

Leveraging eBPF for Enhanced Kubernetes Observability and Security

MSc Research Project
Cloud Computing

Pham Ngoc Thanh Hung
Student ID: 22232338

School of Computing
National College of Ireland

Supervisor: Sudarshan Deshmukh

National College of Ireland
MSc Project Submission Sheet
School of Computing



Student Name: Pham Ngoc Thanh Hung
Student ID: 22232338
Programme: Cloud Computing **Year:** 2023
Module: Research Project
Supervisor: Sudarshan Deshmukh
Submission Due Date: 16th September 2024
Project Title: Leveraging eBPF for Enhanced Kubernetes Observability and Security
Word Count: 5669 **Page Count:** 22

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Pham Ngoc Thanh Hung

Date: 16th September 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	<input type="checkbox"/>
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	<input type="checkbox"/>
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Leveraging eBPF for Enhanced Kubernetes Observability and Security

Pham Ngoc Thanh Hung

22232338

Abstract

With the development of cloud computing, Kubernetes is an orchestration tool used in a modern microservice application. However, the architectural complexities and the shared nature of the kernel have caused serious security and observability challenges. This paper explores how eBPF can help in harvesting detailed insights into system behavior, application logs, and real-time security threats for greater observability and security in Kubernetes. In this research, eBPF will be considered in its implementation for security breach detection at runtime with the least performance overheads. Moreover, the integration of tools such as Falco, which are based on eBPF with already established monitoring systems like Prometheus and Grafana, and AlertManager which triggers any important alerts to administrators, giving high visibility and helping in the real-time mitigation of security threats from within the kernel.

Keywords: Observability, Kubernetes, EKS, eBPF, Falco, Run-time Security Detection, Prometheus, Grafana, AlertManager

1 Introduction

Containerization has brought a revolution to application development, deployment, and management. Leading the rank of revolutionary tools is Kubernetes, a container orchestration platform and probably the most mature technology within the Cloud Native Computing Foundation. Even though it is so popular in the market as the best tool, Kubernetes also comes with the large number of security issues including multi-layer networking, shared underlying hardware and clustering components. These vulnerabilities can be exploited without the administrator's awareness due to many attack surfaces. Therefore, there is a strong need for monitoring activities in Kubernetes clusters and gaining insight into the system behavior and application interaction of the clusters toward potential security threat identification.

Traditional monitoring tools, however, are not able to capture the required granularity of operation for effective security analysis. The limitations of these tools are adding unacceptable overheads or complexity for deeper observability. eBPF emerges as one of the very promising solutions in this situation. This modern Linux kernel technology offers a way for users to write and deploy custom programs directly into the kernel space. Running these programs in kernel helps to achieve even more detailed data collection on various system events and metrics. This technology extends to modifying system behavior for enhanced security measures, such as policy enforcement and anomaly detection.

The research in this work uses eBPF technology to address both observability and security challenges in Kubernetes environments. I try to answer the question: how can eBPF be leveraged to enhance observability and security in Kubernetes, focusing on real-time detection and security policy violation alerts? This paper makes an attempt at adopting eBPF in Kubernetes and the integration of current security tools such as Falco and powerful monitoring tools including Prometheus, AlertManager and Grafana. A quantitative measurement of eBPF effectiveness is the metrics used to measure the number of detected security incidents. This study will contribute to the understanding of Kubernetes security through the integrated deployment of eBPF-based solutions for an organization relying on containerized environments.

2 Related Work

Extended Berkeley Packet Filter (eBPF) has become an essential technology for observability and security in the fast-changing landscapes of cloud-native environments. This section critically reviews the contributions of some research in this area to fill a research gap, from network observability to security enhancements, performance tuning, and network policies.

2.1 eBPF Design

In the eBPF design, it manages the Just-In-Time (JIT) compilation for efficient performance, stateful processing through maps, and the use of kernel libraries to be able to perform complex operations [5]. It should be noted here that the described feature enables user space applications to inject the code related to the kernel at runtime, without a need to recompile it or add additional modules. It is possible to write eBPF programs in C or eBPF assembly, cross-compile to bytecode, and after making all possible verifications to ensure the safety of the code and its proper execution, load the bytecode into the kernel by a `bpf()` system call. The eBPF programs are event-driven, which means they are invoked by specific events. They contain the ability to modify the context associated with that event. All of these make maps a vital data structure in eBPF, used for sharing information between program runs, between different programs, and also between programs and user space. This new development in kernel technology is enabling the building of sophisticated networked applications. This provides a flexible, powerful, and secure mechanism for running code at runtime within the Linux Kernel. Therefore, it bridges the gap between practical eBPF implementations for building complex network services and network policy evaluation, making it possible for future research and development in eBPF-based network solutions. However, the steep learning curve associated with eBPF can deter its adoption. However, the steep learning curve associated with eBPF can deter its adoption.

2.2 Observability with eBPF and Kubernetes

The investigation was pioneered by Liu et al. [1] using a protocol-independent approach for observability study of container networks that uses eBPF inside Kubernetes clusters. Three important ideas result from the distributed system's observability: distributed tracing, metrics, and logging. The use of eBPF tools is very instrumental in fine-grain monitoring of all the container network metrics. Being non-intrusive, eBPF scales transparently with low overhead from monitoring the kernel and user applications through dynamic updates to eBPF programs

of the respective targeted events being monitored without any modifications to the kernel or application code. This is helpful in precise performance analysis through the correlation of event-related contexts with some Pod instances based on detailed data obtained from kernel operations. While eBPF does improve scalability and offer a solid framework for static and dynamic tracing in the Linux kernel, its real power in cloud platforms like Kubernetes seems to be hindered due to the actual lack of native support and comprehensive monitoring solutions for the container network. Although they do not provide strong support for Kubernetes setups and have limited programmability and extensibility, current tools like Dtrace and Sysdig satisfy the granularity requirements for data at the kernel level. However, in the container environment, fine-grained observability data would pose a significant barrier to conventional monitoring tools, creating a gap that is being filled by eBPF-based solutions and the likes of new MicroRCA tools. This study sheds light on how eBPF may be predicated on future security research and deliver fine-grained observability in cloud-native contexts.

In hybrid systems, there is a high possibility that processes consume an extremely high volume of system resources, enough to lead to a crash of the process. An anomaly detection model in CPU, memory, and I/O resource consumption processes is proposed using eBPF technology through the Isolation Forest algorithm. eBPF enables fine-grained and real-time data extraction directly from the kernel than traditional tooling allows, thus leading to higher accuracy. Afterwards, the collected data are processed and analyzed using the Isolation Forest algorithm, which is very useful in the detection of anomalies in data spaces of very high dimensions. For optimal precision and recall, some key parameters in the algorithm are fine-tuned. Experimental results have shown this approach to be effective in detecting anomalies of process resource usage with high accuracy and reliability in the obtained model [6]. The eBPF monitoring solution enables fine-grained monitoring with granularity and accurate anomaly detection, improving system reliability in the detection and mitigation of resource-related anomalies.

2.3 Security Enhancements using eBPF

Instead of focusing on network services, Sadiq et al. [2] present a novel method for detecting DoS attacks in Kubernetes-based cloud environments by utilizing eBPF. Before the Berkeley Packet Filter (BPF) was introduced in the early 1990s, packet monitoring and analysis depended on a traditional filtering method. There was a significant packet processing latency since every packet was copied from kernel space to user space via the kernel. Using BPF, Steven McCanne and Van Jacobson refined this technique by copying packets only when necessary to reduce overhead. BPF was designed for 32-bit machines, so it had a fixed-length instruction set. BPF was first used in the implementation within the Unix operating system. It has grown with its application over time to different operating systems, including Linux and Windows. Now it has become an Extended Berkeley Packet Filter (eBPF), which allows the safe and efficient running of code within the Linux kernel. Initially, the memory utilization in eBPF followed a different, conventional method that made extensive use of the memory. The excellent outcome, from 15% to only 5%, indicates the eBPF performance impact optimizations. Real-time DoS attack detection in containerization environments is greatly advanced by this study.

The increasing number of IoT devices has brought significant security challenges recently, especially many devices being hijacked for botnets to execute DDoS attacks. Feraudo et al. [7] integrate the Manufacturer Usage Description (MUD) standard with eBPF-based traffic filtering to mitigate such threats. The MUD standard enables manufacturers of devices to clearly define which communication patterns are allowed for a device, enforced at the gateway level. This research proposes extensions to MUD that incorporate fine-grained rate-limiting capabilities and develops two enforcement backends, one with eBPF and one with iptables. The experiments show that these methods efficiently protect against attacks with minimal impact on legitimate traffic. Moreover, the use of eBPF allows efficient and low-latency processing, so it is fit for a real-time threat-mitigation system. This approach makes security even stronger by disallowing the interception of unauthorized traffic. It is a scalable solution that can be easily deployed into standard home routers, making it applicable in broader network environments.

