Cryptomining Detection in Container Clouds Using System Calls and Explainable Machine Learning

Rupesh Raj Karn, Prabhakar Kudva, Hai Huang,
Sahil Suneja, and Ibrahim (Abe) M. Elfadel, *Senior Member, IEEE*

Abstract—The use of containers in cloud computing has been steadily increasing. With the emergence of Kubernetes, the management of applications inside containers (or pods) is simplified. Kubernetes allows automated actions like self-healing, scaling, rolling back, and updates for the application management. At the same time, security threats have also evolved with attacks on pods to perform malicious actions. Out of several recent malware types, cryptomining has emerged as one of the most serious threats with its hijacking of server resources for cryptocurrency mining. During application deployment and execution in the pod, a cryptomining process, started by a hidden malware executable can be run in the background, and a method to detect malicious cryptomining software running inside Kubernetes pods is needed. One feasible strategy is to use machine learning (ML) to identify and classify pods based on whether or not they contain a running process of cryptomining. In addition to such detection, the system administrator will need an explanation as to the reason(s) of the ML's classification outcome. The explanation will justify and support disruptive administrative decisions such as pod removal or its restart with a new image. In this article, we describe the design and implementation of an ML-based detection system of anomalous pods in a Kubernetes cluster by monitoring Linux-kernel system calls (syscalls). Several types of cryptominers images are used as containers within an anomalous pod, and several ML models are built to detect such pods in the presence of numerous healthy cloud workloads. Explainability is provided using *SHAP*, *LIME*, and a novel auto-encoding-based scheme for LSTM models. Seven evaluation metrics are used to compare and contrast the explainable models of the proposed ML cryptomining detection engine.

Index Terms—Cryptomining, docker, kubernetes, containers, machine learning, explainability, pod, anomaly

1 Introduction

In CLOUD computing, containers are operating-system-level virtualization abstractions for running isolated systems on a host using a single kernel. A container image is a lightweight, stand-alone, executable package that includes sufficient components such as code, system tools, system libraries, and settings to run cloud applications. The use of containers in cloud computing has grown significantly. To provide cloud service elasticity and timely response, various commercial clouds service providers such as Amazon, Google, IBM, and Microsoft are investing more into the micro-service instantiation using containerized environments. Renowned containerizing tools include Docker, Kubernetes, and OpenVZ. In this paper, we use Kubernetes, an open-source system for the automated deployment, scaling, and management of applications inside containers. It provides capabilities for load balancing, storage orchestration, automated roll-outs and rollbacks, batch execution, self-healing, horizontal and vertical scaling on the top of containers [1].

The advancement in container technology has in turn triggered several cyber security threats. To counter them, several types of cyber-attack detection schemes are available to secure applications in containers [2], [3]. But all of these approaches assume that the underlying container's image is healthy with botnets injecting faults in running containers. But unfortunately, this is not always the case as an attacker can replace a healthy container image with an infected one, which can run undetected under traditional cyber-security detectors. Recent reports [4], [5], [6] indicate that Dockerhub has found many of its images were being used maliciously to mine for cryptocurrencies. Since the Docker image can be accessed without authentication, attackers have used it to deploy cryptojacking malware. The infected Docker images were subsequently removed from the Docker repository. These miners hijack the container, sharing the computing resources (CPU and memory) between deployed applications and miners. In an extreme case, mining operations use most of the pod's compute resources so that only a fraction of resources is available for legitimate applications whose performance will be necessarily degraded. McAfee malware protection software has reported that new families of miners were found to attack Microsoft Windows and Mac OS, with mining attacks increasing by 29 percent between the fourth quarter of 2018 and the first quarter of 2019 [7]. The Nanshou campaign [8] has found that over 50,000 servers belonging to companies in the healthcare, telecommunications, media,

- *Rupesh Raj Karn and Ibrahim (Abe) M. Elfadel are with the Center for Cyber-Physical Systems, Khalifa University, Abu Dhabi, UAE. E-mail: {rupesh.karn, ibrahim.elfadel}@ku.ac.ae.*
- *Prabhakar Kudva, Hai Huangh, and Sahil Suneja are with IBM Research, Yorktown Heights, NY 10598. E-mail: {kudva, haih, suneja}@us.ibm.com.*

Manuscript received 27 Jan. 2020; revised 25 July 2020; accepted 30 Sept. 2020. Date of publication 6 Oct. 2020; date of current version 22 Oct. 2020. (Corresponding author: Ibrahim (Abe) M. Elfadel.)
Recommended for acceptance by J. Lange
Digital Object Identifier no. 10.1109/TPDS.2020.3029088

and IT sectors were hijacked for cryptomining. As per the Kaspersky lab, in 2018 the number of cryptomining attacks increased by more than 83 percent with respect to the previous year [9]. Cryptominers prevent the deployed application from using full container resources. Before deploying, booting or running the desired application, it is therefore crucial to perform the health checks on the container base image. In this work, we design an ML-based cryptomining container detection framework using syscalls as a monitoring mechanism. The cryptomining anomaly detection is based on the principle of establishing an application behavior baseline and then evaluating subsequent events against this baseline. Anything "too far" from this baseline can be regarded as anomalous and should be investigated. We use several statistical and rule-based ML algorithms, and back up their detection results with several explainability tools to investigate the cause of the ML outcomes. These ML algorithms are then compared in terms of their performance metrics.

1.1 Cryptomining Signatures

Cryptocurrency mining malware refers to software developed to use the computer's resources for cryptocurrency mining without a user's explicit permission. Attackers have attempted to profit from cryptocurrency mining by harnessing the processing power of a large numbers of computers, smartphones and other electronic devices.

The detection of cryptocurrency malware has been performed by generating its signatures in terms of power consumption, network traffic behavior, operating system processes, and patterns in hardware performance counters. In [10], an anatomy of the browser-based cryptomining is presented, in which the attacker infects a web page with JavaScript code that auto-executes when the web page is loaded by the victim's browser. The attacker takes advantage of the browser to activate the necessary JavaScript mining module. The term "illegal leverage" states that javascript is used maximally and forcibly taking full privilege without the victim's consent for a mining operation. The unauthorized execution of JavaScript can therefore be used as a signature for cryptomining malware in this scenario. If the browser behavior is measured using any profiling metrics (e.g., syscalls or processor/memory metrics like instruction per clock cycle, CPU utilization, virtual memory page faults, context switches, etc.) a definite pattern of those metrics is marked for the legitimate and healthy operation. If the browser is hijacked by the mining operation, a significant deviation is shown by such profiling metric. Such a deviation is called a mining signature of the browser. In the Windows operating system, mining is run as an executable file in memory that establishes an alteration in the system registry. In this scenario, the monitoring of registry can signal the presence of malware. Network signature extraction is also possible because mining programs contact the central botnet server to register its presence and to download relevant files depending on the architecture of the victim's system. The network transactions generate significant network traffic before the actual cryptomining begins. Tracing such traffic is relatively easy because the communication is unencrypted. In [11], the cryptominer signature is extracted in terms of power consumption for IoT devices. The energy consumption patterns of various processes on the victim's processor are monitored to differentiate malicious miners from non-malicious applications. In [12], performance counters on the victim's processor are used to assist in profiling various mining algorithms and generating their signatures. The performance counters are collected by the profiling tool *perf* and include number of page faults, executed instructions, and cache misses. Experiments with several mining algorithms result in completely different CPU/GPU signatures between mining and non-mining applications. Other findings include similar signatures for different implementations of the same mining algorithm and overlapping signatures for various miners despite implementing different proof-of-work algorithms. It is claimed in [12] that there is unique common signature among all the miners that set them apart from other healthy workloads such as SPEC CPU, CloudSuite, Parboil, and Rodinia.

1.2 Motivation

In public cloud computing services, access to the hardware resources is typically not available to the customer. Instead, the Linux-kernel system calls at the operating system level can be used as a proxy to signal the possibility of threat in a running container. The system call (syscall) is the fundamental interface between an application and the Linux kernel. A syscall is generated every time the application interacts with the Linux-kernel. Cryptominers have to repeatedly run a core Proof-of-Work (PoW) algorithm that the currency is based on [12]. Such repeated runs would result in the repeated occurrence of particular patterns for certain syscalls. System call monitoring helps to track such patterns, and any unanticipated change in the patterns of an application can signal the presence of a threat in the container. Under this scenario, an unusual syscall pattern can be used as an alert for cryptomining. Prior research that uses syscalls and behavioral models as detection mechanism include [13], [14] while research using neural network models include [15], [16], [17].

Most of the anomaly detection models are considered as black-boxes where no information is returned to the user regarding the cause of the anomaly classification. Yet, this information must be transparent to the system administrator who will need an explainable classification model in order to take the appropriate action. An explainable model generates an auditable set of explanations that describe key factors associated with the prediction. It can recommend the critical signals that need to be carefully monitored or recommend specific actions such as increasing the sampling frequency to get finer-grain details of the event that is responsible for the anomaly. It can also explain the association among the signals which are needed to manage the false prediction rate. The best use of such association rules is in fault tracing where impact at one pod might be due to some event in another pod, which cascades in some manner to a third pod, and so on. Explanations obtained from machine learning models help trace key features or sequences and eventually detect the root cause. In this work, a methodology is formulated and implemented to detect cryptominer anomalies using system calls as proxies for mining events and using an explainable machine learning (ML) model as the cryptomining detector.

2 Problem Statement and Contributions

Several methods to analyze syscall patterns are available [18], [19], [20]. A method that provides the best accuracy

across cryptominers needs to be identified along with the best possible explanation for such identification. One such method is described in [18] where a histogram of syscalls is created to find the distribution of distinct syscalls in each time window. Another method is given in [19] where the semantics of each syscall is interpreted to detect infected pods. A flow-graph analysis is used in [20] to deduce the relationships between different syscalls and generate application signatures. A Markov chain [21] is mostly used to graph the syscalls as a sequence of events in which the probability of each event depends only on the state attained in the previous event. Similarly, in [22], a weighted directed graph is built using syscalls for Android malware detection. Such graph is used as a malware signature and is compared with other container syscalls graphs to detect the anomaly. These state-of-the-art methods suffer from the following disadvantages:

1) They require a significant amount of manual intervention to analyze and interpret syscall semantics.
2) They are difficult to use in the analysis of applications that produce a large number of distinct syscalls in a small time window.
3) They are, by and large, platform-dependent. For example a change in image versions from 32-bit to 64-bit, or in the operation system type (e.g., UBUNTU DEBIAN, REDHAT) causes the prediction model to change.

We explore the use of explainable ML of syscalls to classify anomalous containers. Machine learning models have the ability to handle large data (syscalls in this case) and produce a model in comparatively smaller time than graphical methods. They remove the human intervention required in interpreting syscalls as in opcode analysis. Also, it is more effective than signature-based anomaly detection. In such detection, the "longest continuous matching syscalls sub-sequence" is extracted from all the cryptominers to get the signature of anomalous pods. Such signature extraction requires extensive computational resources. Instead of finding the exact signature of anomalies, standard MLs are capable of finding a sub-signature with comparatively less computing resources. These sub-signatures create a baseline or boundary which is sufficient to precisely classify anomalous pod behavior from normal ones. Such sub-signature is also used by explainable tools to produce a reasonable justification to support ML's classification outcome. Machine learning models are also capable of pursuing progressive learning and classification. As the platforms for a few of the containers change, the model can progressively learn the characteristics of the new version while still maintaining the classification accuracy of the older versions [23], [24], [25], [26].

To fulfill the above requirements, a methodology for anomaly detection through system calls in the Kubernetes pods is proposed, designed, and implemented as depicted in Fig. 1. Several types of cryptominer images are used in the creation of anomalous pods. Proxies based on Linux-kernel syscalls are extracted and compared against healthy applications that exhibit similar domain behavior as the cryptominers. Four different ML algorithms are used for classifying a given pod as either a crypto-hijacked or a normal pod. These algorithms are compared in terms of accuracy, runtime, and resource utilization.

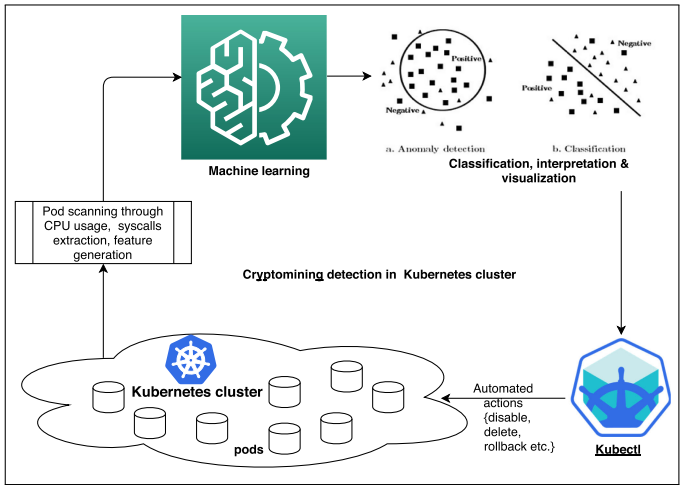


Fig. 1. Block diagram of cryptominer pod detection.

