A Deep Learning Approach to Distributed Anomaly Detection for Edge Computing

Okwudili M. Ezeme, Qusay H. Mahmoud and Akramul Azim
Department of Electrical, Computer and Software Engineering
Ontario Tech University
Oshawa, Ontario, Canada
Email: {mellitus.ezeme, qusay.mahmoud, akramul.azim}@ontariotechu.net

***Abstract*—One of the multiplier effects of the boom in mobile technologies ranging from cell phones to computers and wearables like smart watches is that every public and private common spaces are now dotted with Wi-Fi hotspots. These hotspots provide the convenience of accessing the internet on-the-go for either play or work. Also, with the increased automation of our daily routines by our mobile devices via a multitude of applications, our vulnerability to cyber fraud or attacks becomes higher too. Hence, the need for heightened security that is capable of detecting anomalies on-the-fly.**

However, these edge devices connected to the local area network come with diverse capabilities with varying degrees of limitations in compute and energy resources. Therefore, running a process-based anomaly detector is not given a high priority in these devices because; a) the primary functions of the applications running on the devices is not security; therefore, the device allocates much of its resources into satisfying the primary duty of the applications. b) the volume and velocity of the data are high. Therefore, in this paper, we introduce a multi-node (nodes and devices are used interchangeably in the paper) ad-hoc network that uses a novel offloading scheme to bring an online anomaly detection capability on the kernel events to the nodes in the network. We test the framework in a Wi-Fi-based ad-hoc network made up of several devices, and the results confirm our hypothesis that the scheme can reduce latency and increase the throughput of the anomaly detector, thereby making online anomaly detection in the edge possible without sacrificing the accuracy of the deep recurrent neural network.

***Index Terms*—Anomaly Detection, Machine Learning, Distributed System, Parallelism, Edge Devices**

I. Introduction

From the Internet of Things (IoT) to the Internet of Everything (IoE), the common denominator is *connectivity* driven by ubiquitous mobile devices. The goal of these devices and infrastructures is to build a cyber-physical framework that predicts and automates the mundane, enabling people to concentrate on more productive and creative things [1]. This digital infrastructure puts sensors on everything (animate and inanimate), links the sensors, does data analysis on the information generated by these *things* to create a knowledge-base, makes predictions based on the available information and if it detects an anomaly, may take corrective, preventive or stabilizing action on the system. In this way, it produces a new utility in the form of time which is a positive aspect. On the downside, this increased connectivity era implies that an automated system handles both our safety and non-safety critical data and actions, and a breach in the intended behavior of the connected devices could prove catastrophic as they manage our daily activities from medicine to autonomous vehicles, avionics and power systems. For edge devices which are characterized by their bespoke nature and constrained compute and energy resources, this era of increased connectivity increases their vulnerability. Hence, the need to run other security and safety control modules to maintain the integrity of their operations in addition to the primary functional requirement, especially devices that manage the routines of people living with disabilities. This additional requirement means that these edge devices have to manage their available resources to satisfy both the primary objective and the anomaly model. While some of the devices may not require online anomaly monitoring due to the low-risk level associated with their operations, some others that perform critical tasks within a constrained time window require a constant update on the integrity of its operation. Hence, the need for an online anomaly model that can be integrated into these devices to monitor the conformity of the behavior of the applications installed on the devices using the prescribed operational standards. To optimize the impact of the anomaly framework on the performance of the device to its primary objective, we introduce a task offloading mechanism that leverages any available resources within the network created by a Wi-Fi or Ethernet hotspot. The offloading ensures that the connected devices satisfy both the demands of the main application and that of the anomaly framework.

If the behavior of the edge devices and their applications are well-characterized, then the state transition checks espoused by the authors in [2], [3] can be applied to detect the anomalies. However, there are two challenges associated with this method of anomaly detection in this era of data and connectivity explosion; **a)** it is daunting to define all the possible states of the tuples associated with a particular process running in the embedded system because of the increasing complexity of tasks performed by these processes. **b)** the increasing complexity of the functions performed by these cyber-physical embedded systems demands a high level of dynamism which makes the use of static state transition analysis untenable. Therefore, we propose a *modular, distributed* and *scalable* anomaly detection framework that utilizes the temporal information in the system call execution sequences to detect anomalies.

To facilitate the capture of long and short-term temporal dependencies, we use a stack of the Long Short-Term Memory

(LSTM) [4] that learns the temporal relationships amongst the kernel events while the attention layer determines in small details, the impact of each feature in one another. This strategy helps to filter out the effect of the randomness prevalent in kernel events as a result of interrupts. Our anomaly framework uses a context-aware variant of the attention mechanism which does three functions; **a)** within tuples in a window under consideration, it improves the target tuple prediction accuracy by diminishing the effect of unessential source inputs for any target output. **b)** it narrows the dimension of the hidden state output of the LSTM and introduces flexibility in handling the size of the output vector. **c)** between predictions, the attention layer controls the influence of the previous output on the next target by forming a component of the context vector that controls the alignment of the next prediction. And this helps to answer the *why* question in the computation of the prediction by giving us a view of the *features* that influence each output. The logic in our reasoning is that just like natural language, each tuple has different contexts based on usage and the effect of the present tuple $\boldsymbol{V_t}$ on next prediction $\boldsymbol{V_{t+1}}$ should be derived from a deeper and longer context than just the present tuple $\boldsymbol{V_t}$ and the present attention vector $\boldsymbol{Z_t}$ because concurrently running tasks can make the order of the traces complex to analyze. Therefore, in comparison with other design architectures, our design varies on how the context vector is constructed and used in both the attention layer and next target prediction. And this is one of the principal technical contributions of the work.