The exploration of advanced runtime security mechanisms for Kubernetes through the use of eBPF describes significant advancements over traditional Linux Security Modules (LSM) such as AppArmor and SELinux [3]. By employing security policy enforcement, the proposal of KRSIE will provide fine-grained control over the security of the Kubernetes workload while allowing for dynamic adaptation. Unlike the current standard Linux Security Module (LSM) implementation of SELinux and AppArmor, which requires container restarts in order to allow complete policy enforcement during runtime. This suggests that there is an operational problem in the cloud context, where the performance implications associated with container restarts would make it impossible to deploy. Unlike path-based AppArmor and label-based SELinux, KRSIE directly interfaces to LSM function parameters and offers a more tangible method for kernel object management, hence enforcing fine-grained security. In contrast, AppArmor and SELinux are static in nature despite helping more efficiently with the system-wide policy application. They do not have the possibility of being dynamically set and adapted to changes in policies without a restart of a container. Other solutions, such as Cilium, are more concerned with the network security field and may not be as comprehensive in protecting runtime workloads. On their part, Tetragon and KubeArmor are solutions that offer a wider security capability but do not exactly leverage the power of LSM hooks or give the user adequate policy control to enable their use cases without potentially falling short such as protecting from TOCTOU attacks or in accessing kernel contexts like the bprm structure. The proposed solution is designed by policies being authored as a CRD in YAML files, with the policy implementation being done by a manager and enforcer mechanism that results in actual eBPF-LSM programs. The limitations of KRSIE are mainly in the manner in which duplicated LSM security functions are covered over all the potential vulnerabilities, although the approach is innovative.

2.4 Performance with Kubernetes

Performance is a strong exhibit for eBPF within Kubernetes environments, as significant improvements have been shown in networking overhead without reliance on iptables for network configuration and routing. However, one of the main barriers for most practitioners is the unavailability of accessible interfaces for eBPF tools. Additionally, eBPF integration with existing network management tools can complicate deployment in legacy systems.

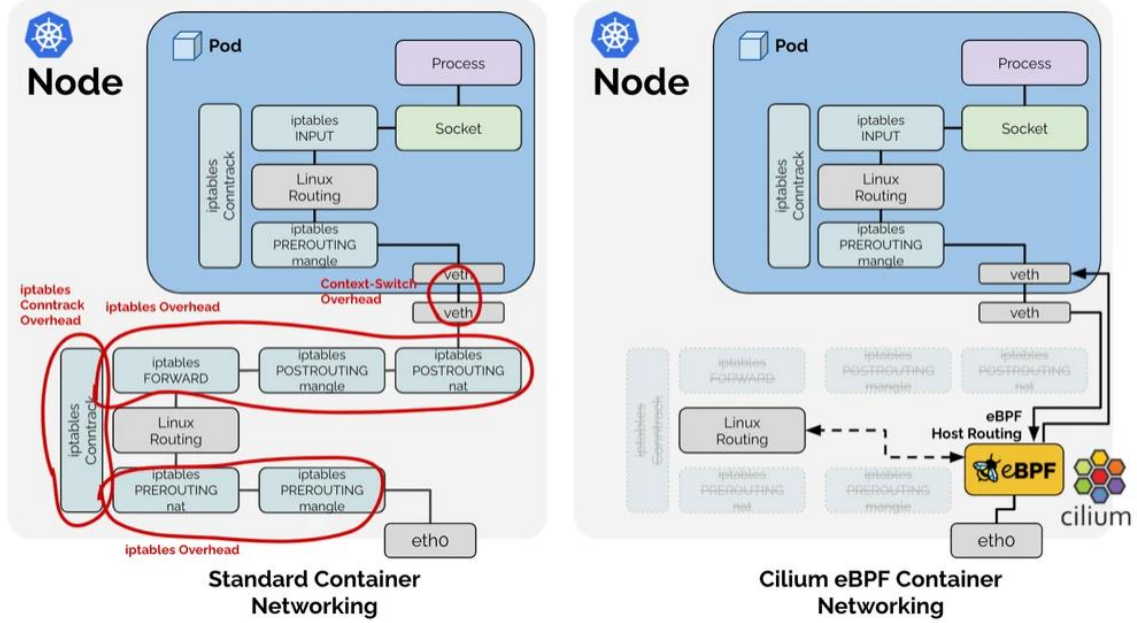


Figure 1: Iptable overhead removal by eBPF [8]

Budigiri et al. [4] further on the topic of Kubernetes network policies. This is a clear indication that there is a need for proper integration of network policies in a Kubernetes environment to enhance low-overhead security during inter-container communication using eBPF instead of iptables. In fact, the different CNI plugins analyzed demonstrate network policies. Moreover, those applied by Calico instead of Cilium hardly bring any performance overhead. Since Calico works with pure BGP-based Layer 3 routing and innovates through XDP for filtering packets effectively in order to reduce Denial of Service attacks. That the differences are more apparent, although Calico leverages eBPF technology to increase its scalability, it does so by using connection tracking for all other packets and inspecting policies only for the first packet in each new flow. In addressing and resolving any threats and vulnerabilities, it is proactive and responsive in line with the recognized attacker model. Therefore, eBPF promotes cutting-edge, low-latency security solutions. The move from network observability to network policy assessment is a significant advancement in containerized environment security, allowing for even more system efficiency without sacrificing security measures.

2.5 Summary of Findings

All of the evidence leads towards the transformational role of eBPF in Kubernetes observability and security as follows:

Fine-Grained Observability: eBPF is great at providing detailed monitoring and recording of container network metrics with a low overhead [1][6]. It provides non-intrusive scalable solutions in this space for performance analysis where traditional tools, such as Dtrace and Sysdig, fail to keep up inside Kubernetes environments [1].

Dynamic Security Enforcement: eBPF can detect DoS attacks in real-time while reducing resource usage [2]. Moreover, eBPF has the capability to scale effectively with large clusters of devices, such as IoT environments [7]. Unlike traditional Linux Security Modules (LSMs)

such as AppArmor and SELinux, eBPF allows for runtime policy adjustments without restarting containers, hence providing more flexible and effective security management [3].

Performance in Kubernetes: eBPF has also been integrated into the Kubernetes environment for performance. It shows an important improvement in accuracy and overhead reduction compared to the traditional tools which have many limitations [4].

Despite these advancements, eBPF-based solutions are not easy to integrate with current network management and monitoring tools, which may make the deployment in legacy systems more complicated. Furthermore, the eBPF steep learning curve is one of the factors that prevent it from broader usage.

My research attempts to bridge this gap by integrating traditional monitoring solutions with eBPF-based tooling to provide an increased level of observability and security in a Kubernetes environment. This integration will also bring together the detailed, low-overhead monitoring capabilities of eBPF with its dynamic security capabilities into workflows of traditional tools. Therefore, I will integrate Falco (eBPF) with well-known traditional monitoring and alert stacks on Kubernetes, including Prometheus, Grafana, and AlertManager. This demonstrates the compatibility and effectiveness of eBPF in monitoring and alerting systems in cloud-native environments, avoiding the risks related to the containerization of applications and deploying them at a cloud scale.

3 Research Methodology

3.1 Problem Definition

The complexity of modern cloud-native environments, specifically Kubernetes, brings different observability and security challenges. As an organization adopts a microservices architecture and containerized applications, monitoring and security become important. Traditional tools applicable to this environment generally do not deliver the needed granular insight that could be of great help to detect potential threats in real time without imposing considerable overhead on performance.

This study addresses the problem: how eBPF (Extended Berkeley Packet Filter) technology should be leveraged to add observability and security to Kubernetes environments. The general objective of this research is, therefore, bridging the gap between high-level observability, already available through such tools as Prometheus and Grafana, and more insightful monitoring down at the kernel level, represented by eBPF. Fundamentally, at the core of the problem is the efficiency in which security incidents can be detected. This research will prove that adding eBPF to your existing observability and security tooling introduces the efficiency of tracking and mitigating threats in real-time, making the entire security posture of Kubernetes deployment more effective.

3.2 Proposed Solution

My proposed solution is the use of eBPF technology for better visibility and security in a Kubernetes cluster. With the included eBPF system, events happening at the kernel level are monitored in real-time such that security breaches could be alerted to and acted upon as they happen. Falco is an open-source project that enables the capture and analysis of system calls and other events in applications, in order to determine the security of runtimes within a cloud-native environment. It is going to be integrated with Prometheus and visualized through Grafana, which will help monitor and take actions against potential security threats in distributed environments. This monitoring stack will watch over a Kubernetes cluster running a production-like e-commerce application built on the microservices architecture [9]. This approach enhances the security postures of Kubernetes clusters by re-enforcing our capacity to monitor, detect, and alert upon security issues.

Falco is used for the log collection and violation detection using eBPF. Falco is a cloud-native runtime security tool, running on Linux operating systems. It looks to identify and alert users of any strange behavior and potential security threats in real time. The Falco rule engine is a kernel monitoring and detection agent that watches system calls. According to customized rules, it can also enrich these events by incorporating metadata from container runtimes and Kubernetes. In addition, off-host systems can further process collected events, such as SIEM or data lakes.

Experiments to validate the proposed solution are carried out by simulated security incidents. The effectiveness of the setup is tested by simulating various attacks using exploited containers on the Kubernetes cluster. These attacks are a deliberate series of harmful actions including log clearance, removal of bulk data from the disk, reading sensitive files, symlinks creation over sensitive files, opening port connections, and crypto mining. The criteria for evaluation include detection accuracy and log details Falco captures and how good it is in compatibility with the stacks of Prometheus in a Kubernetes environment.