We have conducted a thorough comparison between our methodology and those in the literature that we have summarized in Table 1.

This paper makes the following specific contributions:

1) Design and implementation of a novel automated cryptomining pod detection in a Kubernetes cluster.
2) Development and implementation of real-time, syscall extraction methods for Kubernetes pods.
3) Implementation of statistical and rule-based ML models to detect anomalous pods.
4) Implementation of two statistical explainability mechanisms for ML models: one using open-source components and another with home-grown software.
5) Comparative analysis of explainable ML implementations with their differences quantified using well-defined performance metrics.

As for the organization of this paper, it is as follows. Section 3 describes a methodology where different types of cryptominers and workloads are explained along with the tools used in the experiments. The details of syscall collection in two container platforms are given in Section 4. Implementation of the proposed methodology using three types of datasets is performed in Section 5 where feature extraction, ML implementations, and their cross-validation accuracies are described. The use of explainable tools to describe the outcomes of the ML models is given in Section 6. A summary of the ML results in terms of accuracy and performance is given in tabular format in Section 7. In Section 8, a set of syscalls unique to cryptomining and normal applications are discussed to verify the ML classification operation. Finally, the paper concludes in Section 9.

3 Proposed Methodology

The setup shown in Fig. 1 has been implemented in Kubernetes. Minikube [39], [40] is used to create a single-node Kubernetes cluster on an Ubuntu 16.04 virtual machine. Eight different types of cryptomining containers are used. They are bitcoin [41], bytecoin [42], dash [43], litecoin [44], ethereum [45], zcash [46], ripple [47] and Vertcoin [48]. These miners are chosen so as to account for variety of

TABLE 1
Literature Comparison

Reference	Machine Learning	Accuracy	Explainability	Platform	ML Features	Comments
[18]	None	100%	Syscalls histogram	Docker cluster	Syscalls	Signatures are created using a histogram of different syscalls during a sliding window. Such signatures are used for detection of anomalies. This approach is neither real-time nor accurate for the cloud because the number of syscalls generated in a time window varies depending on application usage nature and traffic.
[20] ([21])	None (Random forest)	100%	Syscalls flow graph	Hardware server (None)	Syscalls (User's dataset)	A Markov chain is used to trace the occurrence of syscalls and generates an offline signature. It is rather complex for an application that generates a large number of different syscalls in a small time window.
[27]	None	100%	None	Virtual machines (VMs) and containers	Syscalls	Detection is carried out by correlating the collected syscalls with normal or anomalous sequences. It is impossible to rely on a single model. The correlation cannot be used for the causal prediction of anomalies. There is no explanation for the anomaly classification.
[28] ([29])	Random forest, decision tree, and SVM (None)	95%(100%)	Resource pattern	Supercomputer (VMs)	Hardware resource usage of CPU and Network (Memory)	Each application has a specific pattern of resource usage. Anomalies are predicted if resource usage is unusual. It is susceptible to large false positives because the traffic pattern of cloud applications is highly dynamic, causing frequent changes in the resource consumption.
[30]	None	100%	File system semantic	Docker cluster	syscalls	It protects the container by encoding/decoding application data on a per-file-descriptor basis. Threats are detected from the semantics of the file system. This method fails to detect miners because the mining processes are hidden from file monitoring tools.
[31]([32])	SVM, Random forest, Lasso, and Ridge (Random forest, Bayes network, and K-neighbor)	93% (99.97%)	None	Android, MALINE (None)	Syscalls (Usage datasets)	ML's performance is only measured in terms of preciseness. All models are used as black boxes. There is no discussion about the reason or cause of ML's abnormality prediction. Explainability through Random forest is not shown. In [32], the extraction of ML functions from raw data and the cloud implementation are not reported.
[33], { [34]}	SVM, {RNN}	99% , {96.67%}	Drebin tool, {None}	None	Uses dataset	It only uses statistical ML. One cannot rely on a single model. The extraction of ML functions from raw data and the cloud implementation are not reported.
[35] ([19])	None	100%	Instructions semantic (Syscalls semantic)	Hardware server (Docker cluster)	Wasm instructions (syscalls)	It uses classification by semantic signature matching. Signatures are collected off-line for normal applications and cryptominers. It requires an expert to analyze the various semantics manually.
{ [36]}, { [37]}	{None}, {Probability forecasting}	100%	Behavioral signature	{Docker cluster}, {Server workstation}	{Files, PIDs}, {CPU, Memory, PIDs, Network, Threads}	It uses detection through behavioral analysis. Each application has its own behavior pattern, the signature of which is extracted and compared to the non-native applications. One disadvantage is that the analysis of behavior is time-consuming.
[38]	Random forest	100%	Instructions opcode pattern	Hardware server	Assembly language instructions	Detection is based on dynamic opcode analysis of the program-generated instructions. However, most of the cryptominers remain hidden from many of the program monitoring tools. Furthermore, the detection based on a single model is not sufficiently reliable.
This work	Decision tree, ensemble learning, artificial neural network	99.2%	*SHAP*, *LIME*, and Autoencoder	Container cloud using Docker and Kubernetes with implementation scripts	Syscalls, CPU usage	We use ensemble learning, decision trees, and two architectures of neural network. We use several open-source tools to explain and justify the ML classifications. Further, we use an autoencoder neural network to extract the miner signature for ML explainability. The use of syscalls as proxies mitigates any hardware dependency. Our ML implementations are faster than the methods of semantics and opcode analysis. Details are shown in Sections 5 and 6.

mining algorithms under the detection framework of Fig. 1. For instance, they include SHA256, cryptonight, X11, Lyra2RE, and Equihash, which is meant to provide maximum coverage across the Proof of Work algorithm space as pointed out in [12] and [49]. Cryptominers are known to be CPU intensive as has been verified by inspecting the CPU usage of mining containers *(command: docker stats < container-id >)*. One method for detecting cryptominers may be based on CPU usage or usage rate. Although it is a good first-order metric, some healthy application might be CPU-intensive and show significant changes in CPU usage, creating false alerts or in the worst case, triggering the disabling of an important healthy workload. Syscalls provide a second-order metric. To enable an environment that supports CPU-intensive, healthy application pods, containers are created that are dedicated to such heavy computational loads as picture classification using deep learning [50], MySQL performance testing [51], Cassandra stress [52] workloads, Apache Spark from CloudSuite [53], map-reduce model in a single-node Hadoop cluster [54], Docker bench for security [55], a graph analytics benchmark [56], and a media streaming benchmark [57]. These workloads are selected to cover a broad range of applications, including cloud benchmarks, scientific computing, AI simulations, data mining, graph analytics, and internet search. Since we have 8 miners, we have selected only 8 benign applications to match up the number of infected and healthy containers in our framework. This is in order to avoid inference problems related to

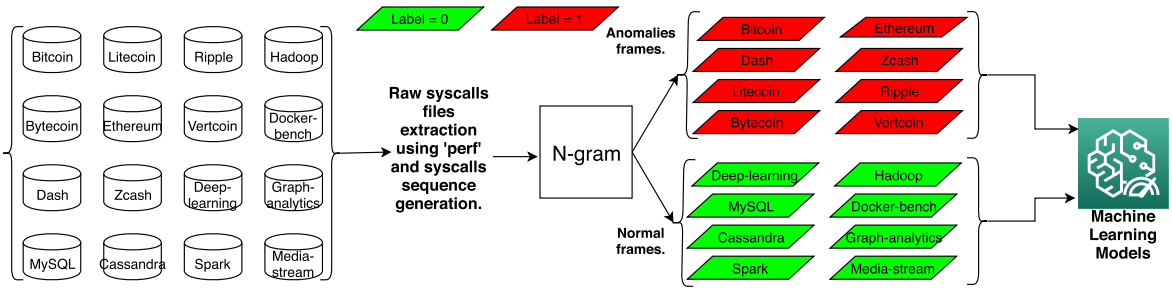


Fig. 2. Block diagram of syscalls processing using machine learnings.

data imbalances. Syscalls are collected for all these healthy application pods along with those of the cryptomining-hijacked pods.

The list of syscalls for each pod is converted into a set of frames by n-gram [58], [59]. There are other standard approaches like the principal component analysis (PCA) used in [60] to extract the ML features from syscalls sequence for malware detection. PCA is known to be computationally intensive and therefore unsuitable for real-time detection. To reduce the complexity of data formatting and feature extraction, we use the n-gram method. The n-gram frames are split into a training and validation set as per the standard ML convention (ratio of $70:30$). These frames are considered as samples while the n-grams are considered as features for the ML model inputs. The modeling setup is shown diagrammatically in Fig. 2. Four different ML methods are used: vanilla feed-forward neural network [61], feedback recurrent neural network (RNN) [62], decision tree [63], and XgBoost ensemble learning [64]. Both neural network models use statistical formulation to build the model while decision tree and XgBoost use rule-based formulation to learn the model. These MLs have been frequently used for anomaly detection (as seen in Table 1) and provide wide coverage of the machine learning methods used in practice. We have therefore used them in our cryptomining detection framework as well. Explainable model building algorithms are used so that enough information is produced by the model for effective mining malware detection. This information is required by developers and system administrators to analyze the results and get confidence into the accuracy of the ML-predicted anomaly as well as understanding of the system level rationale of pod classification. For the vanilla neural network and the decision tree model, an explanation tool *LIME Local Interpretable Model-agnostic Explanations*) [65], [66] is used. For the feedback RNN, an auotencoder [67] is used. For the ensemble learning model, the Python package *SHAP* (*SHapley Additive exPlanations*) [68], [69] is used. Such tools are open-source and have been widely used for image recognition, text identification, sentiment analysis, banking fraud and detection, among many other applications. However, they have not been used in the case of container cloud anomaly detection. To the best of our knowledge, this paper is the first to make such use of them.

As shown in Fig. 1, upon detection of a cryptomining-hijacked pod and the provision of an explanation regarding pod classification to the administrator, the methodology performs automated actions including disable, delete, rollback, and restart, based on previously specified rules. One of the possible rules may be to restart the pod first to inspect whether the mining operation is removed. If the restart doesn't help, the next action could be a rollback to use the base image of the container. A rollback failure confirms that the base image is itself infected. In this case, the action should be either a temporary disable of the pod or a permanent removal from the cluster. Likewise, several rules can be generated taking into account the cluster management constraints and business decisions. Once the application is set to be hosted in the cloud, a key problem for the application managers is how to implement and manage such rules and actions on their legacy systems in support of business operations. The use of ML inference for managing the infected containers is an important topic for future research. In this paper, we focus on the machine learning and explainability aspects of the cryptomining detection framework.

4 Syscalls Collection and Anomaly Detection Challenges

4.1 System Calls Collection for Kubernetes

The Linux performance profiling tool *perf* [70] is used to collect syscalls traces. The scripts for syscall collection on docker containers are known and easily available in the literature but little information is available regarding collection of syscalls on Kubernetes pods. In this paper, we explain the steps and scripts that need to be used in the Docker and Kubernetes context for syscalls collection. In the $64-bit$ Linux-kernel, there are 313 distinct syscalls made during the execution of each program [71]. The docker images of miners and applications described in Section 3 are used to create a Kubernetes-YAML file. They are deployed in a cluster using the Kubernetes API "*kubectl create -f < yaml file name >*". After deployment, "*perf*" is run to collect the raw syscalls in every time window. Syscall collection in the Docker platform is simple but in Kubernetes, extra steps have to be followed. The scripts for both platforms are shown in the appendix attached to this paper, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2020.3029088. These scripts will help anyone who wants to extract syscalls from Kubernetes pods with open-source *perf* tool. Apart from *perf*, there are several other tools such as *strace* and *ptrace* that one can use for syscalls collection using scripts similar to the *perf* ones. A standard tool *Sysdig* [72] has been consistently used by researchers to collect syscalls for container cloud. It has been primarily used in Kubernetes clusters and for microservice monitoring. However, *Sysdig* is not completely open-source as its premium version requires a licenses for installation and use. On the other hand, open-source tools such as

TABLE 2
Raw Syscalls File Size and the Generated Sequence Length

Pod name	Syscall raw file size (MB)	Syscalls length	Average interval between syscalls (s)
Bitcoin	554	2038829	0.000029
Bytecoin	134	481466	0.00012
Vertcoin	8	28850	0.00207
Dashcoin	18	61634	0.00097
Litecoin	83	290657	0.000206
Deeplearning	22	77981	0.000769
Mysql	5	14756	0.00406
Cassandra	246	910472	0.0000658
Hadoop	480	1656599	0.0000362
Graph Analytics	113	270981	0.000221

For each of the workloads, syscalls are collected with a one-minute sampling interval. The average time interval between syscalls is calculated by dividing the sampling interval with the syscalls length.

perf, *strace*, *and ptrace* are pre-installed in the Linux operating system and can be effectively used without incurring any additional costs.

