eBPF-based Cryptojacking Container Detection Framework in Kubernetes

Ri-Yeong Kim

Department of Future Convergence Technology Engineering

The Graduate School of Sungshin Women's University

eBPF-based Cryptojacking Container Detection Framework in Kubernetes

A Master's Thesis
Submitted to the
Graduate School of Sungshin Women's University
in partial fulfillment of the requirements
for the degree of
Master of Future Convergence Technology Engineering

Ri-Yeong Kim

NOV, 2024

This is to certify that we have examined the
Master's Thesis of
Ri-Yeong Kim
Submitted to Department of Future Convergence
Technology Engineering

Approved as to style and content:

Thesis Advisor ..... Seongmin Kim

Committee Chairman ..... Il-Gu Lee

Committee Member ..... Yeon-sup Lim

The Graduate School of Sungshin Women's University

ABSTRACT

eBPF-based Cryptojacking Container Detection Framework in Kubernetes

Ri-Yeong Kim
Department of Future Convergence Technology Engineering
Graduate School of
Sungshin Women's University

As the use of containers has become mainstream in the cloud environment, various security threats targeting containers have also been increasing. Among them, a notable malicious activity is a cryptojacking attack that steals resources without the consent of an instance owner to mine cryptocurrency. However, detecting such anomalies in a containerized environment is more complex because containers share the host kernel, making it challenging to pinpoint resource usage and anomalies at the container granularity without introducing significant overhead. To this end, this study proposes a new framework for real-time detection of malicious mining behavior in the cloud-native environment. By utilizing Tetragon, an eBPF-based runtime security tool, we capture system call sequences in real-time and convert them into n-gram feature sets, which were used to train machine learning models. As a result of the experiment, our framework delivers up to 99.75%

classification accuracy with low runtime monitoring overhead. Based on the findings of this study, the proposed framework is expected to serve as an effective end-to-end security solution that is well-suited for Kubernetes environments by capturing system calls from containers at runtime and detecting malicious mining behavior.

Contents

Table Contents

Figure Contents

Ⅰ. Introduction

A recent innovation in cloud-native technology with the use of containers enables automated, flexible deployment and scalable management of microservices. Container technologies offer a lightweight architecture by leveraging the host OS and minimizing deployment overhead, making them an optimal solution for ensuring the performance of real-time applications. With the emergence of representative container-based technologies, Kubernetes and Docker, enterprises are migrating their services from traditional environments based on virtual machines(VMs) to containerized environments. Kubernetes is a system that provides automation for deploying, scaling and managing docker containerized applications. Accordingly, major cloud service providers(CSPs) are offering container-based Kubernetes platforms, such as Amazon Elastic Kubernetes Service(EKS) and Azure Kubernetes Service(AKS).

However, the widespread use of containers also incurs a rise in security threats targeting these environments. In particular, cryptojacking has become one of the most severe threats, as it stealthily hijacks victim's computing resources of consolidated servers for cryptocurrency mining. According to Google Cloud's research, 86% of attacks in container environments are due to cryptojacking containers[1]. In addition, the 2022 Cloud Native Security and Usage Report by Sysdig reveals that the most frequently discovered malicious container images

are cryptojacking images, occurring at more than twice the rate of the second-highest malicious image[2]. Such unauthorized activities not only increase resource usage costs for the instance owner but also degrade performance by abusing container resources and interfering with the resource usage of benign containers.

In contrast to conventional host-based cryptojacking malware, attackers can easily distribute malicious containers through container image registries (e.g., Docker Hub) in the cloud-native environment. Basically, cryptojacking attacks involved embedding mining scripts into websites or distributing binary malware to hijack the computing power of various devices, such as servers and IoT devices[3]. However, due to the nature of the container ecosystem, anyone can upload and download images to public repositories. In other words, attackers can control numerous cryptojacking containers by exploiting widely deployed container images, leveraging the scale and flexibility of cloud-native environments to conduct extensive cryptojacking operations. Worse yet, since containers share the host kernel, attacks originating from one container can adversely affect neighboring containers, potentially compromising the entire host system and other co-located services[4,5].

Previous research primarily focused on detecting cryptojacking activities in the form of browser-based or binaries. For example, it utilizes performance-relevant metrics, such as processor time and interrupts, to achieve the signature of browser-based cryptojacking attacks[6,7]. While these metrics can detect abnormal resource usage patterns indicative of malicious activities like cryptojacking, it can result

in false positives for legitimate CPU- and memory-intensive workloads. In addition, performance monitoring in container environments is more complex than in native environments, posing challenges in utilizing these metrics effectively. Container processes are isolated and virtualized with namespaces and control groups(cgroups). Still, because containers share the host kernel, it is difficult to accurately separate and measure multiplexed metrics like interrupts and memory pages for each container. This complicates the monitoring process, making it challenging to pinpoint resource usage and anomalies at the granularity needed to detect cryptojacking activities within individual containers effectively.

To address these challenges, recent studies utilize machine learning(ML) models by constructing the feature sets acquired from the container-level system information[8,9]. In fact, datasets collected from genuine mining workloads (e.g., Bitcoin and Ethereum Miner) and blockchain network management workloads were used and labeled as malicious mining bots in [9]. Furthermore, a significant challenge arises from the use of legacy tools that continuously extract system calls in a Kubernetes environment to collect system-level information for training, which can introduce substantial runtime overhead as the number of running containers scales out.

To this end, we propose a novel cryptojacking container attack detection framework that extracts system calls from benign and cryptojacking containers to analyze the dynamic behavior information of cryptojacking containers. To monitor and capture system call sequences in real-time, we leverage Tetragon[10], an eBPF-based cloud-native

security solution, enabling security event tracking for containers deployed in clusters. Extended Berkeley Packet Filter(eBPF) is a technology that allows user-defined programs, such as tracking network activities and process executions, to run in the kernel at runtime without modifying the Linux kernel code while minimizing performance overhead as it runs in kernel space[11]. To the best of our knowledge, this study first introduces a framework that unifies both container behavior monitoring and ML-based cryptojacking detection functionality in real-time on the Kubernetes environment.

Our study's primary contributions can be summed up as follows:

- We propose a novel framework for detecting cryptojacking attacks in Kubernetes by integrating eBPF-based runtime monitoring with a ML-based detection module.
- To evaluate the versatility and efficiency of the proposed framework, we explore the optimal utilization of n-gram system call sequences for model training, using two anomalous cryptojacking containers and four benign containers.
- Our evaluation demonstrates that the proposed framework ensures scalability and real-time monitoring in cloud-native environments while effectively detecting cryptojacking attacks with high accuracy and low additional overhead.

The rest of this paper is organized as follows. Section II introduces anomaly detection in containerized environments and describes Tetragon, an eBPF-based tool. Section III discusses the structure and operational mechanism of the proposed cryptojacking container attack detection

framework. Section IV presents the results of the detection performance evaluation and performance overhead analysis of the proposed framework. In Section V, an analysis of cryptojacking workloads is provided from the perspectives of system calls and network traffic. Finally, Section VI concludes the paper by summarizing the findings and suggesting directions for future research.