4 Design Specification and Implementation

This section describes the architecture, software configuration, hardware setup, and tools used to implement the proposed solution for the extension of Kubernetes security and observability using eBPF in the current work.

4.1 System Architecture

The system architecture is laid for having an integrated solution inside an Amazon EKS cluster with three worker nodes of type t3a.xlarge. This type of instance will give a balanced mix of compute, memory, and network resources necessary for the loads of observability and security management without adding much overhead in performance.

- Kubernetes cluster: Managed by Amazon EKS to provide us with the container orchestration environment.
- Falco: Deployed as a DaemonSet on all the worker nodes. Falco uses eBPF to make real-time monitoring of the system calls, thus enabling the detection of any suspected activity in compliance with its pre-defined rules.

- Prometheus: A monitoring tool used to scrape metrics from Falco, and store these metrics for analysis.
- Grafana: Visualize the collected metrics from Prometheus into graphs with a predefined dashboard.
- FalcoSidekick: A service that forwards Falco events to an endpoint Prometheus can scrape.
- FalcoSidekickUI: A central GUI for specifying event details captured by Falco.

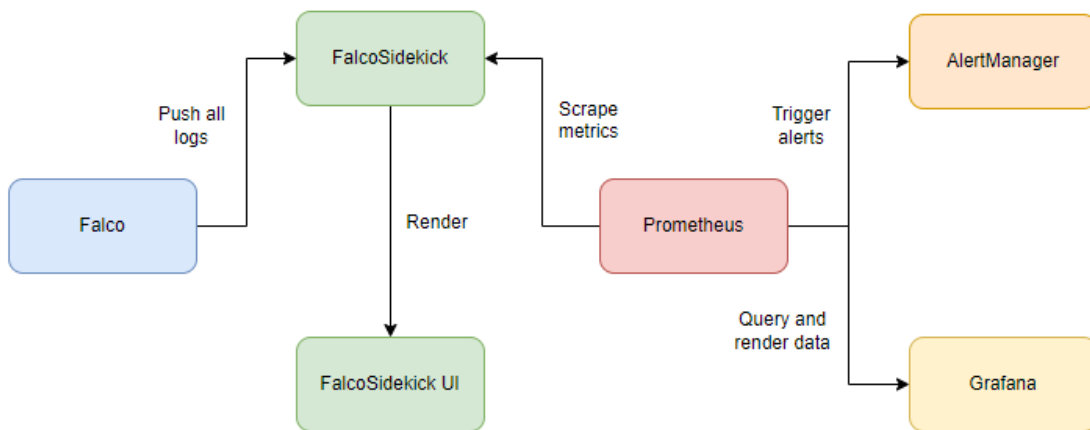


Figure 2: The architecture of the monitoring system

4.2 Software Configuration

Amazon EKS: The Kubernetes control plane is managed by AWS, while the worker nodes are configured with t3a.xlarge instance types. The deployment installs all necessary addons such as kube-proxy, coreDNS and Amazon VPC CNI into the EKS cluster. The cluster runs Kubernetes the latest version 1.30.

Falco: Configured as a DaemonSet to make sure it runs on every worker node. Further customization of the Falco rule engine is made using additional rules to be able to detect certain specific threats. The rule are defined in YAML files and install via Helm

Prometheus: Configured to scrape Falco metrics by using the FalcoSidekick integration. It is deployed using Helm to ensure consistency of deployment across the cluster.

Grafana: Configured to present the metrics being collected by Prometheus. Customized dashboards are developed where the security events or system performance metrics and other related data are displayed in GUI.

4.3 Hardware Configuration

Computation resources: The cluster has three t3a.xlarge instances with 4 vCPUs and 16GB of memory. The balance was right-sized to the needs of processing eBPF operations and Kubernetes workloads.

Storage: Each worker node has a 20 GB gp2 EBS volume which is sufficient for storing system logs and application data.

Networking: The EKS cluster uses public and private subnets. The Amazon VPC CNI, a networking addon, is deployed to enable networking functionality operate properly across pods in the cluster.

4.4 Tools and Technologies

eBPF (Extended Berkeley Packet Filter): eBPF (Extended Berkeley Packet Filter): A core technology that enables effective monitoring of system calls and kernel-level events. eBPF programs are deployed inside the Linux kernel of worker nodes to collect fine-grained metrics without much performance overhead.

Falco: An open-source security tool for runtime security monitoring and detecting abnormal actions at the kernel level, using eBPF. It is set up to alert system administrators about potential security threats in real time.

Prometheus: A time-series database and alerting solution that scrapes metrics from Falco and other sources in order to store them for analysis and alerting.

Grafana: A visualization tool useful for creating dashboards from metrics queried from Prometheus to provide a central GUI. It allows real-time monitoring of the security and performance of the entire system.

Helm: A package manager for Kubernetes, which used to deploy Prometheus, Grafana, Falco, and FalcoSidekick

FalcoSidekick: A service that forwards Falco events to different endpoints in a fan-out way

4.5 Implementation

This section describes detailed implementation process in setting up a Kubernetes cluster on AWS, deploying a microservices applicatio, and integrating Falco with Prometheus and Grafana for a full monitoring stack.

4.5.1 Deploy an EKS cluster

Create an EKS cluster with version 1.30. This cluster allows administrator access with EKS API and ConfigMap.

Kubernetes version

Info

Select Kubernetes version for this cluster.

1.30

Upgrade policy

Info

Choose one of the following options. You can switch the setting later while the standard support period is in effect.

☐ Extended

This option supports the Kubernetes version for 26 months after the release date. The extended support period has an additional hourly cost that begins after the standard support period ends. When extended support ends, your cluster will be auto upgraded to the next version.

☒ Standard

This option supports the Kubernetes version for 14 months after the release date. There is no additional cost. When standard support ends, your cluster will be auto upgraded to the next version.

Cluster access

Info

Control how IAM principals can access this cluster.

Bootstrap cluster administrator access

Info

Choose whether the IAM principal creating the cluster has Kubernetes cluster administrator access.

☒ Allow cluster administrator access

Allow cluster administrator access for your IAM principal.

☐ Disallow cluster administrator access

Disallow cluster administrator access for your IAM principal.

Cluster authentication mode

Info

Configure which source the cluster will use for authenticated IAM principals.

☐ EKS API

The cluster will source authenticated IAM principals only from EKS access entry APIs.

☒ EKS API and ConfigMap

The cluster will source authenticated IAM principals from both EKS access entry APIs and the aws-auth ConfigMap.

☐ ConfigMap


The cluster will source authenticated IAM principals only from the aws-auth ConfigMap.

Figure 3: EKS cluster configuration

All other steps leave as default. Once the cluster is created, a nodegroup is created with three t3a.xlarge nodes to join the cluster as worker nodes.

Context: arn:aws:eks:eu-west-1:256738637992:cluster/x22232338
Cluster: arn:aws:eks:eu-west-1:256738637992:cluster/x22232338
User: arn:aws:eks:eu-west-1:256738637992:cluster/x22232338
K9s Rev: v0.32.5
K8s Rev: v1.30.2-eks-db838b9
CPU: n/a
MEM: n/a

<e> Cordon <u> Uncordon
<ctrl-d> Delete <y> YAML
<d> Describe
<r> Drain
<e> Edit
<?> Help



NAME	STATUS	ROLE	TAINTS	VERSION	PODS	AGE
ip-172-31-14-11.eu-west-1.compute.internal	Ready	<none>	0	v1.30.0-eks-036c24b	9	168m
ip-172-31-21-203.eu-west-1.compute.internal	Ready	<none>	0	v1.30.0-eks-036c24b	21	168m
ip-172-31-34-27.eu-west-1.compute.internal	Ready	<none>	0	v1.30.0-eks-036c24b	9	168m

Figure 4: EKS cluster with three worker nodes

Install k9s to access the cluster through simple terminal UI:

```
curl -sS https://webinstall.dev/k9s | bash
```

Then, we download the kubeconfig file of the EKS cluster :

```
aws eks --region eu-west-1 update-kubeconfig --name x22232338
```

Now, we can access the cluster through k9s tool

4.5.2 Install the microservices application

To simulate a production environment, a demo microservices application is installed on the cluster. For this project, the Google Cloud Platform's Microservices Demo is used:

```
git clone https://github.com/GoogleCloudPlatform/microservices-demo.git
```

Apply all the Kubernetes manifest to deploy the entire microservices application:

```
kubectl apply -f ./release/kubernetes-manifests.yaml
```