4.2 Issues in Syscalls Collection

During syscalls collection, several issues are encountered, especially in relation to the performance of the detection framework when it is deployed in cloud clusters. These issues are scalability, storage overhead, runtime overhead, and security monitoring.

1) *Scalability:* Given that the proposed solution is designed for container clouds, the scalability of the monitoring approach is a potential concern. The syscalls are collected periodically in a round-robin fashion for all containers. The syscalls collection period for a given container is linearly proportional to the number of containers in the cluster. Consider there are N containers to monitor and that at each container, the syscalls are collected with a sampling interval of Δt then the period T_{pod} of syscalls polling at each pod is given by

$$T_{pod} = N\Delta t. \tag{1}$$

The polling interval at each pod increases linearly with N. Since mining is a long-term activity that is usually run for days (as opposed to short-term malware), a sampling interval of Δt of 1 minute is typically sufficient. Moreover, a polling interval of ($N\Delta t$) is also sufficient to detect mining activities. Polling policies can be designed to accelerate syscalls collection, including pod interleaving, random pod sampling, and event-driven monitoring [73].

2) *Storage Overhead:* Syscall monitoring is known to have a significant storage overhead. Many researchers have used heuristics to minimize storage overhead based on the bloom or cuckoo filters [74]. Such overhead depends on the nature of the workload. To better quantify such aspects, syscalls are collected with a sampling interval of $\Delta t = 1$ minute across legitimate and mining applications. The size of the raw syscalls file and the syscalls sequence length are shown in Table 2. The dynamic range of these two figures of merit is rather large. Specifically, Bitcoin, Bytecoin, and Cassandra have significant storage overhead. As a result, the proposed miner detection framework of Fig. 1 may end up using a significant amount of storage for certain workloads that can be accommodated using network storage resources of the cloud cluster. Storage needs may be controlled by decreasing the period Δt of syscalls sampling. Here also, dynamic policies of storage control may be defined but they are outside the scope of this paper. On a related score, the execution of the '*perf*' tool also consumes significant amounts of memory. But such memory usage can be mitigated using the *C-group* of the cluster node [75] and by the applying a dynamic resource provisioning policy to the containers [76] .

TABLE 3
Runtime Overhead in Syscalls Collection

Pod name	Workload's job completion time (s)		Overhead (%)
	Without syscalls 'T_1'	With syscalls 'T_2'	$\frac{T_2-T_1}{T_1}\times 100\%$
Docker-bench	30	31	3.33
Hadoop	81	84	3.7
Spark	18	20	11.11
Graph-analytics	933	944	1.11

3) *Runtime Overhead:* Syscalls monitoring is also known to have significant runtime overhead. The overuse of the tool *perf* can slow down process execution in any given container. The slowdown of job completion has led to limiting syscalls extraction to a smaller subset. Another control knob is to increase the monitoring span ($N\Delta t$). To show these aspects quantitatively, workloads have been randomly selected and their job completion times measured twice, once without collecting syscalls and once with syscalls collection. The differences in runtime and overhead percentage are shown in Table 3. In our implementation, the runtime overhead with syscalls is approximately 5 percent. Given the advantages of miner detection in the container cloud, we believe such overhead to be reasonable.

4) *Security Monitoring:* The container monitoring tool *perf* is run on the host server where all the containers are deployed. It should be noted that there can be more than one host server in the cloud cluster. Container security can be compromised in several ways, including opening to any internet traffic, having more than one port number for traffic feed, having no access credentials or compromising such credentials. If access credentials are compromised, an attacker can build a tunnel path from one container to another and reach up to the host server where *perf* is running. An attacker can then run malicious code on the host server to kill the *perf* process itself. With this, no syscalls will be generated any more for any of the containers. But as per our setup in Fig. 1, syscalls are being collected with a fixed sampling

interval across all containers in the cluster in round-robin fashion. Within a sampling period, if there are no syscalls generated then no data is passed to the ML model inference engine. The ML model would be idle, which would raise an alert to the system administrator for the inspection of containers. The fact that none of the containers generated syscalls, even though they are all active and running applications, is deemed a confirmation that the monitoring system is faulty. An alert for such a hijack of the monitoring system should therefore be added in the "automated actions" list of Fig. 1. The details of the container management policies as a result of this alert should be mapped while taking into account the cluster management constraints and business decisions as discussed in Section 3.

4.3 Anomaly Detection Challenges in Cloud

Anomaly detection is an effective way of helping administrators of cloud platforms to monitor, analyze, and improve cloud behavior. However, given their large-scale, complex architectures and their design for resource sharing, cloud platforms pose severe challenges to accurate anomaly detection. Here, we elaborate on such challenges in the specific context of virtual-machine and container clouds.

1) Virtual-machine clouds: Virtual machines (VMs) ensure security mainly by isolation but at the cost of running a complete operating system (OS). Since VMs are bulky, more than one core application is run in a VM to lower storage and compute costs. Security enhancements such as network firewalls and smaller VM images are difficult to implement, given the need to take into account the superset of all application activities. With the broader set of VM core applications, the anomaly detection framework needs complex machine learning with a wide spectrum of feature sets. Such a complex model needs time-consuming tuning for its large set of hyper-parameters, which results in a comparatively longer time in the training process. Also, such a model may be susceptible to a high rate of false positives.
2) Container clouds: Unlike VMs, containers are light-weight objects, having ample resources to run a single core program. Compared to VM, containers do not have to run a complete OS, and therefore the container image is based on a much smaller collection of packages and binaries. At a small scale where only few containers are to be monitored with each running a reduced number of known binaries and packages, the anomaly can be detected by simply verifying that only previously existing binaries are executed in those containers. To capture the activities of few applications per container, the ML model can be made simple with fewer features extracted for training and detection. As a result, training and detection runtime is relatively shorter than in VMs. Because the ML model is compact, the rates of false positives and false negatives are likely to be smaller as well.

In a VM, it is typically challenging to effectively define the total set of application events. On the other hand, in a single-application container, it is possible to define a minimal set of event markers for automating interactions. Further, Kubernetes provides fine-grained controls related to service-to-service messaging. Such controls provide additional indicators to help distinguish malicious from benign behavior and detect violations security policies. In summary, container clouds are more amenable to anomaly detection than VM clouds.

5 Machine Learning Implementation

In this section, we describe the various ML models used for pod classification.

5.1 Feature Extraction

The first ML step is feature generation from raw syscall sequences. These syscall sequences are stored in a Python dictionary. For each pod, syscalls are collected for 1 minute duration in the round-robin fashion. Since $\Delta t = 1$ minute and $N = 16$ (8 miners + 8 healthy workloads), the syscalls monitoring span is $N\Delta t = 16$ minutes. The size of the raw file and the syscall sequence length for some of those pods are shown in Table 2, in which the last column also lists the average elapsed time between syscalls occurrences for each pod. Note that it is not possible to differentiate between mining and benign applications using these time intervals. If such timing information is added as a feature for ML training, it would confuse the model and result in lower detection accuracy. Furthermore, adding the timing information would extend the length of the feature vector and in turn increase the detection computational complexity. It will also increase the training time. We have therefore considered only the syscall sequence and dropped any other arguments such as timing information, stack memory address, and kernel name.

To generate ML features, the syscalls sequences are processed by the non-overlapping n-grams described in Section 3. As an example, consider the syscalls sequence $\{1, 4, 2, 6, 12, 34, 12, 65, 34, 21, 7, 3, 5, \ldots.\}$. With the non-overlapping n-gram method for $n = 5$, several frames are extracted such that $frame_1 = \{1, 4, 2, 6, 12\}$, $frame_2 = \{34, 12, 65, 34, 21\}$, $frame_3 = \{7, 3, 5, .., ..\}$ and so on. Note that these frames are different from those of the overlapping n-grams where the frames are $frame_1 = \{1, 4, 2, 6, 12\}$, $frame_2 = \{4, 2, 6, 12, 34\}$, $frame_3 = \{2, 6, 12, 34, 12\}$ and so on. The overlapping n-grams create several repeated frames across the sequences. These repetitions increase training complexity without any significant improvement in model accuracy. Further, redundancies in the dataset often result in model overfitting, longer training time, and larger memory/disk storage without any advantage in model performance. *For these reasons, non-overlapping n-grams have been used in this work.* Experiments are run with different values of n. An exemplary result for the feedback RNN model is shown in Fig. 3. For different values of n, the model performance (in terms of metrics to be discussed below) remains approximately constant between $n = 25$ and $n = 45$. The representative value $n = 35$ is therefore used in all our other experiments because it is the value of maximum recall rate over all n's. The particular behavior at $n = 50$ corresponds to the central challenge in machine learning, namely, overfitting, where the gap between the training error and

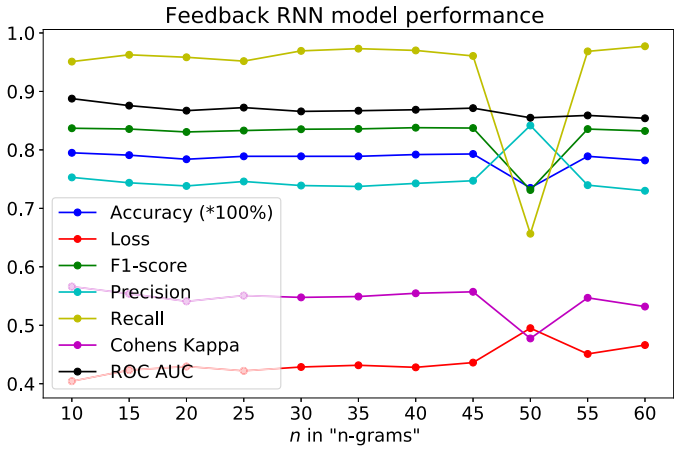


Fig. 3. Feedback RNN model evaluation on the validation set for different values of n in n-grams.

validation error is too large, which is indeed the case at $n = 50$. Over-fitting can be controlled by altering the number of neurons in the hidden layers. Models with excess neurons can over-fit by memorizing properties of the training set that do not serve them well on the validation set. The anomaly at $n = 50$ is also an example of a local maximum for the loss function, where the model has an error peak that disappears after $n = 55$.

Table 4 illustrates the differences in ML evaluation metrics between frames generated from overlapping and non-overlapping n-grams for the above RNN model where $n = 35$. As can be observed, the training time is much larger in overlapping n-grams without any significant improvement in ML evaluation metrics.

The *0* label is assigned to normal application frames and the *1* label to cryptominer frames. All the frames are concatenated into a single Python dataframe having a column size of 35. This dataframe is transformed into two matrices: an X matrix containing features and a Y matrix containing labels. These matrices are divided into training and validation sets in the ratio of $70:30$.

5.2 Machine Learning Evaluation Metrics

For the evaluation of the trained model, standard machine learning metrics are used. They are as follows:

1) *Accuracy*: This is the percentage of correctly classified dataframes in the given test dataset.
2) *Loss*: This is the average difference between the model output and the label at the end of the training phase across all training data samples. This metric is used for the statistical ML models.
3) *Precision*: This is a measure of exactness or quality of model prediction. Mathematically,

$$Precision = \frac{True\ positives}{True\ positives + False\ positives}. \tag{2}$$

The true positives are data samples that are classified as positive by the model and they actually are positive (i.e they are correctly classified). The false positives are data samples that are classified as positive by the model but they are actually negative (i.e they are incorrectly classified).