Ⅱ. Background & Literature Review

1. Container Anomaly Detection

Recent containerized workloads run on multi-tenant cloud environments, sharing the host kernel and underlying computational resources with other containers. Due to the loose isolation guarantee compared to VMs, a malicious container might inhibit the execution of neighboring containers. To mitigate threats, CSPs monitor the behavior of containers to detect anomalies and protect benign containers and the host kernel. However, conventional security solutions (e.g., intrusion detection systems) only provide limited functionalities to detect anomalies in cloud-native environments. Specifically, they have limitations in meticulously tracking the internal activities of each container in an environment where multiple services coexist on a single host, thereby necessitating anomaly detection measures suitable for container environments.

To address this, researchers have proposed methodologies for detecting the malicious behavior of containers by analyzing system information (e.g., resource utilization) and system call traces[8,9,12,13]. One conventional approach involves acquiring the signature of workloads by extracting system call sequences, as these are essential for accessing host resources such as the file system, memory and network or for utilizing functionalities that require OS privileges. Similarly, container

applications can be characterized by monitoring these traces. And they are run as a single process from the host kernel's perspective, it is possible to perform anomaly detection without direct access to resources within the container.

A prior study used system calls acquired by executing various types of cryptocurrency docker images such as Bitcoin, Ethereum and Dash, as training data[9]. However, because Bitcoin and Ethereum require special hardware devices (e.g., GPU, ASIC), mining on containers with lightweight resources is not appropriate. In addition, the docker image used contains images used to manage transactions by participating in the network rather than performing cryptocurrency mining, which is not suitable for the purpose of an attacker who aims to steal resources and mine. In contrast, this study targets malicious cryptojacking that performs mining operations. The cryptocurrency we selected is Monero(XMR), which ensures strong anonymity and can be mined using general computing resources such as CPU without special equipment such as GPU or ASIC. Therefore, mining is possible with fewer resource requirements and recently, Monero mining attacks in container environments have occurred[14].

2. Tetragon

Tetragon[10] is a part of the Cloud Native Computing Foundation(CNCF) open-source project, providing eBPF-based observability and enhanced runtime security for a containerized environment. It detects system activities, such as process lifecycle, file access and host system changes, with minimal runtime overhead. Tetragon is easily deployable and compatible with Kubernetes, Docker Engine and general Linux systems. Figure 1 shows the architecture of Tetragon deployed in Kubernetes. Basically, Tetragon runs an Agent pod on each worker node in the form of DaemonSet and Operator monitors the Tracing Policy. Tetragon manages eBPF-based tracing through Tracing Policy YAML files and utilizes various hook points such as tracepoints, kprobes and uprobes. It also provides the capability to perform eBPF filtering within the kernel using selectors and take necessary actions on matching events. Finally, the Agent pod deployed on each node dumps tracking events into stdout, allowing the log collector to collect them. We deployed Tetragon in Kubernetes to trace runtime system calls and applied the Tracing Policy (See Section Ⅲ.1 for the details).

Ⅲ. Cryptojacking Container Attack Detection Framework

This section describes the overall architecture and the workflow of the proposed cryptojacking container attack detection framework. Figure 1 illustrates the two main modules of the framework: 1) System call tracing module, 2) ML-based attack detection module. In the proposed framework, System call tracing module uses Tetragon to trace the system calls of containers. Tetragon loads the eBPF program into the kernel to monitor the running pods (a set of containers) at runtime. The system calls collected by the tracing module are input to the ML-based Attack detection module. This module extracts features from raw system call sequences of cryptojacking containers and normal containers. These features are then labeled and converted into a dataset for model training. Through this process, dynamic patterns related to cryptocurrency mining can be detected. The proposed framework provides an end-to-end security solution that effectively identifies cryptojacking activities by performing real-time operations from container state capture to system call sequences monitoring through collaboration between the eBPF hooking module and the ML-based detection module. The following subsections explain the environment setup and tools used for system call tracing in Kubernetes environment and the methods for extracting features from the collected raw datasets.

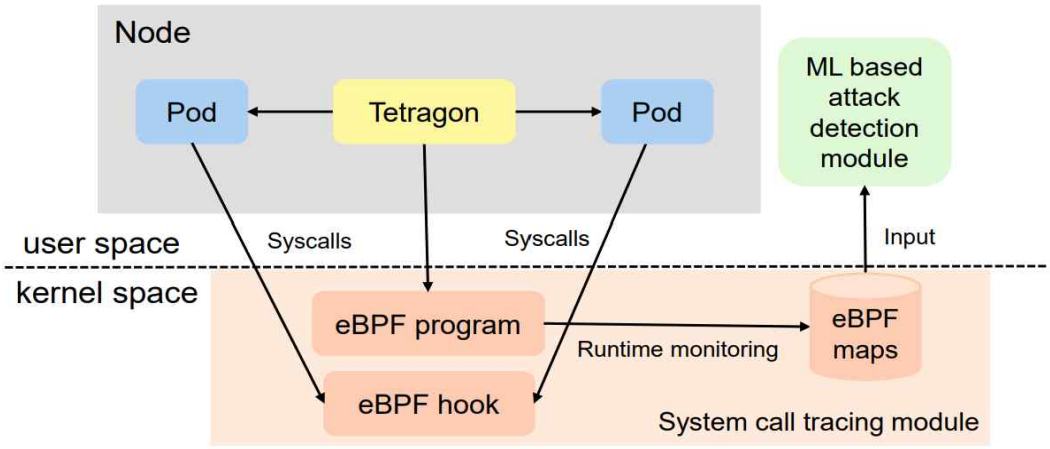


Figure 1. Overview of the Proposed Framework Architecture

1. System call Tracing Module

The conventional Linux toolchain (e.g., strace, ptrace, perf) introduces significant performance overhead when tracing system calls, as each call requires the process to be stopped twice[15]. This overhead can severely impact overall system performance, particularly in large-scale Kubernetes environments where multiple containers are running concurrently. Additionally, monitoring requires a unique identifier to distinguish running containers. Since containers share the host OS, they can be differentiated at the host level by details such as PID, TID, UID and cgroup classes. However, continuously tracking and mapping this information in complex architectures with numerous containers can be inefficient.

To address these challenges, we utilize Tetragon, an eBPF-based tool that offers several advantages. eBPF program attaches to various hooks

(tracepoints, kprobes, uprobes) within the kernel space to capture a wide range of system events. Figure 2 illustrates the operational mechanism of Tetragon in a Kubernetes environment for monitoring system calls.

When a process calls system calls, *sys_enter* event is raised in the kernel and by hooking into this point, you can retrieve the system call number. Therefore, Tetragon's Tracing Policy is designed to hook into the *sys_enter* event triggered within the *raw_syscalls* subsystem when a system call enters the kernel. The system call number is located in the fifth field of the *sys_enter* event output format, therefore the args field is defined with index 4.

eBPF programs are Just-In-Time(JIT) compiled and executed in kernel mode, allowing for immediate tracking of events as they occur. The collected real-time events are transmitted to user space via eBPF maps. Collecting logs in the kernel space prevents performance degradation caused by context switches and because the kernel continuously collects logs even when they are periodically retrieved in user space, there is no loss of log data. This enables accurate periodic inspections for container attack detection in user space with minimal overhead.

Additionally, Tetragon is Kubernetes-aware, enabling it to automatically track containers using Kubernetes metadata (such as pod names) without the need to manually track information like cgroup classes, UID, PID, or container IDs for each pod. Therefore, in this study, we utilized Tetragon to hook into the *sys_enter* event for system calls monitoring, streamlining the container tracking and mapping processes and enhancing the efficiency of the Kubernetes-integrated

detection framework.