The modular design of the anomaly framework of Fig. 3 is to allow partial or full offload of its component to a peer device that augments the computational capacity of the source device. Each monitored application has its own model trained using its own events. This application or process-focused design has the advantage of providing fine-grained security details in case of an anomaly, but the resource demand quickly escalates with an increasing number of applications or process running at the same time; hence the motivation to form an ad-hoc network as explained below.

The motivations in creating the ad-hoc network are; **a)** the devices have different behavioral patterns. Some have applications that work at intervals, others have applications that only respond to the environment, etc. This means that some of these devices have their resources lying idle some of the time because of the nature of the application it is running. **b)** some devices have powerful CPUs and good enough memory size but as expected, they handle more applications than low resource devices. Therefore, these kind of devices tend to generate more traces than their peers that are reactive in nature. For example, there are smartwatches that are equiped with $\boldsymbol{2GB}$ of RAM today but the most common tasks of these devices is notifications. Therefore, it benefits both the devices with low and high capacity to join the anomaly detector ad-hoc network so as to leverage peers' resources when available. Since devices with high capacity is more likely to have more applications that need monitoring, and some low capacity devices behavior tends to be bursty, it produces a symbiotic outcome for both types of devices with diverse capacities and justifies the need to form a network.

Example of the characteristics of system calls sequence is open $\longrightarrow$ mmap $\longrightarrow$ read $\longrightarrow$ close and these are used as features to train the model for each application being monitored. Therefore, we have two problem statements as thus: **a)** given kernel events obtained during the normal operation of an application or a process, is it possible to create a model that uses the information from the normal behavior of the process to detect deviant behaviors in the kernel events? **b)** if yes, can we dynamically deploy this anomaly model to perform online anomaly detection by leveraging a pool of resources provided by an ad-hoc wireless local area network? To answer the questions posed above, we propose a *modular, distributed and scalable anomaly detection framework* primarily targeting edge devices. In summary, our contributions are: **a)** First, we present a modular deep learning anomaly detection framework that uses kernel events of processes to observe the operation of the applications on the device. **b)** Then, we show how to leverage the modular nature of the anomaly framework to create a dynamic, distributed, and scalable network of nodes working together to improve the security of all participating nodes. **c)** Finally, we test the performance of the task scheduling algorithm against the baseline of not joining the network and show why the anomaly detector ad-hoc network is a good bargain for all participating nodes in terms of throughput and improved security.

To ease the comprehension of the work, we have divided the rest of the work into the following sections; Section II briefly highlights the related work in this domain. Section III discusses the technical details of our model while Section IV details our implementation strategies and discusses the tools we use to implement the framework. In Section V, we conduct the experiments to verify our hypothesis and discuss the results. Finally, we conclude the paper in Section VI with an insight into our prospective research directions.

II. Related Work

According to [5], the two broad categories of detecting deviant behavior during system operation are *intrusion* and *anomaly* detection. While both techniques can detect previously seen anomalies, only the anomaly method monitors both known and unknown deviation from the standard performance behavior. The use of *models* instead of *signatures* enhances the capability of the anomaly method to track *zero-day* vulnerabilities. The intrusion (signature) method has extensive details in [6]. The obvious limitation of this signature-based method is that *zero-day* vulnerabilities cannot be detected as it only searches for known signatures. On the other hand, authors in [7]–[10] use the anomaly-based approach which involves the construction of a model to target both known and unknown aberrations. This model-based technique comes at the cost of doing *feature extraction* from the operational profiles, *processing* the extracted features to conform to the model input requirements, and finally, model design and training with the profile features. While this method provides versatility in terms of its target threat domain, it has higher false positives than the signature-based intrusion detection mechanisms.

In [7], the authors explored the use of a vector space model with hierarchical clustering that creates a binary profile to determine if an observed profile sequences are anomalous or not. While this model shows an excellent result in the experiments used by the authors, its scalability is limited because of the enormous number of tuples it needs to make a decision. Authors of [11] also have a vector space model based anomaly detection method which categories system processes using their system call information. This model suffers from the same scalability issue as that of [7] because it requires a long window of observation before it can make a decision. The authors of [8] built an anomaly detection framework called *Deeplog* using two layers of LSTM networks and a workflow construction approach for each key in the log for diagnostics. This *Deeplog* framework uses LSTM also, but there is no notion of attention layer in the framework. Also, the authors of [12] used the statistical metric of entropy to implement an anomaly detection model for network logs but this type of anomaly model is best suited for cases where the volume of logs determine if an anomaly has occurred as obtainable in denial of service attack. In [13], a real-time system anomaly detection model is designed using the principle of inter-arrival curves to detect anomalous traces in a log sequence. However, this inter-arrival curve-based model works offline because it requires a large number of logs before it computes the curves. Reference [9] designed a vector space model to mine console logs for anomalies and [3] used an optimization method of minimum debugging frontier sets to create a model for detection of errors/faults in software execution. In [14], [15], the authors use the kernel event tuples to design an anomaly model based on the concept of the encoder-decoder approach. It uses an attention layer to aid in sequence reconstruction at the prediction phase. Although this approach is close to our method, we differ both in the deep learning model design and the entire framework implementation. Hence, while we introduce a load migration scheme to ensure a truly dynamic and distributed anomaly framework, [14] creates a monolithic anomaly model that places a considerable strain on embedded system resources, thereby limiting its use for online anomaly detection. Also, the methods advocated in [14], [15] involves the use of KNN classifier at the anomaly decision stage which reduces the applicability of such approach in a near real-time application. Also, the authors of [16] did a host-based anomaly detection using a combination of CNN/RNN where the CNN acts as a filter and the RNN helps in remembering contexts over a log span of sequences. Furthermore, the authors of [17] used a mixture of PCA algorithm and KNN classifier to perform host-based anomaly detection on system traces. As expected of many clustering-based processes, [7], [17] suffer from being unusable for online analysis. In [18], a graph-based anomaly detection scheme based on system calls was developed. While this is lightweight and can run online, it cannot handle a previously unseen type of sequences without rebuilding the graph. Finally, [19] use the concept of sequence covering to develop an anomaly detection model using system calls as the sequence of interest.