This application includes

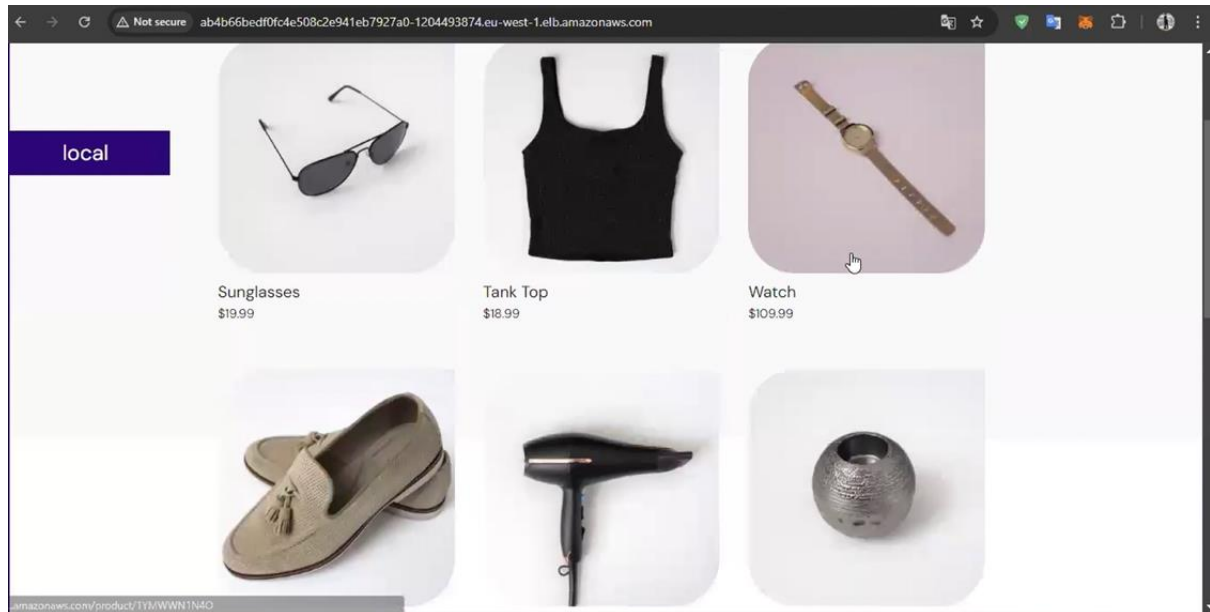


Figure 5: E-commerce application

4.5.3 Install and Configure Falco

Falco is deployed as a run-time security tool for the Kubernetes cluster, utilizing eBPF technology. A custom rule is also configured to detect crypto mining behaviors. New rules can be added to `/etc/falco/falco_rules.local.yaml` in Falco pods. FalcoSidekick is deployed for exporting log events for Prometheus and UI for more details.

```
helm install falco -f custom-rules.yaml ./ -n falco
```

```
# -- For configuration values, see https://github.com/falcosecurity/chart
falcosidekick/values.yaml
falcosidekick:
  # -- Enable falcosidekick deployment.
  enabled: true
  # -- Enable usage of full FQDN of falcosidekick service (useful when a
  fullfqdn: false
  # -- Listen port. Default value: 2801
  listenPort: ""
  webui:
    enabled: true
    redis:
      storageEnabled: false
```

Figure 6: FalcoSidekick and UI deployment

```

1  customRules:
2    custom-busybox-rule.yaml: |-
3      - rule: Terminal busybox instance in container
4        condition: >
5          and not user_expected_terminal_shell_in_container_conditions
6          and user_expected_terminal_shell_in_container_conditions
7        output: >
8          A BUSYBOX instance was spawned in a container with an attached terminal (user=%user.name
9            user_loginuid=%user.loginuid %container.info
10            shell=%proc.name parent=%proc.pname cmdline=%proc.cmdline pid=%proc.pid terminal=%proc.tty
11            container_id=%container.id image=%container.image.repository)
12        priority: WARNING
13        tags: [container, shell, mitre_execution, T1059]
14  detect_crypto_miner_rule.yaml: |-
15    - list: miner_ports
16      items: [
17        25, 3333, 3334, 3335, 3336, 3357, 4444,
18        5555, 5556, 5588, 5730, 6099, 6666, 7777,
19        7778, 8000, 8001, 8008, 8080, 8118, 8333,
20        8888, 8899, 9332, 9999, 14433, 14444,
21        45560, 45700
22      ]
23
24
25

```

Figure 7: Falco custom rules

4.5.4 Install and configure Prometheus, Grafana and AlertManager

Prometheus and Grafana are deployed using Helm for capturing and visualizing metrics from Falco. Prometheus must be configured to collect Falco metrics. This is done by modifying the Prometheus scrape_configs in the Helm chart file.

```

3812  additionalScrapeConfigs:
3813    - job_name: 'falco'
3814      scrape_interval: 30s
3815      scrape_timeout: 10s
3816      metrics_path: /metrics
3817      scheme: http
3818      static_configs:
3819        - targets: ['falco-falcosidekick.falco:2801']
3820      # - job_name: kube-etcd

```

Figure 8: Prometheus configuration to scrape metrics from the FalcoSidekick endpoint

Next, Prometheus rules are added to trigger alerts to AlertManager. Alerts are triggered only warning log level or higher.

kubectl apply -f prometheus-alert-rule.yaml -n monitoring

```

! prometheus-alert-rule.yaml X
! prometheus-alert-rule.yaml
1  apiVersion: monitoring.coreos.com/v1
2  kind: PrometheusRule
3  metadata:
4    name: falco-rules
5    namespace: monitoring
6    labels:
7      release: prometheus
8  spec:
9    groups:
10   - name: Falco Group
11     rules:
12     - alert: FalcoRuleTriggered
13       expr: sum by (k8s_ns_name, rule, priority) (increase(falco_events{priority=~"Warning|Critical"}[30m])) > 0
14       # for: 1m
15       labels:
16         severity: warning
17         source: falco
18       annotations:
19         summary: Falco Rule Triggered
20         description: "Falco Rule Triggered: {{ $labels.rule }} in namespace {{ $labels.k8s_ns_name }}. ({{ $value }} occurrences"
21

```

Figure 9: Alert rule in Prometheus

Then, AlertManager is configured to send notification to a Telegram channel. The alerts will be sent within 1 minute to reach administrators.

```

264  alertmanager:
303  config:
306    inhibit_rules:
321    - source_matchers:
325      equal:
327      - target_matchers:
328        - 'alertname = InfoInhibitor'
329    route:
330      group_by: ['alertname', 'k8s_ns_name', 'rule']
331      group_wait: 30s
332      group_interval: 5m
333      repeat_interval: 12h
334      receiver: 'telegram'
335    receivers:
336    - name: 'telegram'
337      telegram_configs:
338      - api_url: 'https://api.telegram.org'
339        bot_token: [REDACTED]
340        chat_id: [REDACTED]
341    templates:
342    - '/etc/alertmanager/config/*.tmpl'

```

Figure 10: AlertManager configuration to send alerts to the Telegram channel

4.5.5 Custom Grafana UI

Once Grafana is up and running, a custom dashboard is created to show all security breaches captured by Falco in the EKS cluster including total number of security events detected, types of events and a graph in a period of time.

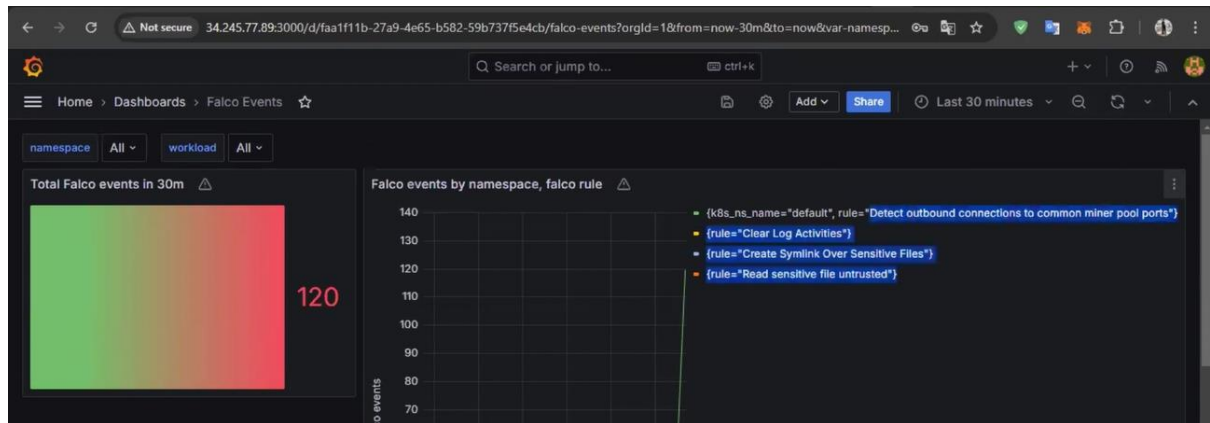


Figure 11: Grafana custom dashboard for captured Falco security events

4.5.6 Access FalcoSidekick, Grafana and Prometheus UIs:

To access the UI of monitoring tools, I created a script using port-forward to access UI endpoint within the cluster without using NodePort service type.

```
$ forward-ports.sh X
$ forward-ports.sh
1  #!/bin/bash
2
3  while :
4  do
5  ps aux | awk '$0 ~ /port-forward/ { print $2}' | xargs kill -9
6  kubectl -n monitoring port-forward --address 0.0.0.0 svc/prometheus-grafana 3000:80 &
7  kubectl -n monitoring port-forward --address 0.0.0.0 svc/prometheus-kube-prometheus-prometheus
9090:9090 &
8  kubectl -n monitoring port-forward --address 0.0.0.0 svc/prometheus-kube-prometheus-alertmanager
9093:9093 &
9  kubectl -n falco port-forward --address 0.0.0.0 svc/falco-falcosidekick-ui 2802:2802 &
10 sleep 300
11 kill %1 %2 %3 %4
12 sleep 1
13 done
14
```