TABLE 4
Differences Between Overlapping and Non-Overlapping n-Grams for the RNN Model

Attribute	Overlapping	Non-overlapping
# of normal frames	4748978	135686
# of anomalous frames	6188383	176811
Training accuracy	81.45 %	79.99 %
Validation accuracy	80.10 %	78.90%
Loss	0.376	0.4315
F1-score	0.845	0.8358
Precision	0.776	0.7374
Recall	0.935	0.9371
Cohens κ	0.604	0.5491
ROC AUC	0.903	0.8669
Training time (s)	895	31

4) *Recall*: This is a measure of completeness or quantity of model prediction. Mathematically,

$$Recall = \frac{True\ positives}{True\ positives + False\ negatives}. \tag{3}$$

False negatives are data samples that are classified as negative by the model but they are actually positive (i.e they are incorrectly classificated).
5) *F1 score*: This is the harmonic mean of the precision and the recall. It is a metric that combines both precision and recall and provides insight into the balance between false negatives and false positives classification of the model. Mathematically,

$$F1\ score = 2 \times \frac{precision \times recall}{precision + recall}. \tag{4}$$

When false positives and false negatives are close to zero, both the precision and the recall are closer to 1 and hence the $F1score$ is 1. In the worst case, the *F1-score* is 0, which means that the model is not performing any correct classification.
6) *Cohen's kappa (κ)*: This is a statistical metric that is used to measure the observed agreement between two classifiers with respect to the case when the agreement is due to chance. This metric can effectively handle both multi-class and imbalanced class datasets. In this paper, Cohen's kappa is used to evaluate ML model performance when the number of frames for miners and normal applications are unbalanced. Mathematically,

$$\kappa = \frac{p_o - p_e}{1 - p_e}, \tag{5}$$

where, p_o is the observed agreement, and p_e is the expected agreement based on chance, which gives the performance of a classifier that simply guesses a class at random according to the frequency of each class [77] in the dataset. Values of κ equals 0 or less indicate random guessing is a better classifier than the ML model, which means that the ML model is useless.
7) *ROC AUC*: ROC stands for Receiver Operating Characteristics and AUC stands for the Area Under the

TABLE 5
Machine Learning Evaluation Summary for the Validation Set

Metrics	Decision tree	XgBoost ensemble	Feed-forward vanilla ANN	Feedback RNN
Validation (%)	97.1	89.4	79.7	78.9
Training (%)	99.6	89.3	81.1	79.99
Loss	-	-	0.3898	0.4315
Precision	0.97	0.9	0.7461	0.7374
Recall	0.97	0.89	0.9703	0.9731
F1-score	0.97	0.9	0.8405	0.8358
Cohens κ	0.9403	0.7808	0.5674	0.5491
ROC AUC	0.9701	0.8838	0.8885	0.8669

The first line in the table gives the validation accuracy while the second line gives the training accuracy. These evaluation results are for the syscalls frames collected in one time window where $\Delta t = 1$ minute and $N = 16$. For all the experiments, $n = 35$ in n-grams has been used for the feature generation. Value of loss is only available for statistical model.

Curve. The ROC is based on a cumulative probability distribution while the AUC measures the degree of separability between classes. [78]. The higher the AUC, the better the ML model is at predicting $0's$ as $0's$ and $1's$ as $1's$.

5.3 First ML Model: Decision Tree

Decision trees are commonly used for classification and regression problems. After training, the model architecture has a tree-like structure where each node represents a data feature, each link or branch represents a rule or decision, and each leaf represents an outcome (categorical label in case of classification or analog value in case of regression). For training the model with the training dataframes consisting of miners and healthy applications, the model's parameters are kept at their default values as set by the Python-SKlearn library. After training, the training data achieves a classification accuracy of 99.6 percent while the validation data achieves an accuracy of 97.1 percent. The model evaluation details are shown in Table 5.

5.4 Second ML Model: Ensemble Learning

Statically-tuned Ensemble Machine Learning (EML) is known to provide better predictive accuracy than a single learning model [79]. In EML, several base learners are built, each of which is given a piece of data and the final result of the model is computed by the ensemble of the outcome of each base learner. During ensemble process, different data subsets are drawn from the training set and each subset is used to train a different base learner classifier. Each of the base learners uses different sets of parameters to create different learning rules over the training subset, thus leading to learning ensembles. AdaBoost and XgBoost are known to be powerful EML models. In this paper, XgBoost is used where the Python parent library provides independent *XgBoost* installable EML modules [80]. Similar to the decision tree model, during training experimentation, the EML model parameters are kept at their default values as set by the Python-XgBoost library. The classification accuracy obtained with this model is 89.3 percent for the training set and 89.4 percent for the validation set. The model evaluation details are shown in Table 5.

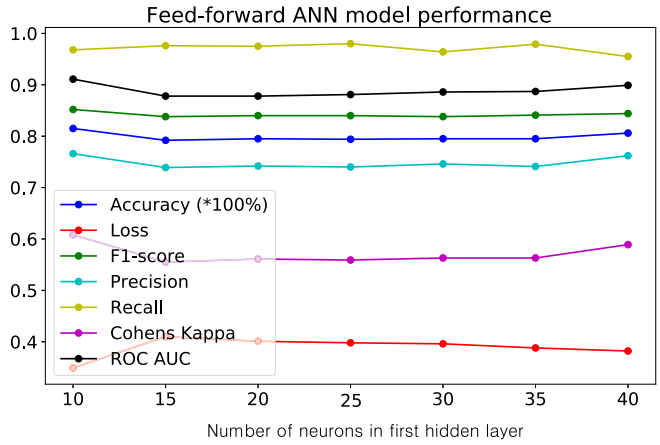


Fig. 4. Vanilla ANN model performance on the validation set versus the number of neurons in the first hidden layer.

5.5 Third ML Model: Feed-Forward Vanilla Artificial Neural Network

The first Artificial neural network (ANN) model has been created with feed-forward vanilla architecture. Feed-forward ANN is a network that has no loops (output is inserted into the input as feedback). The information flows in a simple forward direction. There can be either shallow hidden layers or deep layers. This work uses shallow layers. The main advantage of ANN, in general, is that they provide many hyper-parameters that can be tuned to build an accurate model. We have performed several experiments using $Python - Keras$ and $TensorFlow$ to arrive at the right set of hyper-parameters and obtain a training accuracy of 81.1 percent and a validation accuracy of 79.7 percent with other evaluation metrics shown in Table 5. The hyper-parameter values are as follows: Number of hidden layers = 2, Number of units at input = 35, Number of units in the first and second hidden layers = 30 and 10 respectively, Initialization = Uniform, Optimization = Adam, Activation for hidden layer = Relu, Activation for output = Sigmoid. We have used an open-source neural architecture search (NAS) tool called *autokeras* [https://autokeras.com/] to determine the ANN hyper-parameters. Autokeras produces the numbers of hidden layers and nodes along with several other hyper-parameters that correspond to the most accurate model within the given ranges. This model has been further engineered by tuning the number of neurons in each hidden layer separately. The results are shown in Figs. 4 and 5. For each plot, the values of the other hyper-parameters are kept constant while tuning the number of neurons. There is hardly any noticeable change in accuracy and other evaluation metrics. The ANN model has always performed more poorly than the rule-based decision tree and the XgBoost ensemble learning.

5.6 Fourth ML Model: Feedback Recurrent Neural Network

Given the dynamic, time-series nature of syscalls, it is natural to consider Recurrent Neural Networks [62] as a learning framework, and more specifically their embodiments in Long Short-Term Memory (LTSM) cells [81]. Indeed, since the syscalls sequence is time-series data, it is obvious to use the LSTM RNN model. The repeated run of different proof-of-Work algorithms in miners create several syscalls patterns

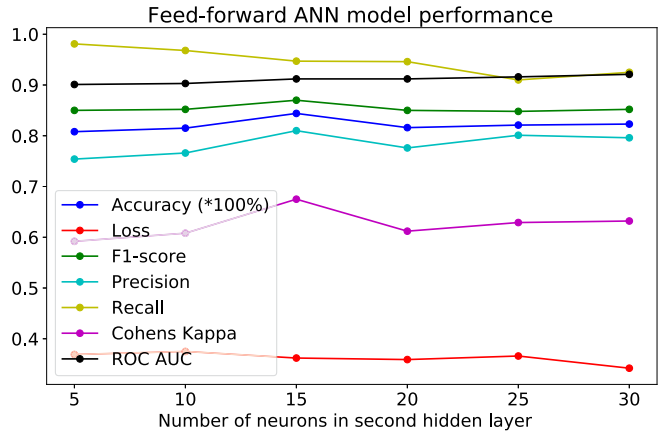


Fig. 5. Vanilla ANN model performance on the validation set versus the number of neurons in the second hidden layer.

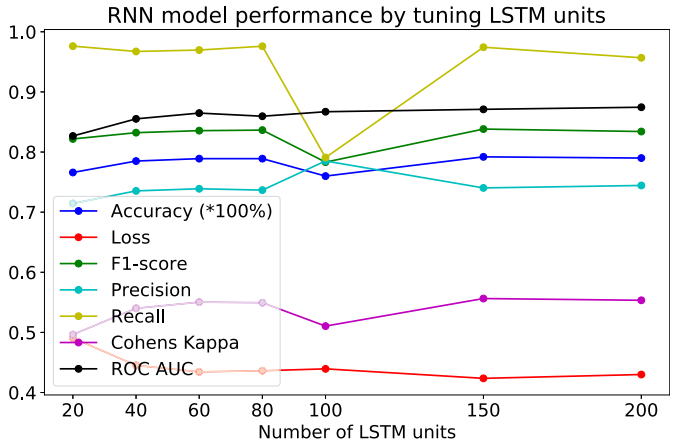


Fig. 6. Feedback RNN model performance on the validation set with changing the number of LSTM units in hidden layer.

whose relationships over time can be extracted using the LSTM RNN model. The results of applying LSTM RNN to the cryptomining problem are shown in Table 5. The training accuracy is 79.99 percent while the validation accuracy is 78.90 percent. The hyper-parameter values are as follows: Number of hidden layers = 1, Number of units at input = 35, Number of LSTM units in the hidden layers = 80, Initialization = Uniform, Optimization = Adam, Activation for LSTM layer = ReLU, Activation for output = Sigmoid. Autokeras does not support RNN. So, we used our own search strategy through a custom script. We built several models sequentially by sweeping the number of nodes, hidden layers, activation functions, and optimizers and selected the one that produces the best value of evaluation metrics. There are standard tools similar to *autokeras* like Google's AutoML and Raytune that performs hyper-parameters tuning and architecture search. For this work, we have used our own scripts with the intent of devoting another publication to the exploration of, and comparisons with, other architecture search tools. Fig. 6 shows the performance of the LSTM model when the number of LSTM units in the hidden layer is swept from 20 to 200 units. There is hardly any noticeable change in the accuracy and other evaluation metrics for this hyper-parameter over the sweeping range. The conclusion is that the LSTM RNN, like the vanilla ANN model, performs poorly in comparison to the rule-based decision tree and XgBoost ensemble learning models.

6 Machine Learning Explainability

In this section, we describe the various tools used to derive rational explanations of the ML outcomes. In the context of this work, explainability is needed to answer such questions as: Why did the ML classify a particular pod as a miner? How does the syscalls sequence change from one pod to another? Which feature has the greatest impact on miner prediction? Is there any way to visualize the ML outcome apart from plotting the evaluation metrics? The explainability tools that we will describe below represent an attempt at addressing some of these questions.

6.1 *SHAP* for XgBoost

SHAP [68], [69] is a software tool to explain the output of any ML model. In this work, *SHAP* is used to explain the classification result of XgBoost EML. *SHAP* uses the "additive feature attribution methods" where the explanation is expressed as a linear regression of the feature indicator functions. *SHAP* converts the original ML features (x_i) into binary variables (z_i) to indicate whether feature x_i is present in the input dataset or not. Such $z_i's$ form the interpretable dataset of the original features $x_i's$. Mathematically,

$$g(z) = \phi_0 + \sum_{i=1}^{N} \phi_i z_i, \tag{6}$$

where, N is the training samples size, $g(z)$ is a surrogate model of the ML model loss function, $f(x)$ and is used for explanation, and the ϕ_i's are the regression weights reflecting the contributions of the features to the output. Unlike traditional least-square linear regression, the ϕ_i's are computed using the game-theoretical Shapley values [82].

Graphically, *SHAP* produces a plot indicating how the training dataset features drive the ML model output away from a base value. The plot further quantifies the relationships between the dataset features and the ML output for each training sample. The base value is the output of $g(z)$, which is typically the average of the model output over the entire training dataset. The plots of three samples, two for the cryptominer and one for the normal application, are shown in Fig. 7 where Fn denotes the feature number within the dataset (note that from the Section 5.1, the dataset feature size is 35). A model output is labeled 1 (cryptominer) when the $g(z)$ is higher than the base value and 0 (normal) when it is lower. A set of features that push the $g(z)$ outcome higher than the base value are shown in red while those push towards lower value is shown in blue. To illustrate the clear difference between a cryptominer and a normal application classifications, 5000 samples are randomly chosen from the Vertcoin cryptominer and the MySQL application syscalls frame. Plots similar to Fig. 7 are generated for all the 5000 samples, stacked horizontally, and rotated 90^0 clockwise to reveal the explanation. The resulting graphs for the Vertcoin and the MySQL are shown in Figs. 8 and 9, respectively. The intersection between the red and blue regions lies above the base value for the Vertcoin cryptominer and below the base value for the MySQL database.