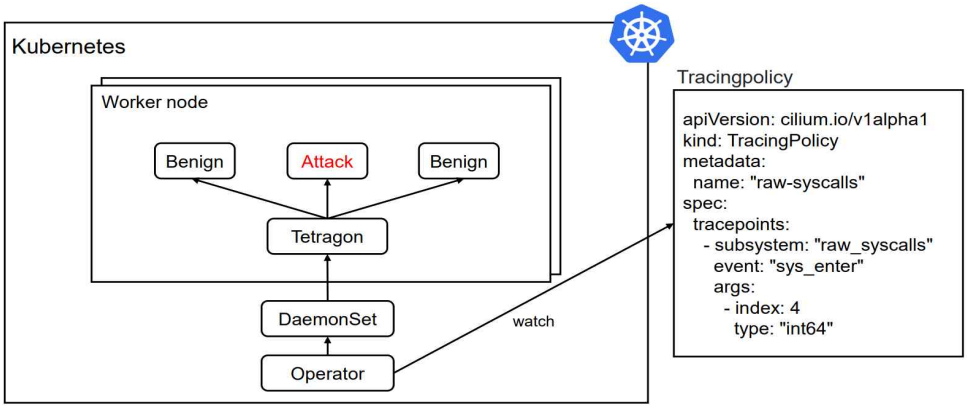


Figure 2. System call Tracing in Kubernetes

3. ML-based Attack Detection Module

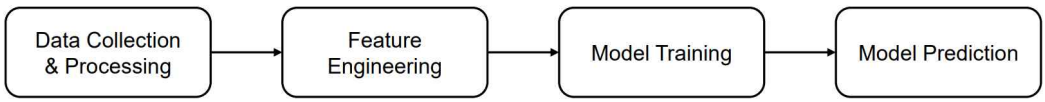


Figure 3. Flow in ML-based Attack Detection Module

To detect cryptojacking containers, we implement a ML-based attack detection system that trains on system call datasets obtained from the system call tracing module. The ML-based attack detection module follows the workflow shown in Figure 3.

In the data processing stage, system call numbers are extracted from the JSON log files generated by the tracing module, resulting in a system call sequences in the format {186, 14, 14, 3, 13, 59, 158, 218, 12, 12, ...}.

In the next step, the system call sequences are converted into feature vectors for input into the ML models. The proposed framework adopts the n-gram method, which can maintain high detection accuracy while reducing the complexity of feature transformation. Existing studies have explored ML-based anomaly detection using system calls and natural language processing(NLP) techniques are used to convert system call sequences into features for learning. Representative NLP methods used for feature representation include frequency-based[16], embedding-based[8,17] and n-gram-based techniques[9,18,19,20,21]. Specifically, [8] used a methodology that converts the system call sequences into a graph and extracts features using anonymous word embeddings. However, this requires significant computational resources for the complex feature extraction process and is time-consuming when processing system call sequences in a large container environment, potentially causing delays in real-time detection. Therefore, we aim for near-real-time detection using a low-complexity method by vectorizing the system call sequences structure into n-gram frames.

N-gram is the most commonly used method to represent system call sequences information while preserving contextual information of sentences. In n-gram, 'n' indicates the number of system calls and a sequence of system calls is extracted from the system call trace in fixed window sizes of n. When sliding the system call sequences into windows of n, it can be configured as overlapping n-grams or non-overlapping n-grams. Considering the system call sequence {186, 14, 14, 3, 13, 59, 158, 218, 12, 12} as an example with a 5-gram approach,the

overlapping n-grams are transformed into sequences such as {186, 14, 14, 3, 13} and {14, 14, 3, 13, 59}, while the non-overlapping n-grams are converted into {186, 14, 14, 3, 13} and {59, 158, 218, 12, 12}. Thus, our framework preprocesses system call logs extracted from containers and converts them into overlapping and non-overlapping n-grams for feature extraction. The choice of n is critical in both configurations as it determines how well the contextual information of the system call sequences are captured. Hence, selecting the appropriate n value is crucial depending on the model and the nature of the data. We conduct empirical analyses on various patterns of system call sequences and compare the performance of classification models through experiments. In our experiments, we vary the n values from 5 to 50 to evaluate their impact on model performance.

As a final step, the framework splits the acquired n-gram-based frame sets into training and test data for model construction. Our framework flexibly employs various ML algorithms, such as Support Vector Machines(SVM), K-Nearest Neighbors(KNN), Decision Trees(DT), Logistic Regression(LR) and Extreme Gradient Boosting(XGBoost), which are rule-based and statistical approaches, or deep learning models, such as Multi-Layer Perceptron(MLP), Recurrent Neural Network(RNN) and Convolutional Neural Network(CNN).

In this study, we assess the cryptojacking container detection performance of five ML classifiers-SVM, KNN, DT, MLP and RNN.

- SVM is a widely utilized algorithm for supervised learning, effective in handling both classification and regression challenges. It identifies

the best hyperplane that maximizes the margin between different classes, effectively separating the data points. By incorporating kernel functions, such as linear and radial basis function(RBF), SVM can effectively address non-linear relationships, making it highly adaptable to a wide variety of datasets.

- KNN is a non-parametric algorithm that classifies data points based on the majority class of their closest neighbors, as determined by a chosen distance metric. Unlike other models, KNN does not require an explicit training phase and relies instead on the dataset's structure during inference. Its straightforward implementation makes it particularly suitable for tasks with unclear or complex decision boundaries.
- DT operates as a tree-structured model that classifies data by applying a series of decision rules based on feature thresholds. By recursively partitioning the data, DT identifies the most relevant features for classification, making it interpretable and flexible for various applications.
- MLP is a type of feedforward neural network composed of multiple layers of interconnected neurons. It is capable of learning complex, non-linear relationships in the data by optimizing weights through backpropagation.
- RNN is structured to process sequential data by using a hidden state that retains information from prior inputs. This design enables RNN to recognize temporal relationships, which is advantageous for tasks involving sequential or time-sensitive data. Their proficiency

in handling sequences is particularly beneficial for applications in time-series forecasting and natural language processing.

This allows comparing the performance of various models and analyzing the impact of each model on detecting malicious cryptojacking activities.

Ⅳ. Evaluation

We evaluated the proposed framework from two perspectives:

1. Detection Performance: Comparison of attack detection performance by ML models using overlapping and non-overlapping n-gram methods, with n=5~n=50.
2. Overhead
   - 2.1: The performance overhead introduced by eBPF-based syscall monitoring compared to the baseline performance under benign workloads.
   - 2.2: Comparison of monitoring overhead between eBPF and perf, analyzed by the number of containers.

We first describe the evaluation environment and metrics, followed by a presentation of the results.

1. Experimental Setup

We use Minikube to set up a single-node Kubernetes cluster in an Ubuntu 20.04 environment. To collect data, we used two types of cryptojacking containers and four types of benign containers. The cryptojacking container images used were XMRig[22] and xmr-stak-cpu[23], both of which mine Monero. Note that Monero is one of the representative privacy-enhanced cryptocurrencies commonly used on the dark web. To diversify the benign container's activity, we used

the Web-Serving, Data-Caching and Media-Streaming benchmarks from CloudSuite 4.0[24], which provides benchmarks for cloud services and represents real-world environments based on actual software stacks. In addition, we performed MariaDB[25] service load benchmarking. Table 1 summarizes the workload specification.

