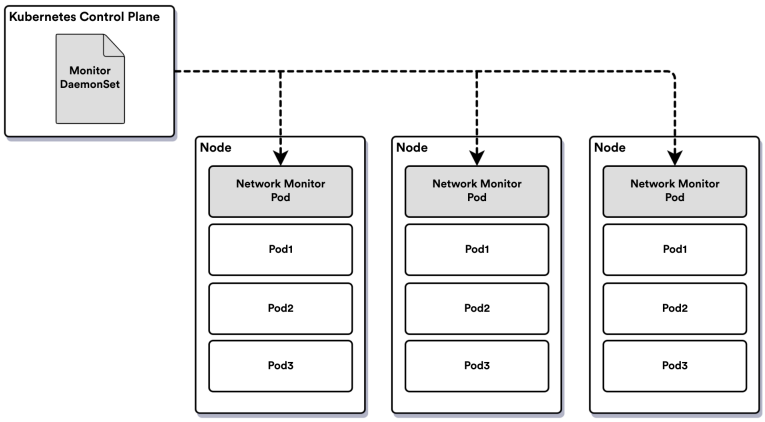

Figure 5-6: The Kubernetes control plane scheduling a monitor pod on all nodes as specified by the DaemonSet.

This proof of concept was evaluated on multiple clusters with different kernel versions and CNI plugins. Calico (with overlay networking), the Amazon VPC CNI, and OpenShift clusters were tested with unmodified copies of the prototype binary. The prototype performed as expected without any cluster specific changes or configuration required, showing portability between cluster configurations. The choice of a stable hook point ensured portability between supported kernel versions. While TC direct-action was introduced in kernel version 4.4 the minimum supported version of the prototype was higher at 4.20 due to the use of the queue eBPF map type. This was adequate for our use case but if support for less recent kernel versions is required hash eBPF maps could be used with the strategies discussed in section 5.4. This prototype demonstrates that our approach can be used to create a drop-in data collector which can support many different Kubernetes cluster configurations and existing monitoring solutions in a lightweight and targeted manner.

Chapter 6

Conclusion

The target of this work was to develop a monitoring solution for collecting the quantity and size of packets between pods in Kubernetes clusters capable of being deployed on existing clusters and integrating with our existing monitoring platform. An evaluation of existing Kubernetes monitoring approaches was performed. Existing solutions were not suitable for this application due to requiring significant amounts of manual instrumentation, performance overheads, being non-targeted or otherwise difficult to integrate with. In this review the extended Berkeley Packet Filter was identified as a technology suitable for developing such a modular high-performance network monitor.

A methodology for creating eBPF network monitors was developed with a specific focus on enabling program portability across kernel versions. A review of eBPF toolkits and loaders was performed for these goals. The BPF Compiler Collection and higher-level Kubernetes BCC eBPF deployment tools were deemed unsuitable for this application due to large memory overheads and unmet build requirements for JIT compilation of eBPF programs. The BPF CO-RE approach and supporting loaders such as libbpf and Cilium/ebpf were identified as supporting our application well. This approach creates portable binaries with minimal overhead.

A prototype network monitor was developed using Cilium/ebpf. This monitor attaches to pod veth network interfaces using the Traffic Control subsystem direct-action hook. This choice of hook points enabled eBPF portability with a low minimum supported kernel version. Attaching to pod veth network interface pairs supports many

common CNI plugins. This approach collects traffic before any network encapsulation or packet mangling is performed for most CNI plugins. Collected packets can be easily correlated to Kubernetes resources using the captured cluster IP address and port number.

This approach is suitably flexible to allow for push or pull operations from instances of the monitor. The prototype monitor was integrated with an existing Kubernetes monitoring system by periodically pushing collected network metadata to the systems's REST API.

6.1 Future work

This work and the implementation process of the proof of concept network monitor has revealed several potential extensions.