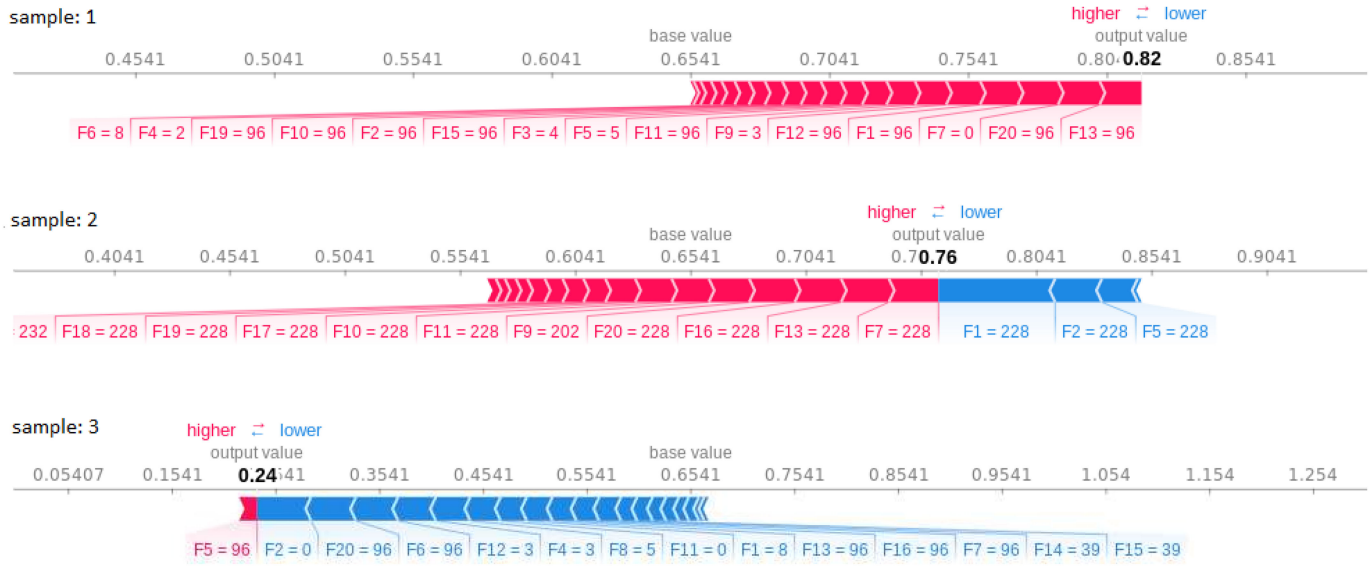


Fig. 7. Model explanation using *SHAP*.

Fig. 8. Model explanation using *SHAP* for the Vertcoin cryptominer. The data frames from 0 to 5000 are along the X-axis. Model output is along the Y-axis. The profile of output values across these frames is at the intersection of the blue and red regions. The curve is plotted in descending order of output values across 5000 frames.

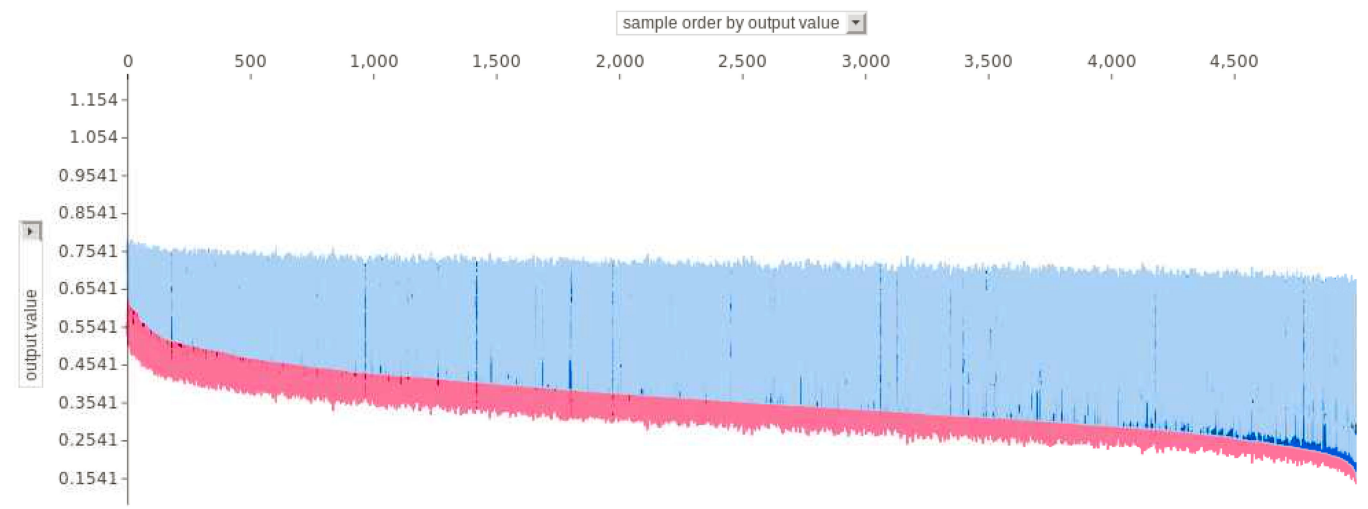


Fig. 9. Model explanation using *SHAP* for the MySQL application.

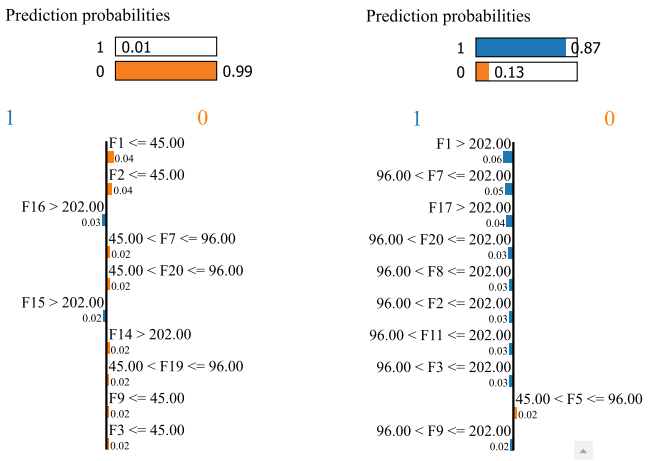


Fig. 10. Explanation for the vanilla ANN model using *LIME*. The 0 label is for normal applications, 1 for cryptominers.

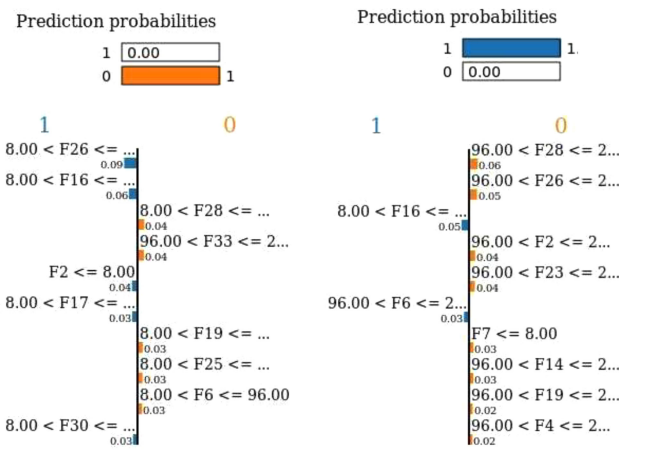


Fig. 11. Explanation of the decision tree model using *LIME*. The 0 label is for normal applications, 1 for cryptominers.

In the scenario when we are given the sequence of syscalls from an unknown pod and asked to classify the pod (normal versus cryptominer), we can proceed as follows. First the frames are extracted from the syscall sequences using the method of Section 3. Next, one of the four ML models is used for classification. Typically, a threshold of 60-70 percent is used for the number of frames with a classification above the base value to declare the entire frame sequence as cryptomining. In our work, we use the more aggressive threshold of 75 percent, which translates into a more accurate and robust prediction of the pod category. Here, for the demonstration purpose, we have displayed the use of *SHAP* for XgBoost only, but the *SHAP* explainability can also be provided for the other ML models accordingly.

6.2 *LIME* for Neural Network and Decision Tree

LIME [66] is another tool to explain ML classification. It can be applied to any ML model. *LIME* is based on perturbing the input data to the model and tracking the resulting prediction changes. Like in *SHAP*, the *LIME* requires an interpretable coding of data features using the actual features of the classification model. One possible coding for mining classification is a binary vector indicating the presence or absence of a particular syscall. Let's denote the classification ML model as $f(x)$ with the input feature vector x, which is the original representation whose prediction outcome needs to be explained. The x is associated with a binary interpretation vector x', whose components indicate the presence or absence of a features in x. The explanation vector x' is associated the explainable model $g(x')$ whose value is also binary. The model g is built using uniform sampling from a neighborhood of x'. The sampled vector is denoted z' and can be considered a perturbation of the interpretable variable x' . The localized model $g(z')$ is built to recover the vector in the original representation z and obtain the label $f(z)$ for the explanation. Let the $\pi_x(z)$ be a proximity distance between the original representation x and z and let $L(f, g, \pi_x)$ denote the loss function for using g to approximate f. One possible expression for the loss function is [66],

$$L(f, g, \pi_x) = \sum_{z,z'} \pi_x(z)(f(z) - g(z'))^2, \tag{7}$$

where $g(z') = w_g^T z'$ is a linear regression in the interpretation vector z'. *LIME* has been extensively illustrated in [66] using image and cloud datasets. In this paper, we illustrate its uses for cryptominer anomaly detection.

LIME is available as an independent Python-installable package from https://github.com/marcotcr/lime. The code takes the classifier and data as input and returns the probability distribution of the label. We apply the *LIME* approach to the feed-forward ANN and the decision tree model. The Python library *Scikit-learn* is used to build and train a vanilla multi-layer perceptron ANN and a decision tree model as per the details given in Section 5.5. The prediction results are fed into the *LIME* tool to deduce explanations. Using the feature values for a particular record, *LIME's* probabilistic algorithm computes the class probabilities and selects for interpretation of the class with the highest probability. The plots for the cryptominer and the normal application classifications are shown in Fig. 10 for the ANN model and Fig. 11 for the decision-tree model. The features in blue represent the contribution to the prediction probability of the 0 label (normal application) and those in orange represent the 1 label (cryptomining). In the plot, only the top ten features are shown.

Note that the explainability value is solely dependent on machine learning outcomes. When the same input data sample is fed into the machine learning model, both *SHAP* and *LIME* give the same explainability result. A promising approach that is specific to explainability in a neural network model is to use node sensitivity and hidden-layer relevance criteria [83]. The approach has been implemented for image datasets using a layer-wise relevance propagation algorithm (see www.heatmapping.org) where a heat map is generated to explain the classification of certain image categories. We are currently investigating the use of such method in the context of anomaly detection.

For the decision tree model, the explanation of the classification result is mostly obtained by visualizing the binary-tree structure of the model. In such tree, each node splits the incoming data according to the values of some feature variable. The decision tree is widely adopted as an explainable model because the tree is easily interpretable. This is especially useful when the feature space is different from the raw data space. The tree visualization tool Graphviz [84] is

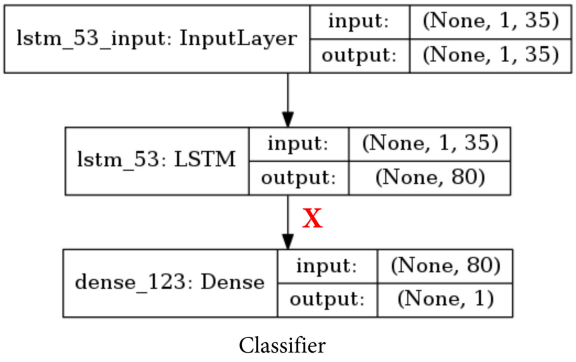


Fig. 12. Structure of LSTM RNN classifier.

used to generate such a graphical representation of decision-making splits. However, since this work uses the n-gram technique to generate frames of syscalls sequence, all the ML features are themselves syscalls. Therefore a graphic representation of the decision tree has little added value. Such a representation is meaningful in case of anomaly detection in cyber-security where network traces are used, and each ML feature has its own definition and relevance in the networking domain. Justification of the ML result is obtained by tracing the leaves and nodes of the decision tree through such extracted networking features.

6.3 *Autoencoder* for Feedback Recurrent Neural Network

An autoencoder [85] is a generative neural network model that learns a representation (or encoding) of the dataset in unsupervised learning fashion. Encoded data are fed to the decoder to recreate the original input dataset. The encoder and decoder together create autoencoders where back-propagation is applied to tune the weights such that the target value is equal to the inputs. In neural-network classification, an autoencoder can be thought of as the part of the hidden layer up to the input layer (generally the midpoint of the model), from which the reconstruction of the input data is possible. The output of this hidden layer is the compressed representation of the input. In this work, we have used an LSTM RNN for the encoder and decoder to generate model explainability for the RNN of Section 5.6. Probing the output from the hidden layer of the RNN creates the encoder part shown in Fig. 12. The RNN structure from the input to the output of the encoder, X, is the same as in the classifier as is shown in Figs. 12 and 13. Combining both of these RNN's through a repeat vector for all the 35 features so as to

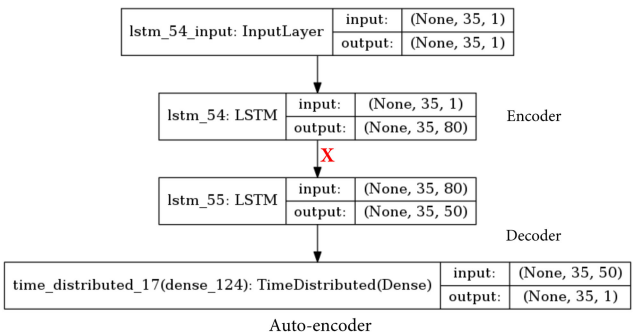


Fig. 13. Structure of LSTM autoencoder.