Figure 12: port-forward script for accessing UIs

By integrating these tools together, we have a comprehensive monitoring stack with capability of capturing real-time breaches using eBPF during runtime in the Kubernetes cluster. There is also the alert system to notify any warning and critical security events to administrators through the Telegram channel.


```
ec2-user@ip-172-31-10-177 ~
$ k get pod -A
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
default	adservice-7fd58465b7-xr4l4	1/1	Running	0	4d12h
default	cartservice-6f4fc7c4c4-zqdjm	1/1	Running	0	4d12h
default	checkoutservice-694758bddf-jbljt	1/1	Running	0	4d12h
default	currencyservice-75d6599b9-psqpf	1/1	Running	0	4d12h
default	emailservice-b964694b9-hvtrh	1/1	Running	0	4d12h
default	frontend-b4467f874-d8svw	1/1	Running	0	4d12h
default	loadgenerator-5fcbdb5cb9-rtjb8	1/1	Running	0	4d12h
default	paymentservice-7d969fdc57-cjkjm	1/1	Running	0	4d12h
default	productcatalogservice-54cf845bc5-7mxj9	1/1	Running	0	4d12h
default	recommendationservice-75b8b64bdc-k2zdb	1/1	Running	0	4d12h
default	redis-cart-7ff8f4d6ff-wkzfq	1/1	Running	0	4d12h
default	shippingservice-85ddd6cdbc-9hvbp	1/1	Running	0	4d12h
falco	falco-dbzkc	2/2	Running	0	7m14s
falco	falco-falcosidekick-76c8dcd565-cqj77	1/1	Running	0	4d12h
falco	falco-falcosidekick-76c8dcd565-v9559	1/1	Running	0	4d12h
falco	falco-falcosidekick-ui-668947ff8f-6tnzz	1/1	Running	0	4d12h
falco	falco-falcosidekick-ui-668947ff8f-b6wzx	1/1	Running	0	4d12h
falco	falco-falcosidekick-ui-redis-0	1/1	Running	0	4d12h
falco	falco-fb55f	2/2	Running	0	7m13s
falco	falco-tl76r	2/2	Running	0	7m17s
kube-system	aws-node-6jtvk	2/2	Running	0	7m34s
kube-system	aws-node-mhvpt	2/2	Running	0	7m32s
kube-system	aws-node-qh8s5	2/2	Running	0	7m31s
kube-system	coredns-67d68bcfdc-b4shk	1/1	Running	0	4d12h
kube-system	coredns-67d68bcfdc-rrprh	1/1	Running	0	4d12h
kube-system	eks-pod-identity-agent-5bctd	1/1	Running	0	7m34s
kube-system	eks-pod-identity-agent-r4k48	1/1	Running	0	7m31s
kube-system	eks-pod-identity-agent-tlckg	1/1	Running	0	7m32s
kube-system	kube-proxy-8md8l	1/1	Running	0	7m32s
kube-system	kube-proxy-qdw8m	1/1	Running	0	7m34s
kube-system	kube-proxy-zgdpn	1/1	Running	0	7m31s
monitoring	alertmanager-prometheus-kube-prometheus-alertmanager-0	2/2	Running	0	2m4s
monitoring	prometheus-grafana-56f54d5c96-9xqs6	3/3	Running	0	2m6s
monitoring	prometheus-kube-prometheus-operator-7564fdddb7-5shzb	1/1	Running	0	2m6s
monitoring	prometheus-kube-state-metrics-754dbb4fd4-2lmxx	1/1	Running	0	2m6s
monitoring	prometheus-prometheus-kube-prometheus-prometheus-0	2/2	Running	0	2m4s
monitoring	prometheus-prometheus-node-exporter-ccg7m	1/1	Running	0	2m6s
monitoring	prometheus-prometheus-node-exporter-gqw7q	1/1	Running	0	2m6s
monitoring	prometheus-prometheus-node-exporter-jg66k	1/1	Running	0	2m6s

Figure 13: All EKS cluster pods

4.6 Workflow

In terms of workflow and data flow in this cluster, I will describe them from monitoring, capturing to alerting. Originally, Falco itself collects Linux system calls by an eBPF probe. It can then be analyzed using the Falco rule engine to detect any violations of predefined rules. Secondly, Prometheus scrapes the metrics through Falco using FalcoSidekick and stores the data for a later analysis. In addition, AlertManager is configured by predefined thresholds and rules via Prometheus, promptly triggering only security issues with a warning priority or higher to a Telegram channel. This detailed workflow ensures a clear understanding of data movement of the monitoring stacks using eBPF and the Prometheus stack within the EKS cluster.

5 Evaluation

This section provides a comprehensive analysis of the results and main research findings of my work, giving the implications from both academic and practitioner viewpoints. In most experiments, the attack surface is a Docker container and not a Pod or a Deployment in Kubernetes because it offers a similar behavior but makes it easier to test. The evaluation focuses on three experimental scenarios that support the research question and objectives.

5.1 Experiment 1: Intrusion Detection and Response

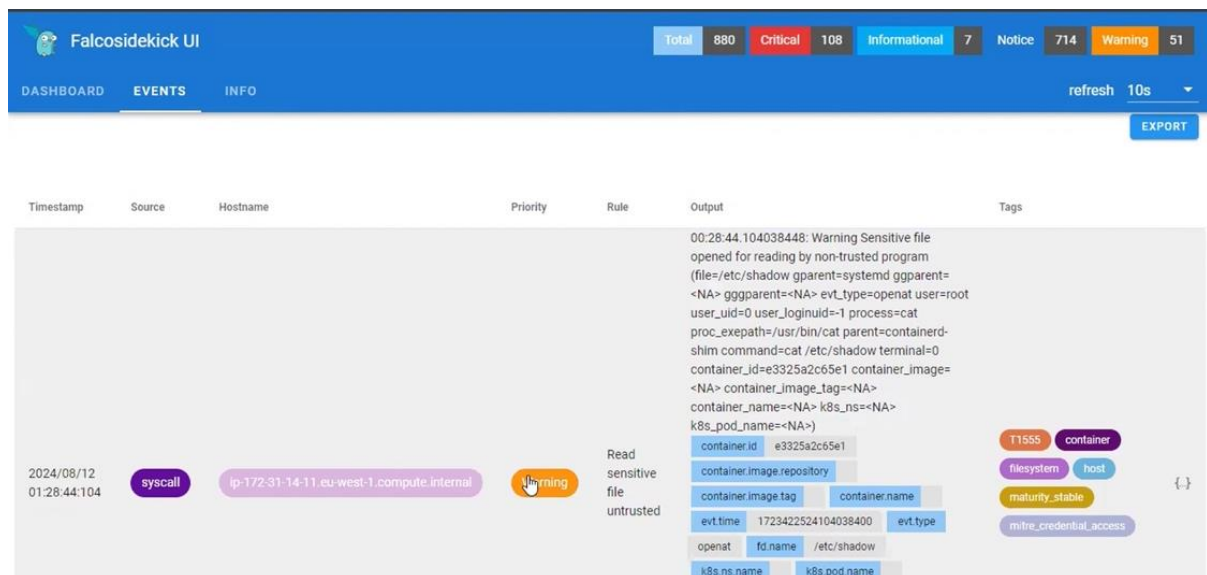
In the first experiment, attackers exploited the system by deploying containers. They did a series of harmful actions from accessing to sensitive files, creating symlinks on sensitive files for privilege escalation, sending data to external servers, and clearing up logs to clear their traces. The attackers's purpose is to harvest useful data on the system and send it to their servers.

Initially, the attacker tried to read sensitive files by executing the command ``docker run -d ubuntu:latest cat /etc/shadow``. Falco detected that the shadow file was accessed by an Ubuntu container and issued a warning priority alert.



```
[ec2-user@ip-172-31-14-11 ~]$ docker run -d ubuntu:latest cat /etc/shadow
e3325a2c65e1802921b278b503c639e4184a217a676ca62de09237fe587112b2
[ec2-user@ip-172-31-14-11 ~]$
```

Figure 14: The attack tried to view the shadow file



Timestamp	Source	Hostname	Priority	Rule	Output	Tags
2024/08/12 01:28:44:104	syscall	ip-172-31-14-11.eu-west-1.compute.internal	Warning	Read sensitive file untrusted	<pre>00:28:44.104038448: Warning Sensitive file opened for reading by non-trusted program (file=/etc/shadow gparent=systemd ggparent= <NA> gggparent=<NA> evt_type=openat user=root user_uid=0 user_loginuid=-1 process=cat proc_exepath=/usr/bin/cat parent=containerd- shim command=cat /etc/shadow terminal=0 container_id=e3325a2c65e1 container_image= <NA> container_image_tag=<NA> container_name=<NA> k8s_ns=<NA> k8s_pod_name=<NA>) container.id e3325a2c65e1 container.image.repository container.image.tag container.name evt.time 1723422524104038400 evt.type openat fd.name /etc/shadow k8s.ns.name k8s.pod.name</pre>	<div>T1555 container filesystem host maturity_stable mitre_credential_access</div>

Figure 15: Reading sensitive file activities detected by Falco

Following this, the attacker created a symlink over a sensitive file using the command ``docker run -d ubuntu:latest ln -s /etc/shadow /tmp/shadow_link``. This was detected by Falco as a potential privilege escalation attempt.