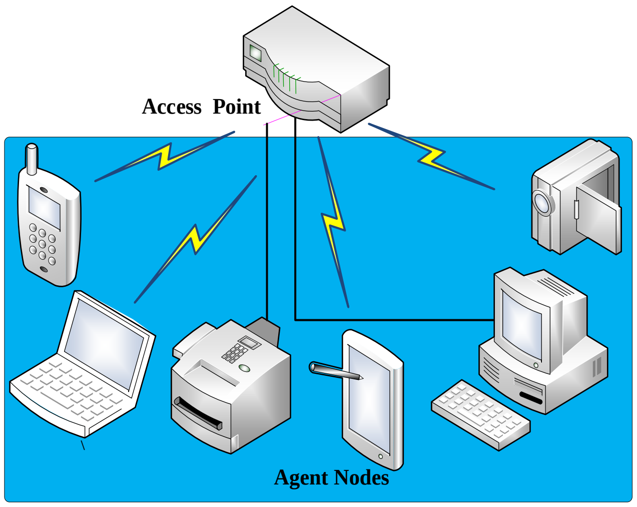


Fig. 1: Network Design with Diverse Nodes

III. Proposed Approach

A. Definition of Terms and Assumptions

The network consists of *nodes* of diverse capabilities and behavior and an *access point* like a router. While the node can access the internet and other external services via the access point, our network does not include nodes in the cloud or outside the local network. Our design brings online capability to a node which may or may not accommodate the extra strain on system resources introduced by doing anomaly detection on applications in the node. While the cloud boasts of unlimited resources, latency constraints and the fact that some of the nodes are mobile restricts our design to a local network of nodes. On the positive side, the proximity constraints provided by the access point device ensures that we have an idea of the worst latency conditions and the nodes can easily adapt their applications according to the prevailing local conditions. As the network condition readily available when the node connects to the access point, the nodes can adaptively export the anomaly application wholly or partially to other peers in the network by running its *task manager* algorithm that is embedded in the application. Overall, the network aims to optimize the processing time of tasks and reduce the number of tasks dropped for exceeding the response time constraint.

Therefore, the application aims to address the question of which part of the anomaly framework should run locally or on a peer to reduce processing time. In making this decision, the capacity of the peers, round trip time to peers, number of tasks on peers, energy level of peers, etc are taken into consideration in the task management algorithm. Hence, there is a competition between the local resources and the transmission medium. Before we introduce the model parameters, we define the following behavior of the ad-hoc network:

- every node that generates a task also acts as a sink for the task whether it was executed locally or in another peer.
- as with all wireless or wired local networks, every traffic passes through the access point. Hence, the capacity of