Table 1. Workloads Description

Workload	Description
Web-Serving	The workload creates a cluster featuring a web server, database server, memcached server and clients. The web server, which operates Elgg, establishes connections with both the memcached and database servers. Meanwhile, the clients are involved in activities such as logging in, messaging, blogging and interacting with posts through comments and likes.
Data-Caching	The client performs requests to access data on the memcached data caching server, simulating the behavior of a Twitter data caching server.
MariaDB	MariaDB is an open-source relational database management system. Using the benchmarking tool sysbench, it performs read and write operations as well as transaction processing commands.
Media-Streaming	The workload runs nginx web server to host videos of various lengths and qualities. The client performs load testing on the server by requesting different videos based on httperf.
XMRig, xmr-stak-cpu	XMRig and xmr-stak-cpu are representative open source-based Monero mining programs that support CPU-friendly RandomX, allowing users to mine Monero efficiently.

2. Performance of the Detection Model

1) Environment and Metrics

To evaluate the classifiers, we perform multi-class classification not only to compare cryptojacking containers and benign containers but also to analyze the different behavior patterns of each application in detail. Therefore, the data transformed into n-gram frames is labeled from 0 to 5. The labeled data is divided into training and test data at 70:30.

We implement SVM, KNN, MLP and DT models using the Python-SKlearn library and train them with default parameters. The RNN was implemented using TensorFlow and Keras, with two LSTM layers containing 35 and 80 units, respectively, each followed by a 20% dropout. A final Dense layer with a softmax activation function was used for multi-class classification.

We used well-known Accuracy, Precision, Recall and F1-Score as evaluation metrics to evaluate the performance of each classification model. Note that the metrics are calculated based on the indicators: TP(True Positives), TN(True Negatives), FP(False Positives) and FN(False Negatives).

- Accuracy: This metric indicates the proportion of observations that are correctly predicted, relative to the total number of observations.

$$Accuracy = \frac{TP+TN}{TP+FP+FN+TN} \times 100 \quad (1)$$

- Precision: This metric calculates the proportion of true positive predictions relative to all predictions identified as positive. It evaluates the precision of positive predictions by focusing solely on those classified as positive.

$$Precision = \frac{TP}{TP+FP} \times 100 \quad (2)$$

- Recall: This metric measures the proportion of true positive predictions out of all actual positive observations.

$$Recall = \frac{TP}{TP+FN} \times 100 \quad (3)$$

- F1-Score: This metric calculates the harmonic mean between Precision and Recall, offering a unified score that equally considers both aspects.

$$F1-Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \times 100 \quad (4)$$

2) Performance Evaluation Results

To determine the most effective n (length of the n-gram), we varied the value of n from 5 to 50 and compared the performance of the five models using overlapping and non-overlapping n-gram frames. Figure 4 and Figure 5 show the model accuracy for overlapping and non-overlapping n-grams, respectively. For SVM, the accuracy steadily increased with larger n-sizes in overlapping n-grams and showed a similar trend in non-overlapping n-grams. However, for KNN, the accuracy decreased as n increased in both cases. MLP showed accuracies ranging from 95.92% to 97.52% in overlapping n-grams, but its performance decreased with larger n values in non-overlapping n-grams, reaching an accuracy of 69.95% at n=50. The rule-based DT maintained high accuracy above 99.59% for all n values in overlapping n-grams, with slight fluctuations. In non-overlapping n-grams, DT showed the highest accuracy of 99.24% at n=5, but the accuracy slightly decreased as the n-gram size increased. In the overall detection performance, the RNN showed the highest accuracy of 99.75% at n=40 in overlapping n-grams, as illustrated by Figure 6, which displays the confusion matrix (0 = XMRig, 1 = xmr-stak-cpu, 2 = Media-Streaming, 3 = Data-Caching, 4 = Web-Serving, 5 = MariaDB). Furthermore, even though the accuracy of the RNN diminished in the non-overlapping n-gram configuration, it continued to outperform the other models.

Table 2 and Table 3 show the best performance of each model in overlapping and non-overlapping n-grams, respectively, indicating that all

models generally performed better with overlapping n-grams. The result can be interpreted as overlapping n-grams capturing sequential features more effectively by overlapping sequences and providing more information for model training, although this increases computational costs as training data increases. Also, many duplicate frames can be generated if specific system calls have high call rates. If these identical frames are biased towards a particular class, it can cause the model to overfit that class's patterns. On the other hand, non-overlapping n-grams, which reflect features without overlapping sequences, allow for more efficient training with relatively less data but may not sufficiently capture continuous contextual information. The optimal n value also varies between models due to how each model learns the contextual information from the data. Overall, the 5-gram demonstrated decent performance across multiple models.

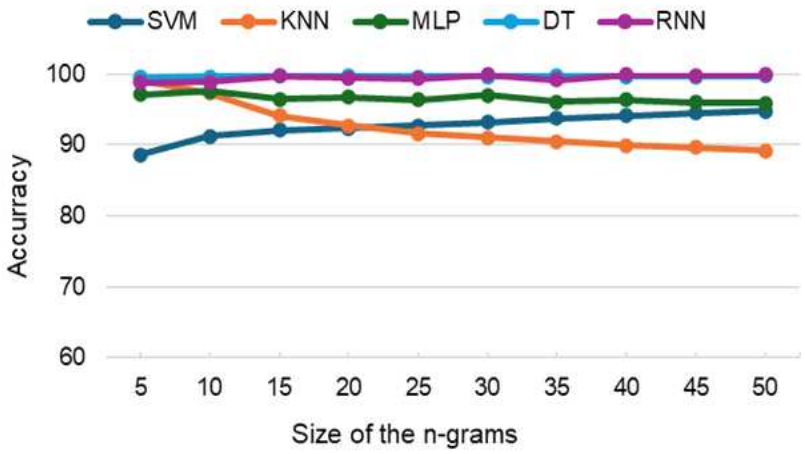


Figure 4. Model Accuracy in Overlapping n-grams (n=5~n=50)

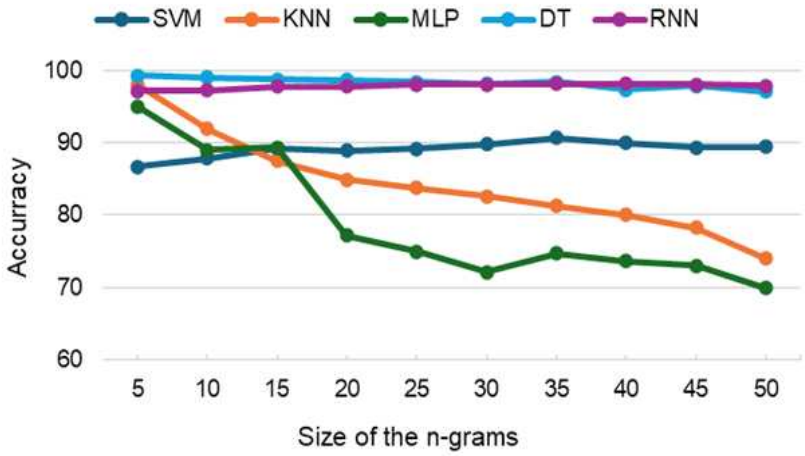


Figure 5. Model Accuracy in Non-overlapping n-grams (n=5~n=50)