The structure of the eBPF system enables access to packet payloads in addition to packet headers. For unencrypted workloads, the prototype could be extended to perform deep packet inspection. This would allow for the creation of application layer metrics. As discussed previously, performing this operation in-kernel requires minimal resource utilization. One significant limitation is that deep packet inspection cannot be performed on encrypted packets. There are several potential methods for collecting unencrypted pod communications in clusters with in-transit data encryption. One approach would be to capture packets at the socket layer as opposed to the TC subsystem. Sidecar based service meshes capture and encrypt packets at a later point in the network stack so network data at the socket level would not yet be encrypted and could be inspected.

An alternative approach could utilize eBPF uprobes to instrument userspace programs and shared libraries. Common encryption libraries such as OpenSSL could be instrumented. A uprobe could be attached to encryption functions to capture plaintext as it is passed to be encrypted and uretprobes could be attached to functions which decrypt ciphertext in order to capture the plaintext as it is passed back to the calling process. An example of this approach can be seen in the BCC tool sslsniff

[19]. This has the advantage of also supporting workloads which perform encryption at the application level as opposed to service meshes which perform encryption at the infrastructure level. The primary complication of this approach in the Kubernetes environment is that each container contains its own copy of any encryption libraries so one cannot instrument a single shared library. This can be worked around by instrumenting every copy of the target library in running containers filesystems on the system. Some existing Kubernetes eBPF based monitoring solutions such as Pixie appear to use this approach.

The prototype currently collects metadata on most packets produced on target clusters and later drops data it is unable to correlate in the userspace control program. Another potential extension of this work could involve implementing in-kernel filtering of data captured. Whitelists and blacklists could be used to target data aggregation more precisely. Whitelists could allow for a monitor to only collect information related to specifically targeted workloads on a cluster. Blacklists could be utilized to selectively drop unwanted packets, for example, control plane workloads could be blacklisted to only collect information on deployed workloads. This could be implemented by utilizing eBPF maps populated by the userspace monitor and acted upon by the eBPF programs. This filtering would decrease noise in the collected metrics and by performing filtering in the eBPF program it would prevent wasting resources on untargeted traffic at higher levels of the system.

The metrics collected with the prototype could also be used to create tools to analyze cluster behavior. A pod and service topology visualization augmented with metrics such as the number of requests between pods could be created. Such a tool could aid in debugging performance problems by visually signifying components of the system which are under heavy load.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A

Listings

```
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_endian.h>

//To avoid pulling in other headers from system
#define ETH_P_IP  0x0800
#define TC_ACT_OK 0
#ifndef memset
# define memset(dest, chr, n)   __builtin_memset((dest), (chr), (n))
#endif

char _license[] SEC("license") = "GPL";

#define SNIFFTOO_MAP_SIZE 1024

struct cnxn_data {
  __u8 ethproto;
  __u8 ipproto;
  __be32 saddr;
  __be16 sport;
  __be32 daddr;
  __be16 dport;
  __u32 size;
  __u64 tstamp;
```

```
25    __u16 padding;
26 }__attribute__((packed));
27
28 struct bpf_map_def SEC("maps") queue = {
29    .type = BPF_MAP_TYPE_QUEUE,
30    .key_size = 0,
31    .value_size = sizeof(struct cnxn_data),
32    .max_entries = SNIFFTOO_MAP_SIZE,
33 };
34
35
36 SEC("classifier")
37 int sniff_action(struct __sk_buff *skb) {
38    __u32 data_len =  skb->data_end - skb->data;
39    void *data_end = (void *)(long) skb->data_end;
40    void *data = (void *)(long) skb->data;
41
42    struct ethhdr *ethh = data;
43    __u64 ethhdr_off = sizeof(*ethh);
44
45    if (data + ethhdr_off > data_end) {
46      return TC_ACT_OK;
47    }
48
49    if (__bpf_ntohs(ethh->h_proto) == ETH_P_IP) {
50      struct iphdr *iph = data + ethhdr_off;
51      __u64 iphdr_off = sizeof(*iph);
52
53      if (data + ethhdr_off + iphdr_off > data_end) {
54        return TC_ACT_OK;
55      }
56      struct cnxn_data msg;
57      memset(&msg, 0, sizeof(struct cnxn_data));
58      msg.ethproto = ethh->h_proto;
59      msg.ipproto = iph->protocol;
60      msg.size = data_len;
```

```
61      msg.saddr = iph->saddr;
62      msg.daddr = iph->daddr;
63      msg.tstamp = bpf_ktime_get_ns();
64
65      if (iph->protocol == IPPROTO_TCP) {
66        struct tcphdr *tcph = data + ethhdr_off + iphdr_off;
67        if (data + ethhdr_off + iphdr_off + sizeof(*tcph) > data_end)
      {
68          return TC_ACT_OK;
69        }
70        msg.sport = tcph->source;
71        msg.dport = tcph->dest;
72      } else if (iph->protocol == IPPROTO_UDP) {
73        struct udphdr *udph = data + ethhdr_off + iphdr_off;
74        if (data + ethhdr_off + iphdr_off + sizeof(*udph) > data_end)
      {
75          return TC_ACT_OK;
76        }
77        msg.sport = udph->source;
78        msg.dport = udph->dest;
79      } else {
80        msg.sport = 0;
81        msg.dport = 0;
82      }
83      bpf_map_push_elem(&queue, &msg, BPF_EXIST);
84    }
85    else {
86      bpf_printk("IPv6 Packet");
87    }
88    return TC_ACT_OK;
89  }
```