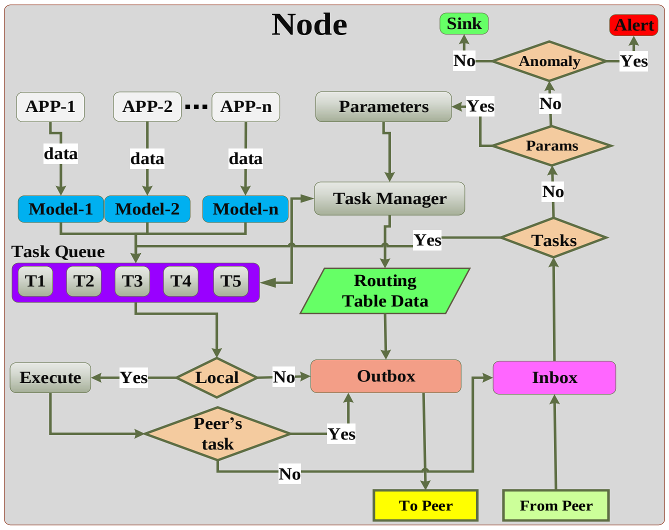


Fig. 2: Implementation Diagram of the Anomaly Detection Application in a Node

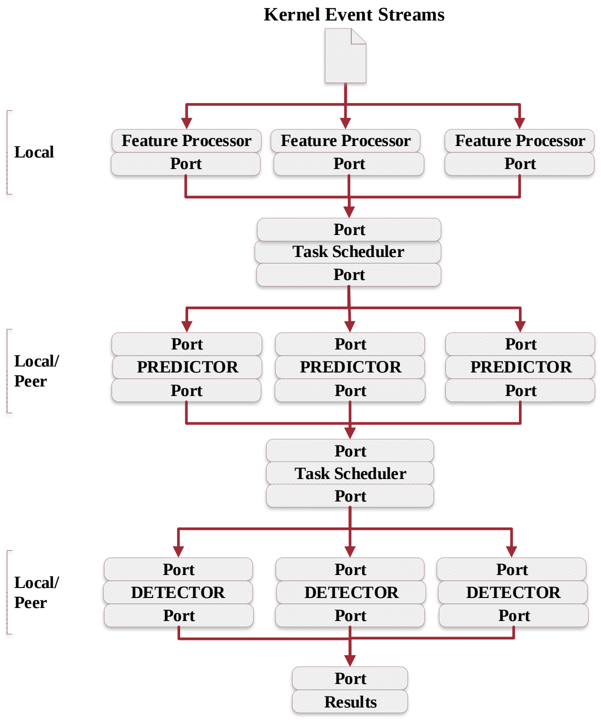


Fig. 3: Dataflow Model of the Anomaly Detection Framework

the network is limited by the capacity of the access point node. Bluetooth and other peer-to-peer technologies that do not support multi-input, multi-output paradigm are not considered because our application is multi-threaded.

- in the dataflow model of Fig. 3, the application has been split into parts with designations of possible points of execution. If a task is offloaded at the *predictor* stage, then the peer that executes the task determines where the detector runs but the result must go back to the source of the task.
- movement of peers within the network does not alter the network topology.
- nodes do not route traffic. Therefore, the ad-hoc network is a one-hop network, and the one-hop is the access point node.

B. Problem Statement

We model the execution flow of Fig. 3 with a directed acyclic graph (DAG) $G = (V, E, s_v)$ made up of a set of v task segments $V = \{v_j | 1 \leq j \leq v\}$, a set of e edges or arcs $E = \{(p, q) | p, q \in N\}$, and a set of instruction counter tag s_v attached to every task segment identifying the number of CPU instructions needed to execute the task v. The task segments enjoy the benefit of parallelism provided via threads and the number of threads that can be created on a particular device is a function of the device resources spared for anomaly detection after the main application has taken its own required resources. If a task, v is to be offloaded to a peer, *ZeroMQ* sockets are used to transfer data between peers in the network. Hence, the ports in Fig. 3 are attached to *ZeroMQ* sockets. We chose *ZeroMQ* in our design because sockets can disappear and reappear without needing to restart the pair. Hence, we can scale up or down the number of concurrent *predictors* or *detectors* at runtime so as to optimize the latency reduction. Therefore, with the knowledge of the properties of the graph G, the CPU capacity, α of the node n, the fraction of the CPU for anomaly detection, β and the capacity of the wireless channels, B, the problem becomes finding the minimum task finish time of v task segments, $V = \{v_j | 1 \leq j \leq v\}$ given a n set of peer nodes $N = \{n_p | 1 \leq p \leq n\}$. To get the peers parameters required for computing the task finish time algorithm, the nodes rely on the access point node to reduce the congestion occasioned by parameter updates. If each peer has to query each other for updates, then each node requires $2 \times n$ messages to get a view of the network. And considering that all these messages pass through the access point, $(2 \times n \times n)$ messages will have to pass through the wireless channel B for parameter update alone. We consider this a significant strain on the network considering that these updates are frequent. Hence, our decision to use the access point node for parameter update where a broadcast signal is used to query the peers and send parameter update. So, parameter update in this instance only requires $((2 \times n) + n)$ messages.

Therefore, given the instruction count, s_v of a task segment, and CPU resource of the node $\beta \times \alpha$, the compute time of task segments, $V = \{v_j | 1 \leq j \leq v\}$ in a given node, n is given in (1).

$$t_{V_n} = \sum_{j \in V} \frac{v_j \times s_{v_j}}{\beta \times \alpha} \tag{1}$$

In (1), j denotes the *predictors* and *detectors* of Fig. 3. If the size of the task to be transferred to another peer via the edge, $e_{n,m}$ is $v_{n,m}$, then the latency is computed in (2).

$$t_{V_{n,m}} = \min_{(n,m \in N)} \left(2 \times \frac{v_{n,m} \times |r_n - r_m|}{e_{n,m}} + \sum_{j \in V} \frac{s_{v_j} \times v_j \times |r_n - r_m|}{\beta \times \alpha} \right) \quad (2)$$

$$\text{where } \sum_{j \in V} v_j \equiv v_{n,m}$$

From (1) and (2), we derive the cost of processing a task as given in (3) where $r_m = 1$, and $r_n \in \{0, 1\}$. Since all the peers share the same wireless bandwidth, then $\sum_{(n,m \in E)} e_{n,m}|r_n - r_m| = B$ and $e_{n,m} > 0$ in (2). Also, in (2), we assume the same data size for both the forward and backward transmission. Simplifying (2) further, the cost of processing a task segment in a peer node becomes (3)

$$t_{V_{n,m}} = \min_{(n,m \in N)} \left(|r_n - r_m| \times \left(2 \times \frac{v_{n,m}}{e_{n,m}} + \sum_{j \in V} \frac{s_{v_j} \times v_j}{\beta \times \alpha} \right) \right) \quad (3)$$

Now, given tasks with latency requirements, t_{max}, the problem reduces to minimizing (4) such that $t^{v_j} \leq t^{v_j}_{max}, \forall j \in V$. Finally, this minimization algorithm is solved by each node independently with the help of the network information supplied by the access point node.

$$t_V = t_{V_n} + t_{V_{n,m}} \quad (4)$$

C. Nodes (Devices)

1) Task Generation: The nodes of Fig. 2 have system resources of different capacities and run diverse types of applications. However, while some of the applications tolerate delay, some others are time-sensitive. Hence, the size of the data that the processes $\{P_1, P_2, ..., P_n\}$ of Fig. 2 emit vary in size and time sensitivity. When kernel events of a process P is fed to the model of Fig. 2 as data, a *task* T_p is created and transferred to the task queue. To aid in task management and routing, each T_p has as properties; *data, task-type, task-uuid, task-owner-name, task-size*, and the *task-creation-time*. The *task-type* determines the stage of the processing (predictor or detector) while *task-uuid* identifies the source node for routing back results if the task is processed in a peer. The size of the task of a process, P in bytes is denoted as d_p while the s_p is the number of CPU instruction cycle needed for the task, T_p's execution. We assume a uniform d_p for each T_p of a unique P since the input data is structured with a known dimension.