```
[ec2-user@ip-172-31-14-11 ~]$ docker run -d ubuntu:latest ln -s /etc/shadow /tmp/shadow_link
510e79713154215a84b9f7392058d8cda29c8b56099903a1793fbccbea33db3a
[ec2-user@ip-172-31-14-11 ~]$
```

Figure 16: The attack tried to create a symlink over the shadow file

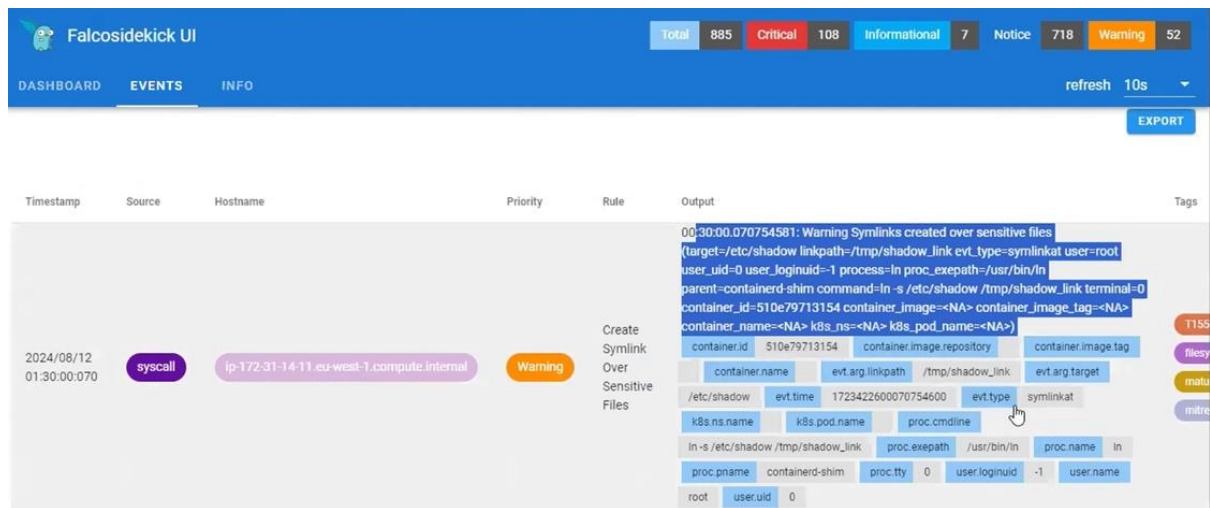


Figure 17: Creating symlink over sensitive files detected by Falco

Next, the attackers exported data to their servers by executing a sequence of commands that opened an SSH connection. Falco detected the SSH outbound connection to unauthorized servers and raised an alert for data exfiltration activity.

```
[ec2-user@ip-172-31-14-11 ~]$ docker exec -it b24 /bin/bash
bash-5.0# ssh -p 4444 ec2-user@34.245.77.89
```

Figure 18: The attack tried to establish the SSH connection to the external server

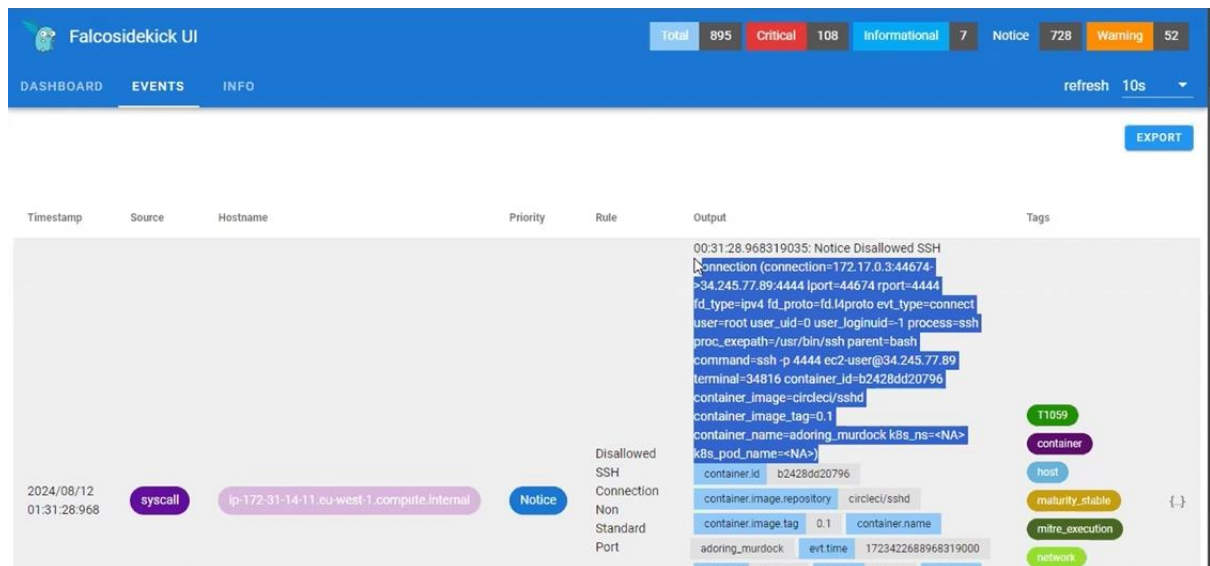


Figure 19: SSH connection detected by Falco

Finally, the attackers finally attempted to clear the log activities by issuing the command ``docker run -d -v /var/log:/var/log ubuntu:latest bash -c \"echo 'test' > /var/log/syslog\"``. This activity was detected due to the modification of log files. An alert was raised due to tampering with log files.

In summary, all detections were accurate, capturing the container ID and the command executed accurately. All the detections were sent to a Telegram channel, and logs and metrics

are traceable in a Falcosidekick UI and visualized on Grafana, providing a complete monitoring and alerting system. The detections by Falco are quite accurate and very fast in catching many kinds of harmful activities, while Prometheus captured all the metrics within 30 seconds.

Timestamp	Source	Hostname	Priority	Rule	Output	Tags
2024/08/11 22:52:42.161	syscall	ip-172-31-14-11.eu-west-1.compute.internal	Warning	Clear Log Activities	<p>21:52:42.161900789: Warning Log files were tampered (file=/var/log/syslog evt_type=opentat user=root user_uid=0 user_loginuid=1 process=bash proc_exepath=/usr/bin/bash parent=containerd-shim command=bash -c echo test > /var/log/syslog terminal=0 container_id=cb3ef49aea5f container_image=<NA> container_image_tag=<NA> container_name=<NA> k8s_ns=<NA> k8s_pod_name=<NA>)</p> <p>container.id cb3ef49aea5f container.image.repository container.image.tag container.name evntime 1723413162161900800 evt.type opentat fd.name /var/log/syslog k8s.ns.name k8s.pod.name proc.cmdline bash -c echo test > /var/log/syslog proc.exepath /usr/bin/bash proc.name bash proc.pname containerd-shim proctty 0 user.loginuid -1 user.name root user.uid 0</p>	<div>NIST_800-53_AU-10</div> <div>T1070</div> <div>container</div> <div>filesystem</div> <div>host</div> <div>maturity_stable</div> <div>mitre_defense_evasion</div>
2024/08/11 22:52:37.051	syscall	ip-172-31-14-11.eu-west-1.compute.internal	Warning	Read sensitive file untrusted	<p>21:52:37.051582592: Warning Sensitive file opened for reading by non-trusted program (file=/etc/shadow gparent=systemd gparent=<NA> gggparent=<NA> evt_type=opentat user=root user_uid=0 user_loginuid=1 process=cat proc_exepath=/usr/bin/cat parent=containerd-shim command=cat /etc/shadow terminal=0 container_id=aef4b9d0c46 container_image=<NA> container_image_tag=<NA> container_name=<NA> k8s_ns=<NA> k8s_pod_name=<NA>)</p> <p>container.id aef4b9d0c46 container.image.repository container.image.tag container.name evntime 1723413157051582500 evt.type opentat fd.name /etc/shadow k8s.ns.name k8s.pod.name proc.pname[2] systemd proc.pname[3] proc.pname[4] proc.cmdline cat /etc/shadow proc.exepath /usr/bin/cat proc.name cat proc.pname containerd-shim proctty 0 user.loginuid -1 user.name root user.uid 0</p>	<div>T1550</div> <div>container</div> <div>filesystem</div> <div>host</div> <div>maturity_stable</div> <div>mitre_credential_access</div>

Figure 20: Clear log activities detected by Falco

5.2 Experiment 2: System Destruction Attempt

In the second experiment, attackers tried to destroy system data by running a container that removes bulk data from the disk using the command `docker run -d ubuntu:latest shred -n 1 /path/to/data`. The detection was captured with the detailed logs of command execution and target data path. This test indicates that Falco can detect destructive activities with details but it cannot block this action from occurring so the data will be lost permanently if no backup measures are in place.

Timestamp	Source	Hostname	Priority	Rule	Output	Tags
2024/08/11 22:53:38.750	syscall	ip-172-31-14-11.eu-west-1.compute.internal	Warning	Remove Bulk Data from Disk	<p>21:53:38.750801852: Warning Bulk data has been removed from disk (file=<NA> evt_type=execve user=root user_uid=0 user_loginuid=1 process=shred proc_exepath=/usr/bin/shred parent=containerd-shim command=shred -n 1 /path/to/data terminal=0 exe_flags=EXE_WRITABLE container_id=f6433e3bd9c4 container_image=<NA> container_image_tag=<NA> container_name=<NA> k8s_ns=<NA> k8s_pod_name=<NA>)</p> <p>container.id f6433e3bd9c4 container.image.repository container.image.tag container.name evntime 1723413218750802000 evt.type execve fd.name k8s.ns.name k8s.pod.name proc.cmdline shred -n 1 /path/to/data proc.exepath /usr/bin/shred proc.pname shred proc.pname containerd-shim proctty 0 user.loginuid -1 user.name root user.uid 0</p>	<div>T1485</div> <div>container</div> <div>filesystem</div> <div>host</div> <div>maturity_stable</div> <div>mitre_impact</div> <div>process</div>

Figure 21: Removing bulk data activities detected by Falco

5.3 Experiment 3: Crypto Mining Deployment

In this third experiment, a custom rule was configured to detect crypto mining into Falco's rulesets. Attackers installed a crypto mining application in the Kubernetes cluster through the use of a predefined deployment YAML file. A mining process was detected by Falco due to the connection to mining pools. The detection is correct, including details of deployment and associated containers. This experiment shows that Falco can detect any bad behaviors based on customs which can be defined to fit the organization's policy. This threat was alerted with the delay of about 2 minutes so that cluster operators may intervene in this behavior before incurring financial losses.