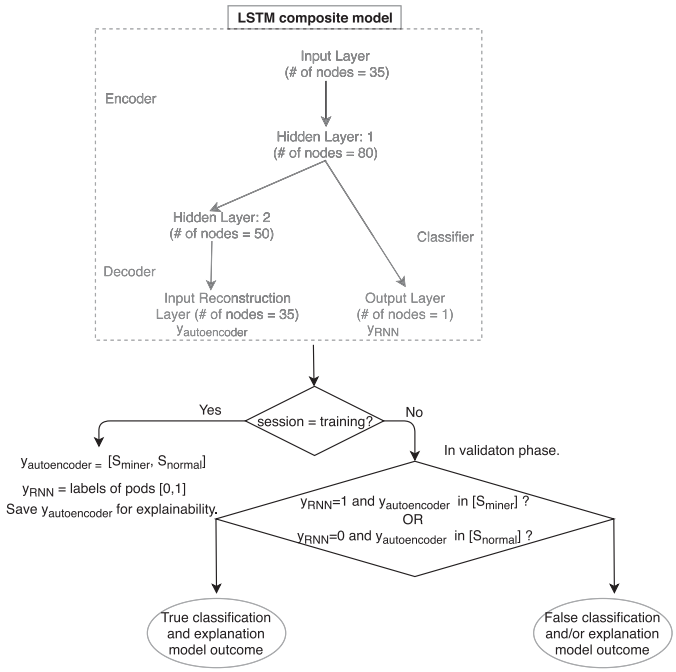


Fig. 14. Flowchart showing the classification and explainability process of the composite LSTM model.

[None, 80] with [None, 35, 80] creates a composite LSTM RNN model that has a single encoder and two decoders, one for reconstruction of the input and one for its classification.

The performance of an autoencoder model is evaluated based on the model's ability to recreate the input sequence. Validation of the autoencoder model also validates the upstream half of the classifier model which, in turn, further strengthens the trust in the classifier's outcome. The flowchart of the LSTM classification and explainability processes is shown in Fig. 14. During the training phase, the reconstructed outputs from the decoder are considered as the signatures, S_{miner} and S_{normal}, for the mining and healthy applications, respectively. Such signatures are saved for explainability. A given batch of syscalls frames from an unspecified pod is fed into the trained composite model. If the classifier predicts more than half of the frames as miners then the pod is classified as a miner, i.e., $y_{RNN} = 1$. For explainability, the output of the reconstructor denoted by $y_{autoencoder}$ is evaluated. Such output should match the miner's signature S_{miner}, i.e $S_{y_{autoencoder}} = S_{miner}$. If the result of y_{RNN} and $y_{autoencoder}$ align then the composite model is treated to be working precisely and the explanation for the miner prediction is received from the S_{miner} and $y_{autoencoder}$.

In our experiments, the composite RNN model is trained as per the following specification: Number of hidden layers = 2, Number of input units = 35, Number of LSTM units in each of the hidden layers = 80 and 50, Initialization = $Uniform$, Optimization = $Adam$, Activation = $ReLU$, Mini-batch size = 64, Epochs = 10. The autoencoder loss curve in every epoch during the training process is shown in Fig. 15. Convergence is a major issue with autoencoder design, especially when the dataset size is large, and the centroids of the various classes have significant variance. Also, when convergence is achieved, it is often the case that it is at a local minimum of the loss function. Such difficulties impact the quality of the explainability method. More details about improving the

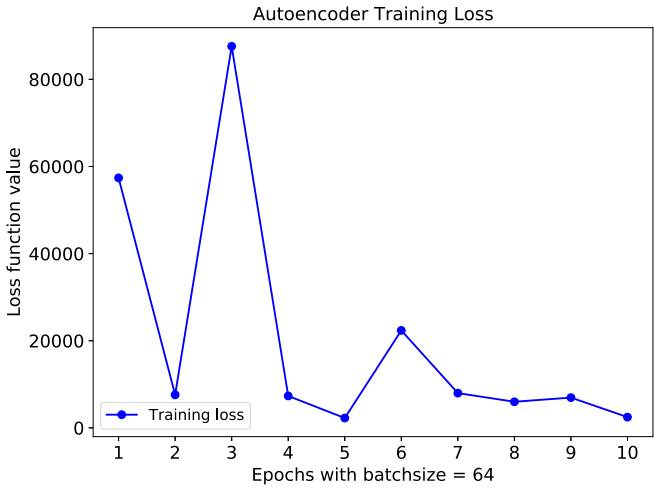


Fig. 15. Autoencoder training loss function.

autoencoder-based explainability method will be given in a future publication. Fig. 16 shows two examples of sequence reconstruction using the LSTM RNN autoencoder.

7 Machine Learning Performance Comparison

The selected ML models are compared in terms of accuracies, training-prediction time, and the compute resource usage. The results are shown in Table 6. The compute resource is measured in terms of CPU and memory. The model is built on a 4-core, memory-optimized, Amazon EC2 instance $m5d.xlarge$ [86]. CPU and memory usages are obtained using the LINUX commands *htop* and *free -m*. The commands *htop* and *free* are used to profile the code at run time and therefore their outputs include percentages of the CPU/memory overhead. However such relative overhead would be the same for all the ML models because the training and inference of all the models are performed using the very same platform, operating system, programming language, and experimental environment. The utilization is measured by sampling the CPU and memory usage every second, and taking the maximum value attained during the training phase. The table shows a CPU usage higher than 100 percent because the instance is multi-core, and the sampled value refers to the percentage usage of all the 4 CPUs, which is $4 \times 100 = 400\%$. Comparing these metrics across all the ML models, the rule-based decision tree and XgBoost models have much shorter runtimes than the ANN and RNN models. In terms of resource usage, the rule-based models are less compute-intensive because they don't require complex algorithms such as stochastic gradient descent or back-propagation to train the model but they do require extensive memory resources to save all the training rules. Further, the evaluation Table 5 suggests that rule-based models are more accurate than statistical models. As for explainability, the *SHAP* and *LIME* tools are to be recommended pending more research on the autoencoder-based explainability, especially in relation to autoencoder convergence and optimality.

```
Input sequence: [3.] [96.] [96.] [96.] [96.] [96.] [96.] [1.] [1.] [96.] [96.] [4.] [9
6.] [96.] [45.] [2.] [5.] [8.] [96.] [0.] [0.] [96.] [96.] [96.] [96.] [96.] [3.] [23.] [9
6.] [39.] [39.] [96.] [96.] [96.]
==================================================
Reconstructed sequence: [2.81] [95.84] [96.06] [95.85] [96.04] [95.96] [95.9] [96.06] [0.96]
[1.01] [95.83] [95.96] [4.] [95.84] [95.82] [45.23] [2.04] [4.93] [7.99] [95.42] [0.2] [-0.0
4] [96.] [95.85] [96.06] [96.1] [95.95] [3.05] [22.78] [95.33] [38.73] [38.81] [95.73] [95.8
8] [96.1]

 ------------------------------------
Input sequence: [232.] [5.] [1.] [40.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.] [1.]
[1.] [1.] [233.] [1.] [232.] [1.] [233.] [1.] [232.] [1.] [5.] [0.] [40.] [0.] [1.] [1.]
[1.] [1.] [1.] [1.] [1.]
==================================================
Reconstructed sequence: [231.91] [5.44] [0.86] [38.08] [1.25] [0.99] [0.74] [0.65] [0.77]
[0.83] [0.88] [0.94] [0.98] [1.01] [1.04] [1.05] [234.] [1.31] [231.5] [0.96] [230.88] [1.2
8] [231.47] [1.29] [5.64] [0.35] [40.39] [-0.44] [0.39] [0.07] [0.32] [0.31] [0.41] [0.45]
[0.53]
```

Fig. 16. Autoencoder sequence reconstruction.

TABLE 6
Machine Learning Performance Comparison

Machine Learning	Accuracy (%)	Training Time(sec)	Prediction Time(sec)	Resource Usage
Decision Tree	97.1	2.743	0.0197	CPU : 97% Mem : 243MB
XgBoost EML	89.4	18.723	0.2499	CPU : 165% Mem : 367MB
Feed-forward vanilla ANN	79.7	35.133	2.001	CPU : 335% Mem : 182MB
RNN with LSTM Autoencoder	78.9	1340	7.632	CPU : 385% Mem : 242MB

8 Syscalls Analysis and Discussion

8.1 Syscalls Analysis for Workloads

Denote the sets of unique syscalls for all miners and all normal healthy workloads by W_{miner} and W_{normal}, respectively. In this paper, these sets are the unions of the individual syscalls sets of the 8 mining applications and 8 normal applications

$$W_{miner} = W_{miner_1} \cup W_{miner_2} \cup \ldots \cup W_{miner_8} \tag{8}$$

$$W_{normal} = W_{normal_1} \cup W_{normal_2} \cup \ldots \cup W_{normal_8}. \tag{9}$$

Note that the syscalls that are in W_{miner} but not in W_{normal} provides the miner syscall signature. Similarly, the syscalls that are in W_{normal} but not in W_{miner} provides the healthy syscall signature. Using the set-theoretic difference, these syscalls signature sets are denoted as

$$Sig_{miner} = W_{miner} - W_{normal} \tag{10}$$

$$Sig_{normal} = W_{normal} - W_{miner}. \tag{11}$$

When the signature sets are non-empty, they greatly contribute to the accuracy of the machine learning models in classifying the normal and anomalous pods. On the other hand, there are syscalls that are common to both the miner and healthy application pods for which we use the notation

$$Sig_{confuse} = W_{miner} \cap W_{normal}. \tag{12}$$

Because they confuse the machine learning models. The set of syscalls that are common to all the miners captures the similarities among the proof-of-work algorithms of the miners. Similarly, the set of syscalls that are common to all the healthy application captures the similarities among the syscalls sequences of the healthy applications. For the workloads used in this paper, these "similarity" sets are denoted

$$Sim_{miner} = W_{miner_1} \cap W_{miner_2} \cap \ldots \cap W_{miner_8} \quad (13)$$

$$Sim_{normal} = W_{normal_1} \cap W_{normal_2} \cap \ldots \cap W_{normal_8}. \quad (14)$$

All the collected syscalls are processed with Python scripts to determine the above sets of syscalls. Accordingly, we get the following for the cardinalities of these sets:

$$\begin{aligned} |W_{miner}| = 100, \quad & |W_{normal}| = 116 \\ |Sig_{miner}| = 12, \quad & |Sig_{normal}| = 28 \\ |Sim_{miner}| = 7, \quad & |Sim_{normal}| = 1 \\ & |Sig_{confuse}| = 88. \end{aligned}$$

$|Sim_{normal}| = 1$ suggests that all the cloud benchmarks considered in this work are very different. On the other hand, $|Sim_{miner}| = 7$ suggests that there is similarity among the proof-of-work algorithms of the miners that helps the machine learning models achieve precise classification.

From a system administrator viewpoint, the more ML explanations, the better, especially when disruptive decisions are required such as disabling, deleting, or creating a new image. The set-theoretic analysis can provide an additional input for setting up management policies in support of the actions that are performed through the Kubernetes API *kubectl* as mentioned in Fig. 1. As examples, we list the following results for some of the syscalls sets defined above: Sig_{miner} = {288, 98, 35, 293, 263, 264, 47, 23, 25, 187, 285, 286}; Sig_{normal} = {131, 267, 140, 160, 162, 40, 43, 53, 57, 186, 62, 63, 192, 268, 204, 84, 213, 88, 90, 58, 95, 100, 229, 234, 109, 111, 112, 115}; Sim_{miner} = {1, 3, 228, 41, 42, 55, 202}, Sim_{normal} = {1}. The reader is referred to [71] for a detailed description of the syscalls.