Also, tasks can come from peers that have determined that it is cheaper (in terms of time) for their tasks to be executed in this node as shown in Fig. 2.

2) Task Manager: The task manager does the cost estimation which determines where a task is executed taking into account the frequency of task generation (both internal and external generation), the nodes profile and the parameters supplied by the access point node. Equation (2) represents the latency of executing task segments in a peers node and this

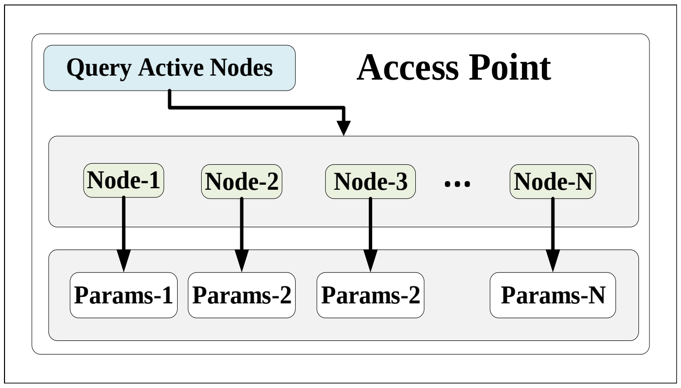


Fig. 4: Access Point Node

information is provided by the access point node which we have designed to have a complete view of the network. Since (3) is the latency of executing a task segment in a peer, (4) can be modified to become (5) which gives the end-to-end latency of a task. Also, because (5) is not trivial to compute, we use the experimental value latency obtained from the network profile parameters (which includes the delay, processing time, queue size and node CPU load information) supplied by each node to the access point node.

$$t_V = t_{V_n} + \text{latency} \quad (5)$$

Since node mobility in and out of the network is common, and some tasks can be bursty, the task manager keeps a routing table which is updated each time the access point node sends the updated network parameters to all the active nodes. This regular update prevents the sending of tasks to a stale node. The table maintains a cyclic ascending latency times from active peer nodes as well as the active peer *uuid*.

3) Handle Task: The task execution and offloading is determined by the information computed by the task manager which annotates the task segments with the point of execution. If a task segment is marked to executed locally, the *execute* module is called, otherwise, it sends the task to the *outbox* module which uses the routing table information to retrieve the peer node *uuid* and the task is dispatched. If the local execution happened on a task, the *task-uuid* is compared with the nodes *uuid* and if the *uuid* match, the execution result is sent to the nodes inbox, else that *task-uuid* is used to route the task result back to the node that created the task.

4) Inbox: The *inbox* module receives *tasks, results*, and parameter *query/update* locally, from peers and access point node, and makes the necessary check using the received data properties to take appropriate action as outlined in Fig. 2.

D. Access Point Node

Our access point node performs two functions: parameter *query* and *update* using broadcast messages. As we explained in Section III-B, this design reduces the number of messages required for update by more than $1/3$. We are aware that this makes the access point a single point of failure but since it is central to the network formation in the first place, we know that the exit of the access point node terminates the network irrespective of whether we use it for parameter update or not since the Wi-Fi technology is provided by the access point.

IV. Implementation

To ensure an easy comprehension of the concepts discussed in this section, we start by describing the deep learning framework which is at the core of the anomaly detection framework. A high-level architecture of our framework is depicted in Fig. 5. The high-level overview depicts a stream of kernel events traces from an application or a process continuously being injected into the anomaly framework. We feed the streams to the deep learning models via the *feature processor* module.

A. Feature Processor

The kernel event features of interest are the *timestamp*, and the *system call ID*. To ensure a deterministic outcome with respect to time, we use the CPU cycle count. In a given process or application being observed, given *timestamp* as $\mathbf{t}$, and system call string name as $\mathbf{k}$, we process these properties to yield a multi-input feature space that we feed to our models. These actions are captured in the *Feature Processor* module of Fig. 5. Equation (6) defines the processing of the time stamp to yield the CPU cycle count which we feed as one of the inputs to the model.

$$\boldsymbol{\beta} : (\mathbf{t}_i, \mathbf{t}_{i+1}) \longmapsto \mathbf{t}_{i+1} - \mathbf{t}_i;\ i \in \mathbb{N} \tag{6}$$

In order to get $\mathbf{S}$ in Fig. 5, we use the function defined in (7) to encode the system call ID strings into their corresponding Linux tables. There are a total of 330 unique system calls in the X_64 platform.

$$\mathbf{S} : \mathbf{k} \longmapsto \mathbf{w};\ \text{where } \mathbf{w} = \{w \in \mathbb{N} \mid 1 \leq w \leq 330\} \tag{7}$$