```

! crypto-miner-faker-deployment.yaml
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    labels:
5      app: crypto-miner-faker
6    name: crypto-miner-faker
7  spec:
8    replicas: 5
9    selector:
10     matchLabels:
11       app: crypto-miner-faker
12   template:
13     metadata:
14       labels:
15         app: crypto-miner-faker
16     spec:
17       containers:
18       - image: quay.io/rhtvjmmiras/crypto-miner-faker:latest
19         imagePullPolicy: Always
20         name: crypto-miner-faker
21

```

Figure 22: The YAML file of crypto mining pods

NAMESPACE+	NAME	PF	READY	STATUS	RESTARTS	IP	NODE	AGE
default	adservice-7fd58465b7-xr4l4	•	1/1	Running	0	172.31.22.149	ip-172-31-21-203.eu-west-1.compute.internal	4d12h
default	cartservice-6f4fc7c4c4-zqdjm	•	1/1	Running	0	172.31.24.88	ip-172-31-21-203.eu-west-1.compute.internal	4d12h
default	checkoutservice-694758bddf-jbljt	•	1/1	Running	0	172.31.24.20	ip-172-31-21-203.eu-west-1.compute.internal	4d12h
default	crypto-miner-faker-5659bb9484-799tr	•	1/1Δ	RunningΔ	0	172.31.43.250	ip-172-31-34-27.eu-west-1.compute.internal	7s
default	crypto-miner-faker-5659bb9484-dzgb6	•	1/1Δ	RunningΔ	0	172.31.8.97	ip-172-31-14-11.eu-west-1.compute.internal	7s
default	crypto-miner-faker-5659bb9484-jhj2v	•	1/1Δ	RunningΔ	0	172.31.4.125	ip-172-31-14-11.eu-west-1.compute.internal	7s
default	crypto-miner-faker-5659bb9484-sk6br	•	1/1	Running	0	172.31.23.45	ip-172-31-21-203.eu-west-1.compute.internal	7s
default	crypto-miner-faker-5659bb9484-trb9w	•	1/1Δ	RunningΔ	0	172.31.46.116	ip-172-31-34-27.eu-west-1.compute.internal	7s
default	currencyservice-75d6599b9-psqpf	•	1/1	Running	0	172.31.18.178	ip-172-31-21-203.eu-west-1.compute.internal	4d12h
default	emailservice-b964694b9-hvtrh	•	1/1	Running	0	172.31.22.12	ip-172-31-21-203.eu-west-1.compute.internal	4d12h
default	frontend-b4467f874-d8swv	•	1/1	Running	0	172.31.19.140	ip-172-31-21-203.eu-west-1.compute.internal	4d12h
default	loadgenerator-5fcbdb5cb9-rtjb8	•	1/1	Running	0	172.31.21.237	ip-172-31-21-203.eu-west-1.compute.internal	4d12h
default	paymentservice-7d969fdc57-cjkjm	•	1/1	Running	0	172.31.28.116	ip-172-31-21-203.eu-west-1.compute.internal	4d12h
default	productcatalogservice-54cf845bc5-7mxj9	•	1/1	Running	0	172.31.30.113	ip-172-31-21-203.eu-west-1.compute.internal	4d12h
default	recommendationservice-75b8b64bdc-k2zdb	•	1/1	Running	0	172.31.20.199	ip-172-31-21-203.eu-west-1.compute.internal	4d12h
default	redis-cart-7ff8f4d6ff-wkz1q	•	1/1	Running	0	172.31.24.38	ip-172-31-21-203.eu-west-1.compute.internal	4d12h
default	shippingservice-85ddd6dbc-9hvbv	•	1/1	Running	0	172.31.22.2	ip-172-31-21-203.eu-west-1.compute.internal	4d12h

Figure 23: Crypto mining application deployed in the EKS cluster

2024/08/11
22:55:07:219
syscall
ip-172-31-34-27.eu-west-1.compute.internal
Critical

Detect
outbound
connections to
common miner
pool ports