8.2 Discussion

Based on the above results, $|Sig_{miner}| = 12$ indicates that there are 12 syscalls that are executed by the miners without appearing in any of the call sequences of the legitimate healthy workloads. Such syscalls can be used to create a *seccomp* profile [87] of the pod, which provides a control mechanism whereby a process is killed if it attempts to call one of the *seccomp* syscalls. Typically, in a container cloud, each container (or each set of similar containers in a Kubernetes pod) will have a specific *seccomp* profile, which will determine the syscalls that the application inside the container is allowed to make. For tenants that have decent security hygiene, the *seccomp* profile will limit the allowed syscalls so that only designated workloads can execute. With very high likelihood, generic malware, including cryptominers, will not be able to execute in the first place as the syscalls needed for mining will be blocked in the *seccomp* profile. As a result, cryptominers can be effectively detected through the *seccomp* policies and terminated without the need of MLs. One example of such approach is the Falco tool [88] by *Sysdig* [72] where the mining rules are written in a *yaml* file. These rules are based on the usage of a fixed set of commands and port numbers used by miners which are ultimately mapped to a subset of the syscalls. However, in a highly sensitive production environment, relying on predefined mining detection policies may not be sufficient. This is especially the case when the profiles of legitimate workloads change or new workloads are added that require syscalls from the Sig_{miner} subset. If the *seccomp* profile is used for miner detection, the healthy application might get terminated, which is very detrimental to the cloud service in the production environment. The situation is even worse when an attacker uses a new type of cryptominer corresponding to a new proof-of-work algorithm. New types of syscalls are likely to be encountered with the possibility that $W_{miner} = W_{normal}$ while $Sig_{miner} = \emptyset$, $Sig_{normal} = \emptyset$. Under such a scenario, it's the pattern of syscalls sub-sequences that differentiates the miners from the healthy applications. Such sub-sequence patterns are effectively extracted by machine learning algorithms to produce precise classification models.

In any type of anomaly detection framework, there is always the question of an adversary that makes some random syscalls to obfuscate malicious mining and throw off the classifier. It is also the case, that ML models for anomaly detection are trained under "legitimate" anomalies (e.g., well defined syscalls sequences) rather than adversarial attacks. If adversaries know that the cloud manager would be looking for a mining signature based on syscalls, they can simply introduce random syscalls in the workload, which would affect machine learning inference. To counter such adversarial attacks, it is important that training sets include randomized syscalls sequences that would mimic confusion and obfuscation attacks. During operation, the deployed ML should continuously track the syscalls sub-sequence pattern of cryptominers and continuously update the inference engine. Machine learning is capable of learning such patterns progressively over time using such algorithms as [23], [24], [25], [26] where older trained tasks of the model remain intact while learning the new ones. A few examples to counter an adversary are given in [89], [90], [91], [92] which can be applied within our framework as well.

Ideally, the data for training and testing should be sampled from the same distribution. But there could be some rare data samples that may be newly encountered for some benign or minor applications which could be at high variance with the training set. In a cloud environment, applications are dynamic, and as a result, monitoring signals have time-dependent values. In particular, some new syscalls emerge out and become highly frequent while some other syscalls that have been most active can become occasional or can disappear for a while. In Other applications, syscalls may remain steady. In such circumstances, the testing data will contain new syscalls frames which are different from those of the training set. To maintain model detection accuracy, the detection engine should be updated dynamically using a progressive learning algorithm [24], where instead of discarding the old model and training a fresh model from scratch with both old and new syscalls, only the new syscalls frames are used to update the existing model. Such learning is pursued in a comparatively shorter time than training a brand new model from scratch. Further, in progressive learning, the inference accuracy based on old training data remains preserved, which is helpful in case the application rolls back to its earlier behavior. The reader may refer to [93] where we have illustrated the use of progressive learning using the synaptic intelligence paradigm [24] for cyber-security attack classification.

9 Conclusion

In this paper, an automated pod anomaly detection setup is demonstrated in a Kubernetes cluster to detect cryptomining applications using explainable ML models. The explainability aspect is important for system administrators who must grasp the system-level rationales to support disruptive administrative decisions such as pod removal from a cluster. Several types of cryptomining algorithms may be used to launch an anomalous pod but the patterns of cryptomining system calls have common features that facilitate anomalous pod identification and discrimination against other CPU-intensive applications such as deep-learning, MySQL, Cassandra, Hydoop etc. Three explainable tools and four ML model have been implemented using syscall n-grams as data features. The syscalls frames from such n-grams achieve an aggregate anomaly prediction accuracy of more than 78 percent. Further, a comparative study of ML explainability among the four models has been performed with the tree decision model found to be the most precise achieving accuracy of more than 97 percent, *SHAP* and *LIME* are most efficient while LSTM autoencoder being least amenable to automated explanation extraction because of longer training time and convergence instability.

Acknowledgments

The first and last authors would like to thank IBM Research for hosting them at the IBM T. J. Watson Research Center, Yorktown Heights, NY, during the preparation of this manuscript. This work has been conducted under the framework of a Joint Study Agreement, No. W1463335, between IBM Research and Khalifa University, Abu Dhabi, UAE.

References

[1] Accessed: Aug. 16, 2018. [Online]. Available: https://kubernetes.io/

[2] A. S. Abed, C. Clancy, and D. S. Levy, "Intrusion detection system for applications using linux containers," in *Proc. Int. Workshop Secur. Trust Manage.*, 2015, pp. 123–135.

[3] M. Mattetti, A. Shulman-Peleg, Y. Allouche, A. Corradi, S. Dolev, and L. Foschini, "Securing the infrastructure and the workloads of linux containers," in *Proc. IEEE Conf. Commun. Netw. Secur.*, 2015, pp. 559–567.

[4] Accessed: Aug. 16, 2018. [Online]. Available: https://www.bleepingcomputer.com/news/security/17-backdoored-docker-images-removed-from-docker-hub/

[5] Accessed: Aug. 16, 2018. [Online]. Available: https://arstechnica.com/information-technology/2018/06/backdoored-images-downloaded-5-million-times-finally-removed-from-docker-hub/

[6] Accessed: Aug. 16, 2018. [Online]. Available: https://news.ycombinator.com/item?id=17309883

[7] Accessed: Jun. 07, 2020. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-aug-2019.pdf

[8] Accessed: Jun. 07, 2020. [Online]. Available: https://www.guardicore.com/2019/05/nansh0u-campaign-hackers-arsenal-grows-stronger/

[9] Accessed: Jun. 07, 2020. [Online]. Available: https://threatpost.com/threatlist-cryptominers-dominate-malware-growth-in-2018/139448/

[10] A. Zimba, Z. Wang, M. Mulenga, and N. H. Odongo, "Crypto mining attacks in information systems: An emerging threat to cyber security," *J. Comput. Inf. Syst.*, vol. 60, pp. 297–308, 2020.

[11] A. Azmoodeh, A. Dehghantanha, M. Conti, and K.-K. R. Choo, "Detecting crypto-ransomware in IoT networks based on energy consumption footprint," *J. Ambient Intell. Humanized Comput.*, vol. 9, pp. 1141–1152, 2018.

[12] R. Tahir *et al.*, "Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises," in *Proc. Int. Symp. Res. Attacks Intrusions Defenses*, 2017, pp. 287–310.

[13] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 122–132.

[14] W. Ma, P. Duan, S. Liu, G. Gu, and J.-C. Liu, "Shadow attacks: Automatically evading system-call-behavior based malware detection," *J. Comput. Virology*, vol. 8, no. 1/2, pp. 1–13, 2012.

[15] X. Xiao, Z. Wang, Q. Li, S. Xia, and Y. Jiang, "Back-propagation neural network on Markov Chains from system call sequences: A new approach for detecting android malware with system call sequences," *IET Inf. Secur.*, vol. 11, no. 1, pp. 8–15, 2016.

[16] D. Čeponis and N. Goranin, "Evaluation of deep learning methods efficiency for malicious and benign system calls classification on the AWSCTD," *Secur. Commun. Netw.*, vol. 2019, 2019, Art. no. 2317976.

[17] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, "Android malware detection based on system call sequences and LSTM," *Multimedia Tools Appl.*, vol. 78, no. 4, pp. 3979–3999, 2019.

[18] A. S. Abed, T. C. Clancy, and D. S. Levy, "Applying bag of system calls for anomalous behavior detection of applications in linux containers," in *Proc. IEEE Globecom Workshops*, 2015, pp. 1–5.

[19] H. Liang, Q. Hao, M. Li, and Y. Zhang, "Semantics-based anomaly detection of processes in linux containers," in *Proc. Int. Conf. Identification Inf. Knowl. Internet Things*, 2016, pp. 60–63.

[20] A. Desnos, E. Petrova, A. Boulgakov, R. Neal, and Z. Mithra, "Flow-graph analysis of system calls for exploit detection," *Tech. Discl. Commons*, Jun. 2018.

[21] M. Salehi and M. Amini, "Android malware detection using Markov Chain model of application behaviors in requesting system services," *CoRR*, vol. abs/1711.05731, 2017. [Online]. Available: http://arxiv.org/abs/1711.05731

[22] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4MalDroid: A deep learning framework for Android malware detection based on linux kernel system call graphs," in *Proc. IEEE/WIC/ACM Int. Conf. Web Intell. Workshops*, 2016, pp. 104–111.

[23] A. A. Rusu *et al.*, "Progressive neural networks," 2016, *arXiv:1606.04671*.

[24] F. Zenke, B. Poole, and S. Ganguli, "Continual learning through synaptic intelligence," in *Proc. Mach. Learn. Res.*, 2017, vol. 70, p. 3987.

[25] J. Kirkpatrick *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proc. Nat. Acad. Sci. United States of America*, vol. 114, pp. 3521–3526, 2017.

[26] C. Fernando *et al.*, "PathNet: Evolution channels gradient descent in super neural networks," *CoRR*, vol. abs/1701.08734, 2017. [Online]. Available: http://arxiv.org/abs/1701.08734

[27] R. Tahir, M. Caesar, A. Raza, M. Naqvi, and F. Zaffar, "An anomaly detection fabric for clouds based on collaborative VM communities," in *Proc. 17th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2017, pp. 431–441.

[28] E. Ates *et al.*, "Taxonomist: Application detection through rich monitoring data," in *Proc. Eur. Conf. Parallel Process.*, 2018, pp. 92–105.

[29] P. Dimotikalis, "Memory forensics and bitcoin mining malware," *Int. Hellenic Univ. Repository*, 2016.

[30] S. Arnautov *et al.*, "SCONE: Secure linux containers with intel SGX," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, vol. 16, pp. 689–703.

[31] M. Dimjašević, S. Atzeni, I. Ugrina, and Z. Rakamaric, "Evaluation of Android malware detection based on system calls," in *Proc. ACM Int. Workshop Secur. Privacy Analytics*, 2016, pp. 1–8.

[32] F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani, "Evaluation of machine learning classifiers for mobile malware detection," *Soft Comput.*, vol. 20, no. 1, pp. 343–357, 2016.

[33] M. Melis, D. Maiorca, B. Biggio, G. Giacinto, and F. Roli, "Explaining black-box android malware detection," in *Proc. 26th Eur. Signal Process. Conf.*, 2018, pp. 524–528.

[34] P. Mishra, K. Khurana, S. Gupta, and M. K. Sharma, "VMAnalyzer: Malware semantic analysis using integrated CNN and bi-directional LSTM for detecting VM-level attacks in cloud," in *Proc. 12th Int. Conf. Contemporary Comput.*, 2019, pp. 1–6.

[35] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "SEISMIC: Secure in-lined script monitors for interrupting cryptojacks," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2018, pp. 122–142.

[36] D. Stopel and B. Bernstein, "Runtime detection of vulnerabilities in an application layer of software containers," U.S. Patent App. 15/278,700, Apr. 6, 2017.

[37] D. Draghicescu, A. Caranica, A. Vulpe, and O. Fratu, "Cryptomining application fingerprinting method," in *Proc. Int. Conf. Commun.*, 2018, pp. 543–546.

[38] D. Carlin, P. O'Kane, S. Sezer, and J. Burgess, "Detecting cryptomining using dynamic analysis," in *Proc. 16th Annu. Conf. Privacy Secur. Trust*, 2018, pp. 1–6.

[39] Accessed: Oct. 12, 2018. [Online]. Available: https://github.com/kubernetes/minikube

[40] C. McEniry, "Kubernetes: Hit the ground running," in *Proc. USENIX Symp. Netw. Syst. Des. Implement.*, 2017.