B. Predictor

The predictor of Fig. 5 is designed using LSTM layers and a dense neural network pair with an interleaving layer of an attention layer that helps introduce flexibility in handling the shape of the data, and also performs filtering operation to ensure high fidelity replication of normal execution sequence of a monitored process or application. The predictor consisting of the LSTM layers and the attention layer are shown in Fig. 5

C. Detector

1) Error Estimator: Given $\boldsymbol{x}_i \in \mathbb{R}$ which serves as the input, the target during learning is a shifted version $\boldsymbol{x}_{i+w} \in \mathbb{R}$ where $\boldsymbol{w}$ is the lookahead window. Therefore, the goal is to replicate the sequence at the output by creating $\boldsymbol{x}' \in \mathbb{R}$. Hence, the perfect result is when $\boldsymbol{x}_k \equiv \boldsymbol{x}'_k$, but this is hardly feasible because of the high randomness caused by interrupts and other events in the traces. Hence, when we have $\boldsymbol{f} : \boldsymbol{x} \longmapsto \boldsymbol{x}'$ given $\boldsymbol{x}$ as the ground truths, the deviation $\boldsymbol{d} = |\boldsymbol{x} - \boldsymbol{x}'|$ is the difference between the ground truth and the predicted value for the given sequence. This deviation becomes the prediction error values which we process further to decide if an anomaly has occurred or not. Our framework is a *multi-input, multi-output* (MIMO) architecture. Hence, the prediction

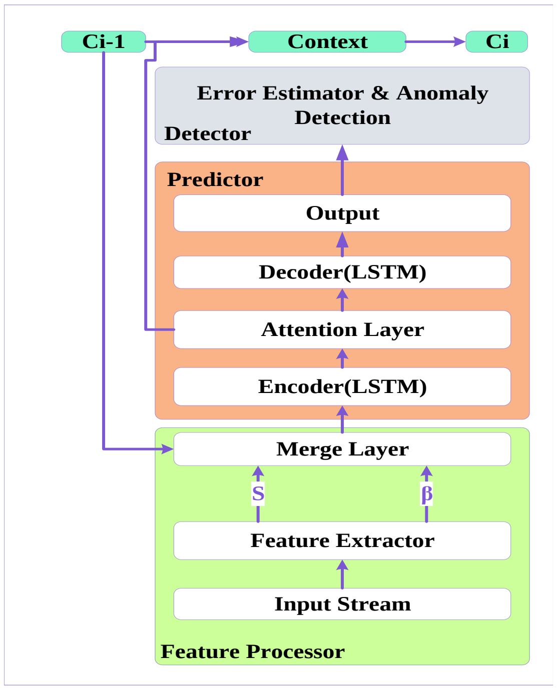


Fig. 5: Deep Anomaly Model View with the Feature Processor, Predictor and Detector

error for the categorical output is the *Boolean error* between the predicted and the truth value. While for the regression output, the prediction error is the absolute difference between the ground truth and the outcome of the prediction.

2) Anomaly Detector: The deviations $\boldsymbol{d}_v = \left\{\boldsymbol{d}_1, \boldsymbol{d}_2, \boldsymbol{d}_3, ..., \boldsymbol{d}_{|V_j|}\right\}$ from the validation dataset constitute random variables which we have no knowledge of the underlying distribution. However, since they are error values, we are interested in creating an upper bound or threshold to detect anomalies. Hence, we try to create a threshold value for the prediction errors. Because we know the sample $\boldsymbol{\mu}$ and variance $\boldsymbol{\sigma}^2$, we employ the *Bienaymé-Chebyshev* inequality to compute the threshold. The inequality guarantees that no more than a certain fraction of values can be more that a certain distance from the $\boldsymbol{\mu}$. The inequality is stated mathematically in (8).

$$\boldsymbol{P}\left(|\boldsymbol{X} - \boldsymbol{\mu}| \geq \boldsymbol{\Upsilon}\right) \leq \frac{\boldsymbol{\sigma}^2}{\boldsymbol{\Upsilon}^2} \tag{8}$$

Although the inequality computes the absolute bound with the assumption of a symmetric distribution, we use it in our model because we are only interested in the range where the prediction error $\boldsymbol{d} - \boldsymbol{\mu} > 1$. Hence, lack of symmetry will not affect the threshold computation. Therefore, we interchange $\boldsymbol{\Upsilon}$ in (8) with $|\boldsymbol{d} - \boldsymbol{\mu}|$ to derive (9) which we use to compute the decreasing probability for increasing deviation $|\boldsymbol{d} - \boldsymbol{\mu}|$ where $\boldsymbol{d} > \boldsymbol{\mu}$.

$$\boldsymbol{P}\left(\boldsymbol{d} > \boldsymbol{\mu}\right) = \boldsymbol{P}\left(|\boldsymbol{X} - \boldsymbol{\mu}| \geq |\boldsymbol{d} - \boldsymbol{\mu}|\right) = \frac{\boldsymbol{\sigma}^2}{(\boldsymbol{d} - \boldsymbol{\mu})^2} \tag{9}$$

One of the advantages that this technique provides is that we do not have to know the other parameters of the underlying probability distribution. Also, instead of creating a binary threshold of *True* or *False* values as was the case in [15], this probability gives us an opportunity to quantize the anomaly scores into bands per one complete cycle of operation like the safety integrity level (*SIL*) provided by safety standards like *IEC 61508* [20]. Our quantized levels are called *anomaly level probability*, ($\boldsymbol{AL_\rho}$) and each level depends on the value of the probability from (9). As (9) measures how the error values are clustered around the mean, our framework should ideally create a high fidelity predictions (reconstruction of the normal profile sequence) to ensure that (9) performs optimally and reduces false negatives.