Listing A.1: Example TC direct-action packet monitor.

```
1  struct __sk_buff {
2      __u32 len;
3      __u32 pkt_type;
```

```
    __u32 mark;
    __u32 queue_mapping;
    __u32 protocol;
    __u32 vlan_present;
    __u32 vlan_tci;
    __u32 vlan_proto;
    __u32 priority;
    __u32 ingress_ifindex;
    __u32 ifindex;
    __u32 tc_index;
    __u32 cb[5];
    __u32 hash;
    __u32 tc_classid;
    __u32 data;
    __u32 data_end;
    __u32 napi_id;

    /* Accessed by BPF_PROG_TYPE_sk_skb types from here to ... */
    __u32 family;
    __u32 remote_ip4;    /* Stored in network byte order */
    __u32 local_ip4;    /* Stored in network byte order */
    __u32 remote_ip6[4];    /* Stored in network byte order */
    __u32 local_ip6[4];    /* Stored in network byte order */
    __u32 remote_port;    /* Stored in network byte order */
    __u32 local_port;    /* stored in host byte order */
    /* ... here. */

    __u32 data_meta;
    __bpf_md_ptr(struct bpf_flow_keys *, flow_keys);
    __u64 tstamp;
    __u32 wire_len;
    __u32 gso_segs;
    __bpf_md_ptr(struct bpf_sock *, sk);
    __u32 gso_size;
```

```
38 };
```

Listing A.2: `__sk_buff` data structure.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Bpf compiler collection (bcc) repository. https://github.com/iovisor/bcc. Last commit: 7c02284.

[2] Cilium ebpf repository. https://github.com/cilium/ebpf.

[3] Cilium project repository. https://github.com/cilium/cilium. Accessed: 2022-04-01.

[4] Flannel repository. https://github.com/flannel-io/flannel.

[5] Kubectl-trace repository. https://github.com/iovisor/kubectl-trace.

[6] Libbpf repository. https://github.com/libbpf/libbpf.

[7] netlink - netlink library for go repository. https://github.com/vishvananda/netlink.

[8] Cilium Authors. Hubble: Network, service & security observability for kubernetes. https://github.com/cilium/hubble.

[9] Cilium Authors. Try ebpf-powered cilium service mesh - join the beta program! https://cilium.io/blog/2021/12/01/cilium-service-mesh-beta, December 2021.

[10] CNI Authors. Cni: Extension conventions. https://github.com/containernetworking/cni/blob/main/CONVENTIONS.md. Latest commit: 34a8a46.

[11] CNI Authors. Container network interface (cni) specification. https://github.com/containernetworking/cni/blob/main/SPEC.md. Latest commit: 269bf61.

[12] Istio Authors. The istio service mesh. https://istio.io/latest/about/service-mesh/. Accessed: 2022-04-01.

[13] Istio Authors. Observability: Distributed traces. https://istio.io/latest/docs/concepts/observability/#distributed-traces. Accessed: 2022-04-01.