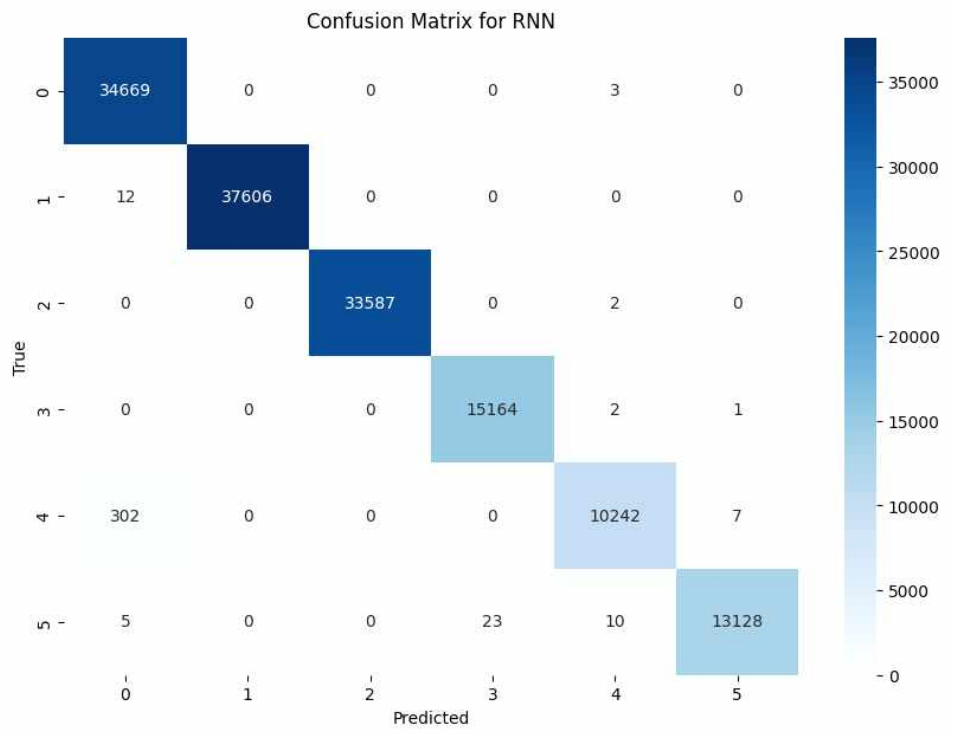


Figure 6. Confusion Matrix for RNN at n=40 in Overlapping n-grams

Table 2. Model Best Performance in Overlapping n-grams

n	ML Model	Accurracy (%)	Precision (%)	Recall (%)	F1-Score (%)
50	SVM	94.75	95.01	94.75	94.72
5	KNN	99.15	99.15	99.15	99.15
10	MLP	97.53	97.56	97.53	97.53
35	DT	99.67	99.67	99.67	99.67
40	**RNN**	**99.75**	**99.75**	**99.75**	**99.75**

Table 3. Model Best Performance in Non-overlapping n-grams

n	ML Model	Accurracy (%)	Precision (%)	Recall (%)	F1-Score (%)
35	SVM	90.65	90.82	90.82	90.26
5	KNN	98.05	98.05	98.05	98.04
5	MLP	95.00	94.97	95.00	94.87
5	**DT**	**99.24**	**99.24**	**99.24**	**99.24**
40	RNN	98.12	98.18	98.12	98.14

4. Systematic Overhead

Using eBPF enables real-time monitoring of system calls at the kernel level. However, The additional resource usage introduced by monitoring has the potential to interfere with the execution of application tasks, possibly leading to increased latency. To evaluate this impact, latency measurements were compared under identical workloads, both with and without system calls monitoring enabled. Each experiment was repeated 10 times for each workload, with the average latency used as the baseline performance. The results of the overhead analysis are shown in Table 4.

In our experiments, the latency overhead varied depending on the characteristics of the workload. For network I/O tasks such as Web-Serving, including interactions with web services or backend databases, the overhead was relatively low, ranging from 1.07x to 1.18x. However, for memory and I/O-intensive tasks such as Data-Caching, the overhead increased to 2.98x. These results suggest that system calls monitoring in memory-centric tasks can incur significant resource costs. To optimize this overhead, adjusting the system calls monitoring frequency according to the characteristics of the workload, especially for I/O-intensive tasks, can help minimize the overhead while maintaining detection performance.

Additionally, as the number of containers to be monitored increases, we evaluated the impact of eBPF-based monitoring and perf-based monitoring, a commonly used Linux monitoring tool on containers. The

experiment was performed on the Mariadb container and latency was measured for 1, 3, 5, 7 and 10 concurrent monitoring instances. The experimental results are shown in Figure 7 and the values in parentheses represent the ratio of latency increase compared to baseline performance without monitoring.

In the case of eBPF, in the case of 1 monitoring instance, the latency increased to 1.52x compared to the baseline, but in the case of 10 monitoring instances, the latency tended to decrease to 1.28x. This indicates that eBPF can maintain relatively stable performance even as the number of containers increases by processing data directly within the kernel and minimizing user space operations. On the other hand, perf-based monitoring showed a latency of 2.15x under a single monitoring instance, which further escalated to 4.04x as the number of monitoring instances increased to 10. This result demonstrates a structural limitation of perf, where the overhead increases linearly with the number of monitored containers due to the frequent context switching between user space and kernel space required during the data collection process. In conclusion, eBPF showed that it is suitable in terms of scalability while maintaining low latency even as the number of monitored containers increases.

Despite the performance overhead, it is important to highlight the important role of eBPF-based system call monitoring in detecting cryptocurrency mining. Cryptojacking is a significant threat that covertly consumes CPU and memory resources, degrading system performance and potentially causing severe system failures. Therefore, considering the

benefits of cryptojacking detection, the overhead introduced by eBPF monitoring is a reasonable cost to ensure early detection and mitigation of cryptojacking activities before excessive resource consumption occurs.

Table 4. Latency Overhead in System calls Monitoring

Workload	Latency (ms)		Overhead
	Without Monitoring	With Monitoring	
Web-Serving (BrowsetoElgg)	406.636	434.058	1.07x
Web-Serving (AccessHomePage)	344.916	370.972	1.08x
Web-Serving (SendMessage)	153.971	181.111	1.18x
MariaDB	19.263	29.24	1.52x
Data-Caching	3.783	11.29	2.98x