D. Networking

As seen in Fig. 3, the dataflow model entails a lot of networking between threads, processes and nodes. In all, we implemented the networking framework using ZeroMQ [21]. Three prominent features of ZeroMQ that endeared us to the library are: **a)** platform agnosticism. **b)** inherently asynchronous by design. **c)** arbitrary connection and disconnection capability without resetting the network, and this aided our parallelism design. Hence, the ports in the dataflow model of Fig. 3 are all binding to ZeroMQ sockets.

V. Experiments and Results

A. Experiments

We setup an experiment with $\mathbf{10}$ nodes that can go on and off the network at anytime. The nodes are of diverse compute capacities ranging from personal computers to Raspberry Pi. Some nodes have a continuous stream of tasks while other nodes have tasks that are emitted intermittently or randomly. In each of the devices, we select the processes to be monitored and the higher the number of processes monitored, the higher the load on the CPU. The tasks are mixture of high and low priority segments where priority is determined by the latency requirement of the task segment. Our experimental objectives are: **a)** determine the throughput (task completion time) of tasks in an independent node as the CPU load varies and use that as the baseline. **b)** determine the throughput of nodes when the nodes in a Wi-Fi form an ad-hoc network and use the offloading algorithm for managing task distribution. **c)** and we show the accuracy of the deep learning model in capturing the anomalies in the system calls. We start the access point node since it provides the interface for networking and we then start the devices. For the purpose of the simulation, the access point node notifies each node to disconnect when $\mathbf{2}$ million tasks have been completed and we show the statistics in Section V-B. The nodes stream data logged from many processes (normal and anomalous) of an unmanned area vehicle application to imitate individual applications in a node.

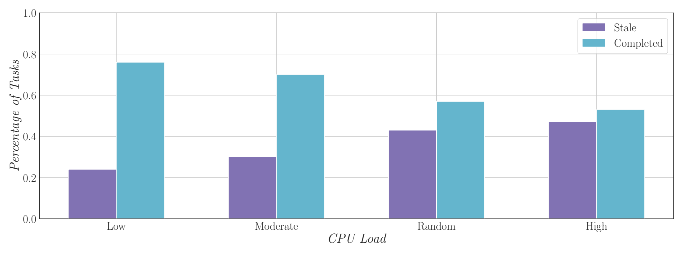
B. Results

The data represents three profiles: *normal* (no anomalies in the data), *delay* (denial of service kind of anomalies) and *random* (information leakage kind of anomalies resulting in new kind of sequence ordering). We set the tasks to varying degrees of latency requirements up to $\mathbf{100}$ milliseconds, and we stop the simulation after $\mathbf{2}$ million tasks have been analyzed. In Fig. 6, we have plotted the percentage of *stale* (anomaly detection tasks that could not finish before their timing constraint) for both the standalone (where each node works independently) situation and the distributed scenario utilizing the offloading algorithm under varying CPU load conditions. Comparing Fig. 6a and Fig. 6b, the latter consistently outperforms the former under the various CPU load conditions except in the *Random* CPU load category. Under the *Random* CPU load category, we allowed the CPU load utilization to fluctuate between the minimum (idle) and maximum (full) value. The result in this category under distributed mode is poor as we observed that bursty traffic by most of the nodes at the same time with short latency requirements made it difficult for the scheduling algorithm to manage the tasks optimally, hence the poor result in this category. The high stale tasks in the *Low* and *Moderate* CPU load categories in Fig. 6a are mostly as a result of low capacity nodes which cannot finish tasks within the time constraints. Under the *High* CPU load category, because some nodes' traffic are bursty in nature (reactive devices), the algorithm was able to manage the anomaly tasks by utilizing the resources provided by these kind of devices to ensure a high throughput.

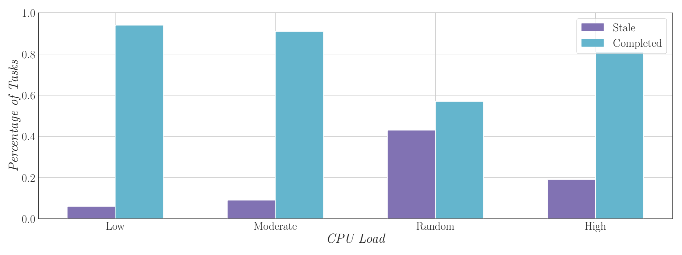
With respect to the accuracy of the anomaly detector model, we have shown a snapshot of the prediction error as the tasks are executed for the three profiles: *normal, random,* and *delay*. As explained in Section IV-C, the accuracy of the model hinges on generating low prediction error for the *normal* profile and high deviation error in any instance of the anomalous sub-sequence that contains an anomaly. We work with the replication error since it is an *unsupervised* anomaly detection framework. As seen in Fig. 7a, the *normal* profile generates low regression error (no anomalies) while the *delay* and *random* profiles exhibit high regression errors when an anomalous sub-sequence task is analyzed. Since an increasing value of the error leads to reduced chance of the sub-sequence being free of anomalies as stated in (9), Fig. 7b displays a snapshot through the *delay* and *random* profile with their level of anomalous probability. The smaller the value of $\boldsymbol{AL_\rho}$ in Fig. 7b, the greater the severity of the anomaly.