```

21:55:07.219887273: Critical Outbound connection to IP/Port flagged by https://cryptoloc.ch
(ip=51.222.200.133 connection=172.31.43.250:39496->51.222.200.133:25 [ports=39496 |port=25
fd_type=ip:ip4 fd_proto=top ev_type=connect user=root user_uid=0 user_loginuid=1 process=main
proc_exe_path=/main parent=containerd-shim command=main terminal=0
container_id=66aaf104cf78 container_image=quay.io/rhtvjmmiras/crypto-miner-faker
container_image_tag=latest container_name=crypto-miner-faker k8s_ns=default
k8s_pod_name=crypto-miner-faker-5659bb9484-799tr)
container_id 66aaf104cf78 container_image_repository quay.io/rhtvjmmiras/crypto-miner-faker
container_image_tag latest container_name crypto-miner-faker ev_type time
1723143307219887400 ev_type connect fd_ipproto top fd_ipport 39496 fd_name
172.31.43.250:39496->51.222.200.133:25 fd_ip 51.222.200.133 fd_ipport 25 fd_type
ip:4 k8s_ns_name default k8s_pod_name crypto-miner-faker-5659bb9484-799tr
proc_cmdline main proc_exe_path /main proc_name main proc_pname
containerd-shim proc tty 0 user_loginuid -1 user_name root user_uid 0

```

T1496 container
host
maturity_sandbox
mitre_impact
network

Figure 24: Outbound connection to mining pools detected by Falco

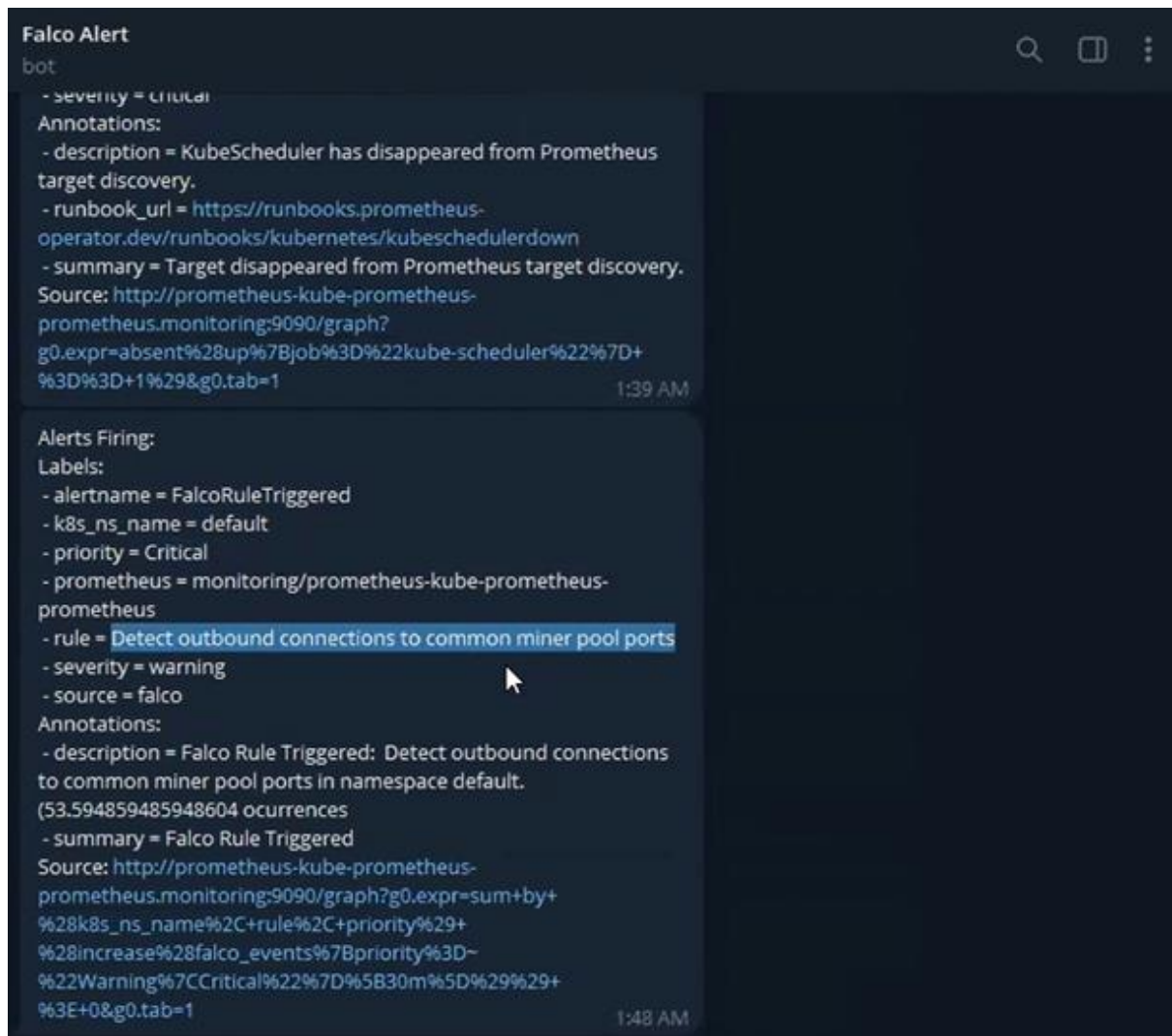


Figure 25: Alerts sent to the Telegram channel

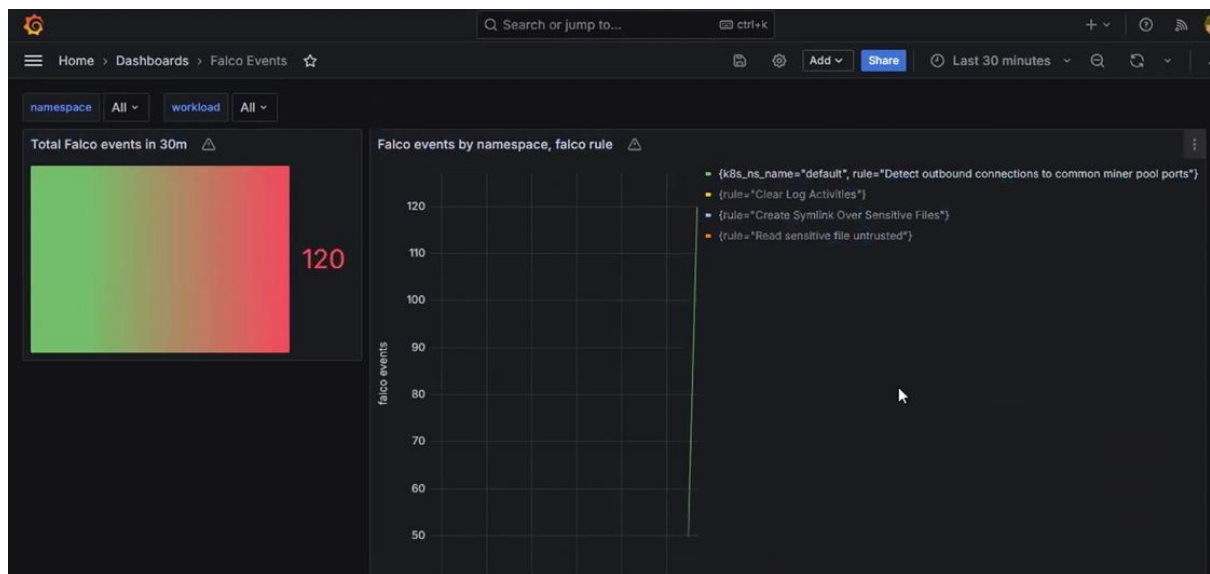


Figure 26: Grafana dashboard showing all occurred attacks

5.4 Discussion

From an academic perspective, this work represents the efficiency of eBPF-based monitoring and security tools for Kubernetes environments. It contributes to the academic understanding of how eBPF can be integrated with traditional monitoring systems to enhance security and observability. This detailed analysis of detection accuracy and system integration is an important reference for future research in the area of cloud-native security and monitoring.

From a practical point of view, the integration of Falco with Prometheus and Grafana is a realistic approach to heightening security within Kubernetes clusters. This work highlights that eBPF-based tooling is very effective in the early detection of a wide range of security threats originating from kernel events, providing a robust framework for security posture enhancement in containerized environments.

In conclusion, evaluating the experimental scenarios proves that the use of eBPF-based tools is very precise and effective at improving Kubernetes security and observability. Thanks to integration with traditional monitoring systems, it guarantees an end-to-end system-related threat detection and alerting process.

6 Conclusion and Future Work

In this work, we discuss the applicability of Extended Berkeley Packet Filter (eBPF) in enhancing observability and security in Kubernetes environments. We solve complex security challenges in the containerization world by integrating eBPF with mature monitoring tools such as Falco, Prometheus, and Grafana. This integration has paved a way for more detailed research into eBPF within Kubernetes for a revolution in cloud-native security frameworks and operational monitoring. My research demonstrates that Falco could be used for fine-grained observability with the help of eBPF, and custom rules could be implemented according to organizational purposes. Through a series of experiments, I have shown the ability of eBPF to capture real-time security incidents by assessing the log levels at the kernel, considerably increasing the granularity of monitoring. This affords detailed insights into Kubernetes operations that are critical for proactive security measures.

eBPF is a promising technology that can address the dual problems in Kubernetes regarding observability and security. The complexity of eBPF programming and its integration challenges with existing tools, alongside a steep learning curve, are headaches for operator engineers. Furthermore, legacy system integration issues can complicate the deployment of an eBPF-based solution.

Looking ahead, eBPF has the potential for extending applicability and simplifying the deployment process. Future research could focus on using eBPF to proactively block threats from occurring by considering the impact on the system performance and its feasibility. Another promising research area is the development of machine learning models that would analyze historical data and recommend optimal security policies which could be deployed using eBPF-based tools. These approaches expand the functionality of eBPF, increase its ease of use, and make it more effective in cloud-native environments.

Addressing these challenges and looking toward future research, eBPF could become an even more important foundation for securing Kubernetes environments and a more resilient and resilient security and observability platform.

References

- [1] C. Liu, Z. Cai, B. Wang, Z. Tang and J. Liu, "A protocol-independent container network observability analysis system based on eBPF," 2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS), Hong Kong, 2020, pp. 697-702, doi: 10.1109/ICPADS51040.2020.00099.
- [2] A. Sadiq, H. J. Syed, A. A. Ansari, A. O. Ibrahim, M. Alohal, and M. Elsadig, "Detection of Denial of Service Attack in Cloud Based Kubernetes Using eBPF," Applied Sciences, vol. 13, no. 8, p. 4700, 2023. doi: 10.3390/app13084700.
- [3] S. Gwak, T.-P. Doan, and S. Jung, "Container Instrumentation and Enforcement System for Runtime Security of Kubernetes Platform with eBPF," Intelligent Automation & Soft Computing, vol. 37, no. 2, pp. 1773-1786, 2023. doi: 10.32604/iasc.2023.039565.
- [4] G. Budigiri, C. Baumann, J. T. Mühlberg, E. Truyen and W. Joosen, "Network Policies in Kubernetes: Performance Evaluation and Security Analysis," 2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit), Porto, Portugal, 2021, pp. 407-412, doi: 10.1109/EuCNC/6GSummit51104.2021.9482526.
- [5] S. Miano, M. Bertrone, F. Risso, M. Tumolo and M. V. Bernal, "Creating Complex Network Services with eBPF: Experience and Lessons Learned," 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR), Bucharest, Romania, 2018, pp. 1-8, doi: 10.1109/HPSR.2018.8850758.
- [6] Ziheng Zhang and Lijun Chen. 2024. Anomaly Detection Model for Process Resource Usage in Hybrid System based on eBPF and Isolation Forest. In Proceedings of the 2023 6th International Conference on Artificial Intelligence and Pattern Recognition (AIPR '23). Association for Computing Machinery, New York, NY, USA, 1511–1517. <https://doi.org/10.1145/3641584.3641812>
- [7] Angelo Feraudo, Diana Andreea Popescu, Poonam Yadav, Richard Mortier, and Paolo Bellavista. 2024. Mitigating IoT Botnet DDoS Attacks through MUD and eBPF based Traffic Filtering. In Proceedings of the 25th International Conference on Distributed Computing and Networking (ICDCN '24). Association for Computing Machinery, New York, NY, USA, 164–173. <https://doi.org/10.1145/3631461.3631549>
- [8] Cilium, "CNI Benchmark," 2021. [Online]. Available: <https://cilium.io/blog/2021/05/11/cni-benchmark/>
- [9] GoogleCloudPlatform, "Microservices Demo,". [Online]. Available: <https://github.com/GoogleCloudPlatform/microservices-demo>