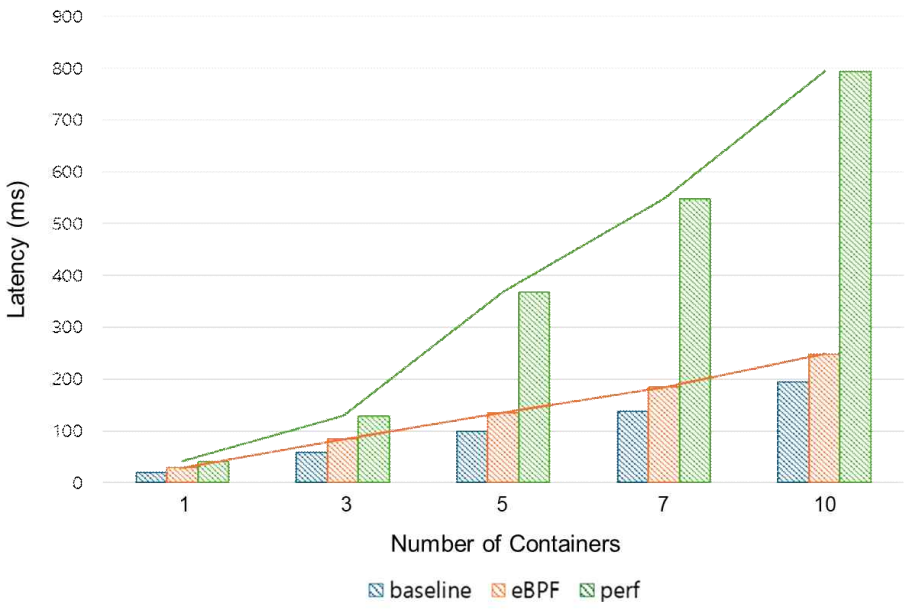


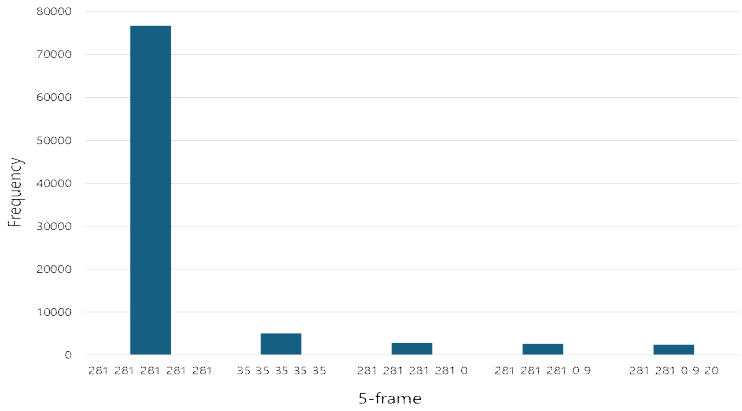
Figure 7. Latency Overhead for eBPF and perf Monitoring for Different Numbers of Containers

Ⅴ. Discussion

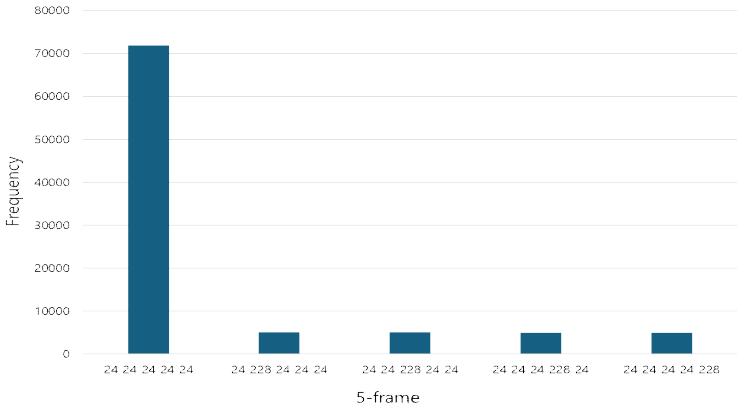
1. System calls Analysis for Cryptojacking Workloads

Finally, we conduct statistical analysis on collected system call sequences to characterize the unique system call pattern of cryptojacking containers. Figure 8 illustrates the top 5 most frequent 5-gram overlapping frames for six different containers. As shown in the figure, the cryptojacking containers XMRig and xmr-stak-cpu system call patterns significantly differ from the benign containers.

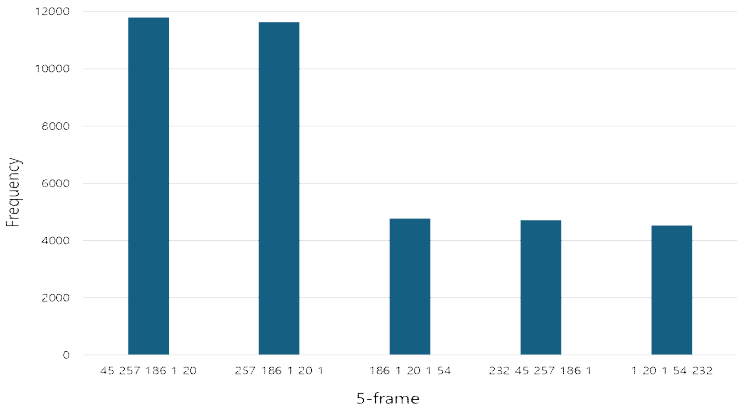
Mining programs primarily perform repetitive tasks such as sending and receiving network-based operations and calculating hash values. For example, they periodically perform computational tasks, submit the results via the network and receive new tasks, forming specific system call patterns. In particular, In the case of XMRig and xmr-stak-cpu, the frames {281, 281, 281, 281, 281} and {24, 24, 24, 24, 24}, respectively, appear with high frequency. System call number 281 corresponds to sys_epoll_pwait, which is used to efficiently handle network I/O operations related to poll communication, maximizing the utilization of system resources. System call number 24 corresponds to sys_sched_yield, which is used to return the allocated CPU time of a specific thread to the scheduler, enhancing the overall efficiency of the program. As such, mining programs frequently generate system calls that optimize resource usage due to the large volume of computational tasks they perform.