[14] Jaeger Authors. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/. Accessed: 2022-04-01.

[15] Linkerd Authors. Linkerd: A different kind of service mesh. https://linkerd.io/. Accessed: 2022-04-01.

[16] OpenTelemetry Authors. Opentelemetry homepage. https://opentelemetry.io/. Accessed: 2022-04-01.

[17] Prometheus Authors. Prometheus homepage. https://prometheus.io/. Accessed: 2022-04-14.

[18] BCC Contributors. Bpf features by linux kernel version. https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md. Accessed: 2022-04-25.

[19] BCC Contributors. sslsniff.py. https://github.com/iovisor/bcc/blob/master/tools/sslsniff.py.

[20] Jonathan Corbet. Reconsidering unprivileged bpf. https://lwn.net/Articles/796328/, August 2019.

[21] Kernel development community. Bpf documentation. https://www.kernel.org/doc/html/latest/bpf/index.html. Accessed: 2022-04-01.

[22] Kernel development community. Classic bpf vs ebpf. https://www.kernel.org/doc/html/latest/bpf/classic_vs_extended.html. Accessed: 2022-04-01.

[23] eBPF Foundation. ebpf documentation. https://ebpf.io/what-is-ebpf. Accessed: 2022-04-01.

[24] Matt Fleming. A thorough introduction to ebpf. https://www.kernel.org/doc/html/latest/bpf/classic_vs_extended.html, December 2017.

[25] Cloud Native Computing Foundation. Kubernetes homepage. https://kubernetes.io/. Accessed: 2022-04-01.

[26] The Apache Software Foundation. Apache log4j security vulnerabilities. https://logging.apache.org/log4j/2.x/security.html, February 2022.

[27] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. PLDI 2019, page 1069–1084, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Kinvolk GmbH. Inspektor gadget. https://kinvolk.io/docs/inspektor-gadget/latest.

[29] Thomas Graf. ebpf - the future of networking & security. https://cilium.io/blog/2020/11/10/ebpf-future-of-networking/, November 2020.

[30] Thomas Graf. How ebpf will solve service mesh - goodbye sidecars. https://isovalent.com/blog/post/2021-12-08-ebpf-servicemesh, December 2021.

[31] Amazon Web Services Inc. Pod networking (cni). https://docs.aws.amazon.com/eks/latest/userguide/pod-networking.html. Accessed: 2022-04-01.

[32] Docker Inc. Docker: Manage swarm service networks. https://docs.docker.com/engine/swarm/networking/. Accessed: 2022-04-01.

[33] NetApp Inc. Cloud insights. https://cloud.netapp.com/cloud-insights/. Accessed: 2022-04-30.

[34] New Relic Inc. Pixie labs homepage. https://pixielabs.ai/.

[35] Tigera Inc. About calico. https://projectcalico.docs.tigera.io/about/about-calico. Accessed: 2022-04-01.

[36] The kernel development community. Bpf type format (btf). https://www.kernel.org/doc/html/latest/bpf/btf.html.

[37] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*, San Diego, CA, January 1993. USENIX Association.

[38] Quentin Monnet. ebpf updates #4: In-memory loads detection, debugging quic, local ci runs, mtu checks, but no pancakes. https://ebpf.io/blog/ebpf-updates-2021-02, February 2021.

[39] Andrii Nakryiko. Bpf co-re reference guide. https://nakryiko.com/posts/bpf-core-reference-guide/, October 2021.

[40] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, 182:111061, 2021.

[41] Weaveworks. Weave scope. https://www.weave.works/oss/scope/.

[42] Andrew Wei, Leiyi Zhang, and Zhao Xinghan. Service mesh overhead measurement in kubernetes container networks. https://nowei.github.io/projects/svc_mesh_measurement_final_report.pdf. Accessed: 2022-04-01.

[43] Wenbo Zhang. Why we switched from bcc to libbpf for linux bpf performance analysis. https://pingcap.com/blog/why-we-switched-from-bcc-to-libbpf-for-linux-bpf-performance-analysis, December 2020.