VI. Conclusion

In this paper, we propose a modular deep learning framework and an offloading algorithm that takes advantage of the increasing capacity of the edge devices to create a distributed anomaly detection framework that brings online anomaly detection capability to every node in the network. This framework brings improved security to the applications or processes in the edge nodes. We test the anomaly framework using kernel event streams, and the results confirm the ability of the offloading algorithm to manage the anomaly tasks with varying timing constraints. The deep learning-based anomaly detector also shows a high fidelity sequence replication power that ensures that anomalous sequences are detected. Our next research direction will be on improving the number of com-

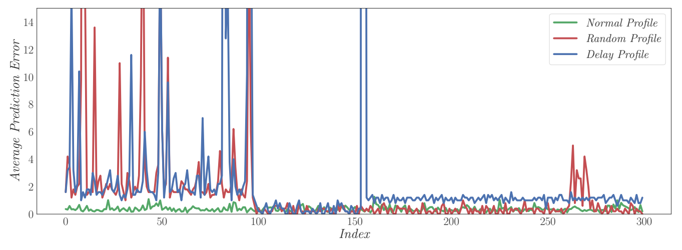


(a) Non Distributed Anomaly Detection Performance

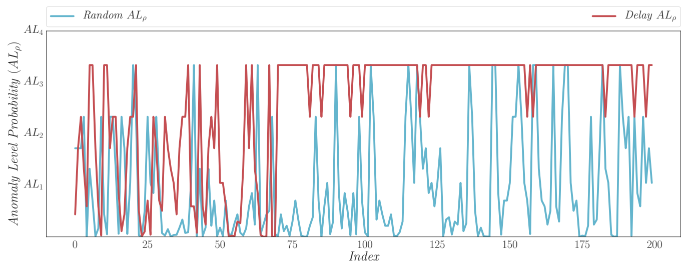


(b) Distributed Anomaly Detection Performance of Nodes

Fig. 6: Comparison of the Number of Completed Anomaly Detection Tasks in Distributed vs Non Distributed Environment



(a) Regression Error for *random, delay* and *normal* profile



(b) Instruction Cycle Count Anomaly Level Probability, $\boldsymbol{AL_\rho}$

Fig. 7: Regression Error for the *Normal, Random* and Delay Profile with the $\boldsymbol{AL_\rho}$ of the Anomalous Profiles

pleted tasks to ensure that we do not miss critical anomalous scenarios.

Acknowledgements

This research was funded in part by the Natural Sciences and Engineering Research Council of Canada (NSERC). The first author also acknowledges the support of PTDF Nigeria towards his graduate studies.

References

[1] M. K. Weldon, *The future X network: a Bell Labs perspective*. CRC press, 2016.
[2] W. N. Sumner and X. Zhang, "Comparative causality: Explaining the differences between executions," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 272–281.
[3] F. Li, Z. Li, W. Huo, and X. Feng, "Locating software faults based on minimum debugging frontier set," *IEEE Transactions on Software Engineering*, vol. 43, no. 8, pp. 760–776, 2017.
[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[5] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.
[6] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *computers & security*, vol. 28, no. 1-2, pp. 18–28, 2009.
[7] M. Ezeme, A. Azim, and Q. H. Mahmoud, "An imputation-based augmented anomaly detection from large traces of operating system events," in *Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*. ACM, 2017, pp. 43–52.
[8] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1285–1298.
[9] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Largescale system problem detection by mining console logs," *Proceedings of SOSP'09*, 2009.
[10] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," in *ACM SIGPLAN Notices*, vol. 51, no. 4. ACM, 2016, pp. 489–502.
[11] M.-K. Yoon, S. Mohan, J. Choi, M. Christodorescu, and L. Sha, "Learning execution contexts from system call distribution for anomaly detection in smart embedded system," in *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*. ACM, 2017, pp. 191–196.
[12] Y. Gu, A. McCallum, and D. Towsley, "Detecting anomalies in network traffic using maximum entropy estimation," in *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*. USENIX Association, 2005, pp. 32–32.
[13] M. Salem, M. Crowley, and S. Fischmeister, "Anomaly detection using inter-arrival curves for real-time systems," in *Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on*. IEEE, 2016, pp. 97–106.
[14] M. O. Ezeme, Q. H. Mahmoud, and A. Azim, "Hierarchical attention-based anomaly detection model for embedded operating systems," in *In 24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2018.
[15] O. M. Ezeme, Q. H. Mahmoud, and A. Azim, "Dream: deep recursive attentive model for anomaly detection in kernel events," *IEEE Access*, vol. 7, pp. 18 860–18 870, 2019.
[16] A. Chawla, B. Lee, S. Fallon, and P. Jacob, "Host based intrusion detection system with combined cnn/rnn model," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2018, pp. 149–158.
[17] G. Serpen and E. Aghaei, "Host-based misuse intrusion detection using pca feature extraction and knn classification algorithms," *Intelligent Data Analysis*, vol. 22, no. 5, pp. 1101–1114, 2018.
[18] M. Al Sharnouby, "Anomaly detection using sequences of system calls," Mar. 26 2019, uS Patent App. 10/241,847.
[19] P.-F. Marteau, "Sequence covering for efficient host-based intrusion detection," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 4, pp. 994–1006, 2018.
[20] R. Bell, "Introduction to iec 61508," in *Proceedings of the 10th Australian workshop on Safety critical systems and software-Volume 55*. Australian Computer Society, Inc., 2006, pp. 3–12.
[21] "Zeromq: An open-source universal messaging library," https://zeromq.org/, accessed: 2019-09-26.