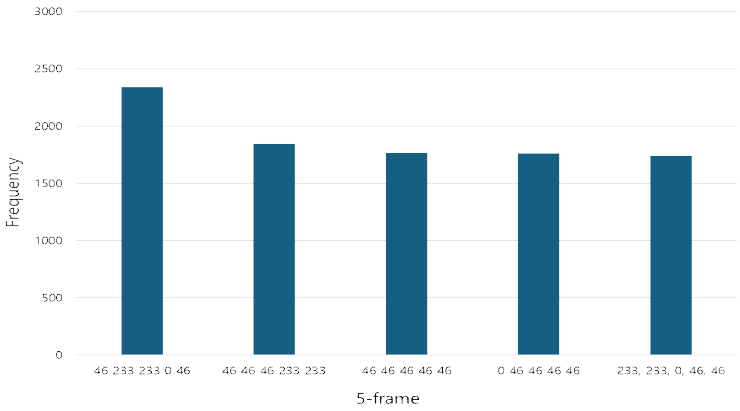
(a) XMRig



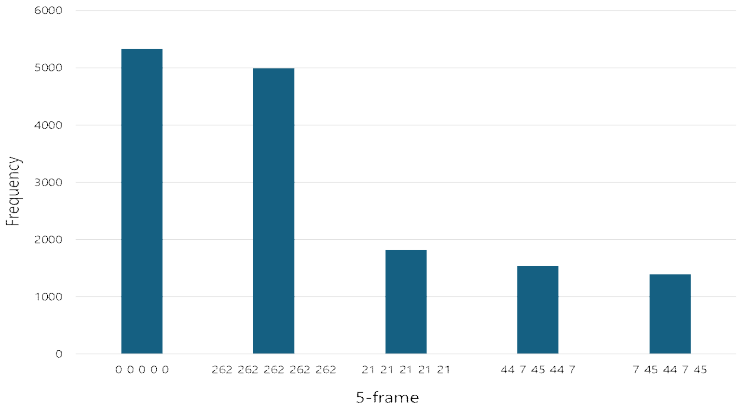
(b) xmr-stak-cpu



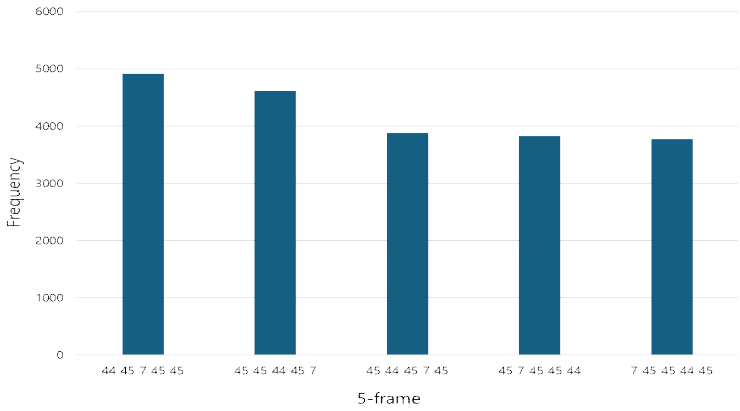
(c) Media-Streaming



(d) Data-Caching



(e) Web-Serving



(f) MariaDB

Figure 8. Top 5 5-gram Frames in Containers Traces

2. Flow-based Analysis

While analyzing system calls offers critical insights into containers' internal behavior, analyzing network traffic provides additional visibility into how containers interact with external systems. In this study, as discussed in Section 5.1, we observed cryptojacking containers communicating with mining pools, sending and receiving tasks through system calls related to networking (e.g., send and recv). This behavior can also manifest as distinct communication patterns in network traffic, indicating mining activity from a network perspective.

Miners typically form mining pools to collaborate, thereby increasing their chances of successfully computing hashes and sharing the rewards. This allows for detecting mining activities by utilizing the IP addresses or domain information of known mining pools. However, miners can evade detection by connecting to unknown servers or using DNS over TLS. Therefore, learning the recurring network communication patterns of mining workloads and applying ML techniques can enable more effective detection[26,27,28].

Tools such as packet analyzers like Wireshark or traffic flow monitoring protocols like NetFlow/IPFIX can be used for data collection, but these tools have limitations when analyzing network traffic in large-scale Kubernetes cluster environments. In Kubernetes, network traffic is distributed across Pods, forming complex mesh network patterns, making it difficult to monitor traffic from a single location. Additionally, monitoring at the host interface can result in the loss of

association with the original Pod, making it challenging to accurately trace the source of traffic. Deploying traditional network monitoring tools to all Pods is also complex and inefficient.

Similarly, eBPF can be used to effectively monitor network traffic in containerized environments, solving this problem. eBPF programs can be attached to Traffic Control(TC) hooks in the network stack to monitor traffic. In virtualized environments, it is particularly important to monitor the container's veth interface before network encapsulation in order to track the container's IP address and port. In this study, we utilized an eBPF-based network flow monitoring tool, bpfFlowMon, to capture network traffic flows of cryptojacking workloads in a Kubernetes environment. The tool was packaged into a Docker image and deployed across the cluster as a DaemonSet, allowing for the capture and analysis of pod-level network activity on each node.

Figure 9 shows the relationship between Out_Bytes and In_bytes, as well as Out_Pkts and In_Pkts, based on the extracted network traffic data. Notably, mining traffic exhibits relatively higher In_Bytes and In_Pkts values compared to Out_Bytes and Out_Pkts. This pattern results from the mining program continuously receiving tasks from the server, performing the tasks and sending the results back to the server. By training ML models on these traffic patterns, we can effectively distinguish between normal and mining traffic. Based on the results above, we conclude that by combining system call patterns with network flow-based features, container behavior can be more accurately reflected, leading to improved performance in detecting cryptojacking containers.

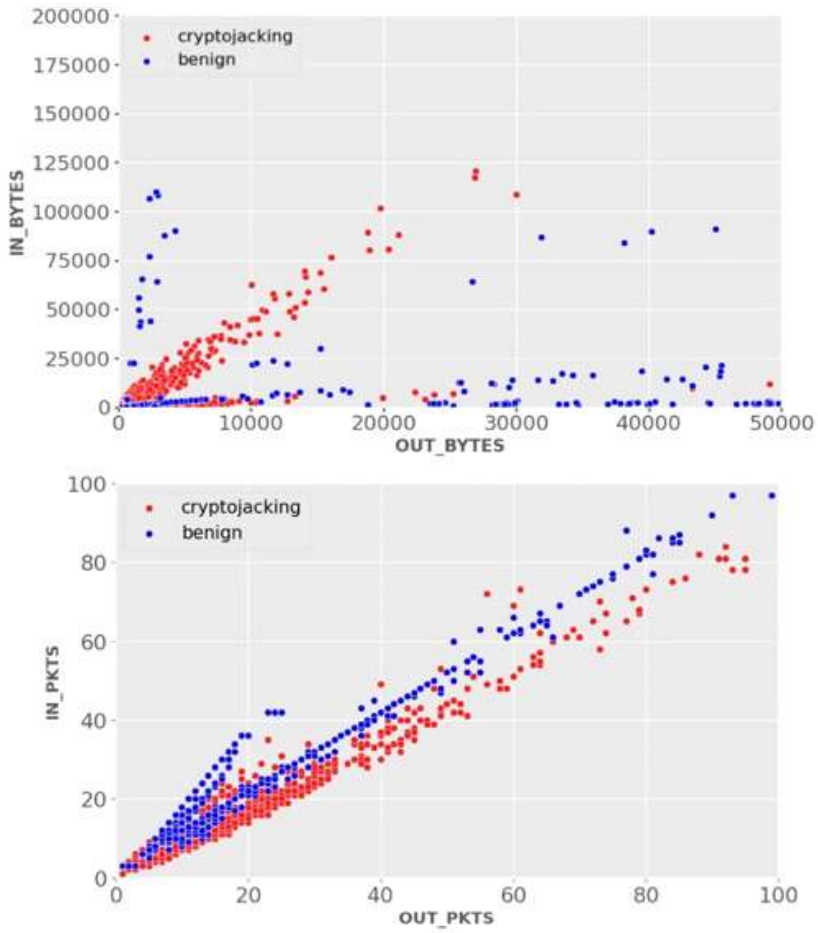


Figure 9. Distribution of Flow-level Feature Between Benign and Cryptojacking Containers

Ⅵ. Conclusion

In this paper, we proposed a framework that combines eBPF-based system calls tracing with ML-based attack detection to analyze the state of containers in real-time and effectively detect anomalous behavior in a Kubernetes environment. eBPF enables precise monitoring by tracking system calls at the kernel level while minimizing performance degradation, allowing for real-time analysis of container activities. By applying the n-gram technique to convert system call sequences into feature vectors and evaluating various ML models, we found that the RNN model achieved the highest performance with an accuracy of 99.75%. Additionally, we observed that the performance overhead caused by system calls monitoring varied depending on the workload, with higher overhead occurring in memory I/O-intensive tasks. To mitigate this, adjusting the monitoring frequency of system calls in such tasks could be considered. And in an experiment comparing the monitoring overhead based on eBPF and perf, as the number of containers increases, eBPF was shown to be more suitable in terms of scalability than perf, maintaining lower latency even when the number of monitoring targets increases.

In addition, we suggest that leveraging eBPF's network traffic monitoring capabilities to analyze mining activities and combining network traffic-based features with system call patterns, can enhance

cryptojacking detection performance. Moreover, the proposed framework can be applied not only to cryptojacking detection but also to other types of attack detection. Future research will focus on introducing additional feature extraction techniques to more precisely analyze network activity and process behavior and evaluating the performance under more complex threat scenarios.