[41] Accessed: Aug. 16, 2019. [Online]. Available: https://github.com/amacneil/docker-bitcoin

[42] Accessed: Aug. 16, 2019. [Online]. Available: https://github.com/RafalSladek/bytecoin-docker

[43] Accessed: Aug. 16, 2019. [Online]. Available: https://github.com/berrywallet/bitcore-node-dash-docker

[44] Accessed: Aug. 16, 2019. [Online]. Available: https://github.com/sreekanthgs/litecoin-docker

[45] Accessed: Aug. 16, 2019. [Online]. Available: https://github.com/ethereum/go-ethereum

[46] Accessed: Aug. 16, 2019. [Online]. Available: https://github.com/zcash/zcash

[47] Accessed: Aug. 16, 2019. [Online]. Available: https://github.com/ripplerm/ripple-wallet

[48] Accessed: Aug. 16, 2019. [Online]. Available: https://github.com/lukechilds/docker-vertcoind

[49] Accessed: Aug. 12, 2019. [Online]. Available: https://www.investopedia.com/tech/most-important-cryptocurrencies-other-than-bitcoin/

[50] F. Chollet *et al.*, "Keras: Deep learning library for theano and TensorFlow," 2015. [Online]. Available: https://keras. io/k

[51] Accessed: Oct. 15, 2018. [Online]. Available: https://github.com/geertvanheusden/mysql-stress-test

[52] Accessed: Oct. 15, 2018. [Online]. Available: https://github.com/geertvanheusden/mysql-stress-test

[53] Accessed: Aug. 16, 2019. [Online]. Available: https://github.com/parsa-epfl/cloudsuite/blob/master/docs/commons/spark.md

[54] Accessed: Aug. 16, 2019. [Online]. Available: https://github.com/big-data-europe/docker-hadoop

[55] Accessed: Aug. 16, 2019. [Online]. Available: https://github.com/docker/docker-bench-security

[56] Accessed: Aug. 16, 2019. [Online]. Available: https://github.com/parsa-epfl/cloudsuite/blob/master/docs/benchmarks/graph-analytics.md

[57] Accessed: Aug. 16, 2019. [Online]. Available: https://github.com/parsa-epfl/cloudsuite/blob/master/docs/benchmarks/media-streaming.md

[58] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, "N-gram-based detection of new malicious code," in *Proc. 28th Annu. Int. Comput. Softw. Appl. Conf.*, 2004, vol. 2, pp. 41–42.

[59] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Proc. Australasian Joint Conf. Artif. Intell.*, 2016, pp. 137–149.

[60] J. Zhang, K. Zhang, Z. Qin, H. Yin, and Q. Wu, "Sensitive system calls based packed malware variants detection using principal component initialized multilayers neural networks," *Cybersecurity*, vol. 1, no. 1, 2018, Art. no. 10.

[61] S. S. Haykin *et al.*, *Neural Networks and Learning Machines/Simon Haykin*. New York, NY, USA: Prentice Hall, 2009.

[62] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *NIPS Workshop Deep Learn.*, 2014.

[63] Y. Zhang, S. Wang, P. Phillips, and G. Ji, "Binary PSO with mutation operator for feature selection using decision tree applied to spam detection," *Knowl.-Based Syst.*, vol. 64, pp. 22–31, 2014.

[64] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 785–794.

[65] Accessed: Sep. 26, 2018. [Online]. Available: https://github.com/marcotcr/lime

[66] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should i trust you?: Explaining the predictions of any classifier," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 1135–1144.

[67] M. Baccouche, F. Mamalet, C. Wolf, C. Garcia, and A. Baskurt, "Spatio-temporal convolutional sparse auto-encoder for sequence classification," in *Proc. Brit. Mach. Vis. Conf.*, 2012, pp. 1–12.

[68] Accessed: Sep. 26, 2018. [Online]. Available: https://pypi.org/project/shap/

[69] S. M. Lundberg, G. G. Erion, and S.-I. Lee, "Consistent individualized feature attribution for tree ensembles," 2018, *arXiv:1802.03888*.

[70] A. C. De Melo, "The new linux 'perf' tools," *Slides from Linux Kongress*, vol. 18, pp. 1–42, 2010.

[71] Accessed: Oct. 12, 2018. [Online]. Available: https://filippo.io/linux-syscall-table/

[72] Accessed: Sep. 05, 2019. [Online]. Available: https://sysdig.com/

[73] M. Berekmeri, D. Serrano, S. Bouchenak, N. Marchand, and B. Robu, "Feedback autonomic provisioning for guaranteeing performance in MapReduce systems," *IEEE Trans. Cloud Comput.*, vol. 6, no. 4, pp. 1004–1016, Fourth Quarter 2018.

[74] Accessed: Aug. 12, 2019. [Online]. Available: https://medium.com/techlog/cuckoo-filter-vs-bloom-filter-from-a-gophers-perspective-94d5e6c53299

[75] Accessed: Aug. 12, 2019. [Online]. Available: https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-i-linux-control-groups-and-process

[76] S. Mousavi, A. Mosavi, A. R. Várkonyi-Kóczy, and G. Fazekas, "Dynamic resource allocation in cloud computing," *Acta Polytechnica Hungarica*, vol. 14, no. 4, pp. 83–104, 2017.

[77] S. M. Vieira, U. Kaymak, and J. M. Sousa, "Cohen's kappa coefficient as a performance measure for feature selection," in *Proc. Int. Conf. Fuzzy Syst.*, 2010, pp. 1–8.

[78] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognit.*, vol. 30, no. 7, pp. 1145–1159, 1997.

[79] H. M. Gomes, J. P. Barddal, F. Enembreck, and A. Bifet, "A survey on ensemble learning for data stream classification," *ACM Comput. Surv.*, vol. 50, no. 2, 2017, Art. no. 23.

[80] Q. Zhang, "Modern models for learning large-scale highly skewed online advertising data," Dept. Statist., 2015.

[81] Y. Wang and F. Tian, "Recurrent residual learning for sequence classification," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2016, pp. 938–943.

[82] I. Kononenko *et al.*, "An efficient explanation of individual classifications using game theory," *J. Mach. Learn. Res.*, vol. 11, no. Jan, pp. 1–18, 2010.

[83] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," 2015, *arXiv:1506.06579*.

[84] V. Ivančević, N. Igić, B. Terzić, M. Knežević, and I. Luković, "Decision trees as readable models for early childhood caries," in *Proc. Intell. Decision Technol.*, 2016, pp. 441–451.

[85] S. Amiriparian, M. Freitag, N. Cummins, and B. Schuller, "Sequence to sequence autoencoders for unsupervised representation learning from audio," in *Proc. DCASE Workshop*, 2017, pp. 17–21.

[86] Accessed: Oct. 12, 2018. [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[87] Accessed: Sep. 05, 2019. [Online]. Available: https://www.wikiwand.com/en/Seccomp

[88] Accessed: Sep. 05, 2019. [Online]. Available: https://sysdig.com/blog/detecting-cryptojacking-with-sysdigs-falco/

[89] Accessed: Sep. 05, 2019. [Online]. Available: https://resources.infosecinstitute.com/category/enterprise/threat-hunting/threat-hunting-process/threat-hunting-techniques/detecting-adversaries/#gref

[90] W. Yang, D. Kong, T. Xie, and C. A. Gunter, "Malware detection in adversarial settings: Exploiting feature evolutions and confusions in Android apps," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, 2017, pp. 288–302.

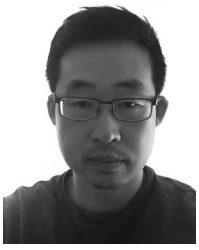
[91] O. Suciu, S. Coull, and J. Johns, "Exploring adversarial examples in malware detection," *IEEE Secur. Privacy Workshops*, pp. 8–14, May 2019, doi: 10.1109/SPW.2019.00015.

[92] A. Al-Dujaili, A. Huang, E. Hemberg, and U.-M. O'Reilly, "Adversarial deep learning for robust detection of binary encoded malware," in *Proc. IEEE Security Privacy Workshops*, 2018, pp. 76–82.

[93] R. Karn, P. Kudva, and I. Elfadel, "Criteria for learning without forgetting in artificial neural networks," in *Proc. IEEE Int. Conf. Cogn. Comput.*, 2019, pp. 90–97.

Rupesh Raj Karn received the bachelor's degree in electronics engineering from the Sardar Vallabhbhai National Institute of Technology, Surat, India, in 2011, the master's degree in microsystems engineering from the Masdar Institute of Science and Technology, Abu Dhabi, UAE, in 2015, and the PhD degree from Khalifa University, UAE, in 2019. He is a postdostcoral fellow at Khalifa University, Abu Dhabi, UAE. His doctoral thesis title was "Machine Learning Methods for the Automated Management of Cloud Computing Workloads". During his graduate work, he interned with the IBM T. J. Watson Research Center, Yorktown Heights, NY, in Summer 2018 and in Summer and Fall 2017. He received the Best Paper Award at the International Conference on Cognitive Computing (ICCC), Milan, Italy, July 2019, and was twice the recipient of a Best Paper Award at the UAE Graduate Student Research Conference (GSRC), in 2018 and 2019. His research interests include AI and machine learning, cyber security, TinyML, data analysis, cloud computing, and power and thermal modeling for servers and datacenters.

Prabhakar Kudva received the PhD degree in computer science from the University of Utah, Salt Lake City, Utah, in 1995. He is a research staff member with the IBM T. J. Watson Research, Yorktown Heights, New York, where he currently leads several projects in the areas of enterprise data centers, cloud computing for business intelligence and analytics, PaaS, and CaaS. He has received several IBM awards for high-value patents and outstanding technical achievements as well as the IEEE Region 1 Award for Outstanding Contributions to the Design Automation of Resilient Chips and Systems. He was on the adjunct faculty of Yale University, New Haven, Connecticut and Columbia University, New York.



Hai Huang received the BSE degree in computer science and engineering (CSE) from the Ohio State University, Columbus, Ohio, in 2000, and the MS and PhD degrees in computer science and engineering from the University of Michigan, Ann Arbor, Michigan, in 2006. He is currently a research staff with the Cloud Computing Department, IBM Research. His research interests include cloud computing, operating systems, distributed systems management, software testing, and anomaly detection.

Sahil Suneja received the bachelor's and master's degree in computer science and engineering from Indian Institute of Technology, Kanpur, India, in 2010, and the doctorate degree in computer science from the University of Toronto, Canada, 2016. He is a research staff member at IBM Research New York, working with the Cloud Computing team. His research interests lie broadly in the fields of AI, cloud computing, virtualization, and parallel and high performance computing. His secondary interests include wireless networking and mobile computing.

Ibrahim (Abe) M. Elfadel (Senior Member, IEEE) received the PhD degree from the Massachusetts Institute of Technology, Cambridge, Massachusetts, in 1993. He is currently a professor of electrical engineering and computer science at Khalifa University, Abu Dhabi, UAE. Between May 2014 and Jan 2019 he was the program manager of TwinLab MEMS, a joint collaboration with GLOBALFOUNDRIES and the Singapore Institute of Microelectronics on micro-electromechanical systems. Between May 2013 and May 2018, he was the founding co-director of the Abu Dhabi Center of Excellence on Energy-Efficient Electronic Systems (ACE^{4}S). Between November 2012 and October 2015, he was the founding co-director of Mubadala's TwinLab 3DSC, a joint research center on 3D integrated circuits with the Technical University of Dresden, Germany. He also headed the Masdar Institute Center for Microsystems (iMicro) from November 2013 until March 2016. From 1996 to 2010, he was with the corporate CAD organizations at IBM Research and the IBM Systems and Technology Group, Yorktown Heights, New York, where he was involved in the research, development, and deployment of CAD tools and methodologies for IBM's high-end microprocessors. His current research interests include IoT platform prototyping; energy-efficient edge and cloud computing; IoT communications; power and thermal management of multi-core processors; low-power, embedded digital-signal processing; 3D integration; and CAD for VLSI, MEMS, and Silicon Photonics. He is the recipient of six invention achievement awards, one Outstanding Technical Achievement Award and one Research Division Award, all from IBM, for his contributions in the area of VLSI CAD. He is the inventor or co-inventor of 50 issued US patents with several more pending. In 2014, he was the co-recipient of the D. O. Pederson Best Paper Award from the IEEE Transactions on Computer-Aided Design for Integrated Circuits and Systems. In 2018, he received (with Prof. Mohammed Ismail) the SRC Board of Director Special Award for "pioneering semiconductor research in Abu Dhabi." He is the lead co-editor of three books: *"3D Stacked Chips: From Emerging Processes to Heterogeneous Systems,"* Springer, 2016, *"The IoT Physical Layer: Design and Implementation,"* Springer, 2019, and *"Machine Learning in VLSI CAD,"* Springer, 2019. Between 2009 and 2013, He served as an associate editor of the *IEEE Transactions on Computer-Aided Design*. He is currently serving as an associate editor of the *IEEE Transactions on VLSI Systems* and on the editorial board of the *Microelectronics Journal* (Elsevier). He has also served on the Technical Program Committees of several leading conferences, including ISCAS, DAC, ICCAD, ASPDAC, DATE, ICCD, ICECS, and MWSCAS. He was the general co-chair of the IFIP/IEEE 25th International Conference on Very Large Scale Integration (VLSI-SoC 2017), Abu Dhabi, UAE.