References

[1] “Threat Horizons - Cloud Threat Intelligence November 2021. Issue 1”, access ed on August 2024, https://bit.ly/41THxbT.

[2] “Sysdig 2022 Cloud-Native Security and Usage Report”, accessed on August 2 024, https://sysdig.com/2022-cloud-native-security-and-usage-report/

[3] Tekiner, Ege, et al. “SoK: cryptojacking malware.” 2021 IEEE European Symp osium on Security and Privacy (EuroS&P). IEEE, 2021.

[4] Sultan, Sari, Imtiaz Ahmad, and Tassos Dimitriou. “Container security: Issues, challenges, and the road ahead.” IEEE access 7 (2019): 52976-52996.

[5] Jun-hee Lee, Jae-hyun Nam, & Jin-woo Kim (2023). “Analysis of the Impact of Host Resource Exhaustion Attacks in a Container Environment.” Journal of the Korea Institute of Information Security & Cryptology, 33(1), 87-97.

[6] Ning, Rui, et al. “Capjack: Capture in-browser crypto-jacking by deep capsule network through behavioral analysis.” IEEE INFOCOM 2019-IEEE Conference on Computer Communications. IEEE, 2019.

[7] Gomes, Fábio, and Miguel Correia. “Cryptojacking detection with cpu usage m etrics.” 2020 IEEE 19th International Symposium on Network Computing and Applications (NCA). IEEE, 2020.

[8] Iacovazzi, Alfonso, and Shahid Raza. “Ensemble of random and isolation forest s for graph-based intrusion detection in containers.” 2022 IEEE International C onference on Cyber Security and Resilience (CSR). IEEE, 2022.

[9] Karn, Rupesh Raj, et al. “Cryptomining detection in container clouds using sys tem calls and explainable machine learning.” IEEE transactions on parallel and

distributed systems 32.3 (2020): 674-691.

[10] “Tetragon”, accessed on August 2024, https://tetragon.io/

[11] “eBPF”, accessed on August 2024, https://ebpf.io/

[12] Zhan, Mengqi, et al. “Coda: Runtime detection of application-layer cpu-exhaustion dos attacks in containers.” IEEE Transactions on Services Computing 16.3 (2022): 1686-1697.

[13] Wang, Yulong, et al. “Unsupervised anomaly detection for container cloud via bilstm-based variational auto-encoder.” ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2022.

[14] “Threat Actors leverage Docker Swarm and Kubernetes to mine cryptocurrency at scale”, accessed on August 2024, https://securitylabs.datadoghq.com/articles/threat-actors-leveraging-docker-swarm-kubernetes-mine-cryptocurrency/

[15] Chang, Hyunseok, et al. “Microservice fingerprinting and classification using machine learning.” 2019 IEEE 27th International Conference on Network Protocols (ICNP). IEEE, 2019.

[16] Ahmed, Maheli, and Mohammed Nasir Uddin. “Cyber attack detection method based on nlp and ensemble learning approach.” 2020 23rd International Conference on Computer and Information Technology (ICCIT). IEEE, 2020.

[17] Darabian, Hamid, et al. “Detecting cryptomining malware: a deep learning approach for static and dynamic analysis.” Journal of Grid Computing 18 (2020): 293-303.

[18] Zhang, Xinrun, et al. “Early detection of host-based intrusions in Linux environment.” 2020 IEEE International Conference on Electro Information Technology (EIT). IEEE, 2020.

[19] Subba, Basant, Santosh Biswas, and Sushata Karmakar. “Host based intrusio n detection system using frequency analysis of n-gram terms.” TENCON 2017 -2017 IEEE Region 10 Conference. IEEE, 2017.

[20] Wang, Xiali, and Xiang Lu. “A Host Based Anomaly Detection Framework Using XGBoost and LSTM for IoT Devices.” Wireless Communications and M obile Computing 2020.1 (2020): 8838571.

[21] Hoang, Dang Kien, and Duy Loi Vu. “Iot malware classification based on sys tem calls.” 2020 RIVF International Conference on Computing and Communicat ion Technologies (RIVF). IEEE, 2020.

[22] “XMRig”, accessed on August 2024, https://hub.docker.com/r/miningcontainers/xmrig

[23] “xmr-stak-cpu”, accessed on August 2024, https://hub.docker.com/r/timonmat/ xmr-stak-cpu/

[24] “CloudSuite 4.0”, accessed on August 2024, https://github.com/parsa-epfl/cloudsuite

[25] “MariaDB”, accessed on August 2024, https://hub.docker.com/_/mariadb

[26] i Muñoz, Jordi Zayuelas, José Suárez-Varela, and Pere Barlet-Ros. “Detecting cryptocurrency miners with NetFlow/IPFIX network measurements.” 2019 IEEE International Symposium on Measurements & Networking (M&N). IEEE, 2019.

[27] Pastor, Antonio, et al. “Detection of encrypted cryptomining malware connecti ons with machine and deep learning.” IEEE Access 8 (2020): 158036-158055.

[28] Caprolu, Maurantonio, et al. “Cryptomining makes noise: Detecting cryptojacki ng via machine learning.” Computer Communications 171 (2021): 126-139.

논문 개요

Kubernetes 환경에서
eBPF 기반 크립토재킹 컨테이너 탐지 프레임워크

김리영
미래융합기술공학과
성신여자대학교 대학원

클라우드 환경에서 컨테이너의 사용이 주류로 자리 잡으면서 이를 표적으로 삼는 다양한 보안 위협도 증가하고 있다. 특히 인스턴스 소유자의 동의 없이 리소스를 무단으로 사용하여 암호화폐를 채굴하는 크립토재킹(Cryptojacking) 공격이 주목할만한 위협으로 대두되고 있다. 하지만 컨테이너화된 환경에서는 컨테이너가 호스트 커널을 공유하기 때문에 리소스 사용 및 이상 징후를 세밀하게 탐지하는 과정에서 상당한 오버헤드가 발생하며 이로 인해 탐지 과정이 더욱 복잡해진다. 이를 해결하기 위해 본 연구에서는 클라우드 네이티브 환경에서 악성 채굴 활동을 실시간으로 탐지하기 위한 새로운 프레임워크를 제안한다. eBPF 기반 런타임 보안 도구인 Tetragon을 활용하여 시스템 콜 시퀀스를 실시간으로 추적하고, 이를 n-gram 피처 세트로 변환하여 머신러닝 모델 학습에 사용하였다. 실험 결과, 제안된 프레임워크는 최소한의 런타임 모니터링 오버헤드로 최대 99.75%의 정확도를 달성하였다. 연구 결과를 바탕으로 제안된 프레임워크는 런타임에 컨테이너의 시스템 콜을 모니터링하여 악성 채굴 동작을 효과적으로 감지하며, Kubernetes 환경에 적합한 end-to-end 보안 솔루션으로 활용될 수 있을 것으로 기대된다.

Acknowledgements

석사 과정 동안 아낌없는 지도와 격려해주신 김성민 교수님께 진심으로 감사의 말씀을 드립니다. 교수님의 지도 덕분에 많이 배우고, 성장할 수 있었으며 이를 바탕으로 본 연구를 성공적으로 마칠 수 있었습니다. 또한 바쁘신 와중에도 조언과 도움을 주신 이일구 교수님과 임연섭 교수님께도 감사의 뜻을 전합니다.