Distributed and Self-Scaling Anomaly Detection Model for Resource-Constrained Embedded Systems

Anonymous Author(s)

ABSTRACT

Kernel events contain information that reflects the state of the system and is a source of knowledge for system monitoring from the kernel layer. However, with a massive volume of the kernel events within a short interval, the use of online anomaly models are limited because of the constrained nature of embedded systems regarding the compute capacity and energy resource. Hence, the widespread use of signature-based tools by practitioners.

In this paper, we introduce a distributed online anomaly model that uses a novel technique to leverage hub/edge/cloud resources to complement the resources of the embedded device in executing the model.

KEYWORDS

Embedded System; Distributed System; Anomaly Detection; Parallelism

ACM Reference Format:

Anonymous Author(s). 2019. Distributed and Self-Scaling Anomaly Detection Model for Resource-Constrained Embedded Systems. In *DAC 2019: Design Automation Conference*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

From the Internet of Things (IoT) to the Internet of Everything (IoE), the common denominator is *connectivity* driven by ubiquitous embedded systems. 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 [15]. 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 [5]. 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 embedded systems 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 in addition to their primary functional requirement, they are expected to run other security and safety control models to maintain the integrity of their operations. This extra requirement means that the embedded devices have to split their available resources to satisfy both the primary objective and the anomaly model. While some of the systems 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 devices with the prescribed operational standards. To optimize the impact of the anomaly model on the performance of the device with respect to its primary objective, we introduce a design mechanism that leverages the hub/edge/cloud resources to ensure that the embedded devices satisfy both the demands of their primary duty and that of the anomaly model.

If the behavior of the embedded devices and their applications are well-characterized, then the state transition checks espoused by the authors in [11, 14] 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 [5] 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 *dynamic, distributed* and *self-scaling* anomaly detection model that utilizes the temporal information in the system call execution sequences to detect anomalies.

The anomaly model processes the traces as streams of tuples and takes into account the temporal drift by building a profile that can create both short and long-term contexts of the sequences. The use of the LSTM cells and context-based attention layer provides medium to short-term contexts while the incorporation of the feedback input from the attention layer creates a long-range dependency between the present and past inputs to the model. This dynamic approach enables us to target both previously known and unknown anomalies and deepens the understanding of the execution contexts of the kernel events *horizontally* and *vertically*. This overall design employs the concept of transfer learning as we start with an unsupervised prediction model and use the knowledge from the

DAC 2019, June 2-6, 2019, Las Vegas, Nevada, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

unsupervised learning phase to create a fast supervised classifier to detect when an anomaly has occurred. The training employs a closed-world approach; hence only the kernel events from the standard operating profiles are used for training the unsupervised model.

To facilitate the capture of long and short-term temporal dependencies, we use a stack of the Long Short-Term Memory (LSTM) [9] which its variants have been used in various tasks to demonstrate its effectiveness in capturing complex non-linear relationships that exist in a sequence. The use of hierarchical LSTM learns the temporal relationships amongst 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 model 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 V_t on next prediction V_{t+1} should be derived from a deeper and longer context than just the present tuple V_t and the present attention vector 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 primary target of the work is system logs from operating systems like kernel events or system calls. To handle bias in the model, in each layer of the system which generates the logs, we train the model only with attributes that cut across the different formats emitted by the different kinds of operating systems for the layer under consideration. 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 and get the classifier for this layer. Therefore, we have two problem statements as thus: **a)** given kernel events obtained during the normal operation of a system, is it possible to create a model that uses the information solely from normal behavior to characterize the standard and deviant behaviors in the kernel events? **b)** if yes, can we dynamically deploy this anomaly model to perform online anomaly detection by leveraging hub/edge/cloud resources? To answer the questions posed above, we propose a *distributed and self-scaling anomaly detection model* primarily targeting resource-constrained embedded systems. In summary, our contributions are: **a)** we present a dynamic, distributed and self-scaling mechanism to leverage hub/edge/cloud resources to complement the operation of an embedded system. **b)** we present an online deep context-aware architecture for anomaly detection in semi-structured sequences with bias to kernel events.

To ease the comprehension of the work, we have divided the rest of the work into the following sections; Section 2 briefly highlights the related work in this domain. Section 3 discusses the technical details of our model while Section 4 details our tests as well as discussion of the results. In Section 5, we conclude the work with an insight into our prospective research directions.

2 RELATED WORK

According to [2], 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 [7]. 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 [3, 4, 16, 18] 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 [4], 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 [17] 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 [4] because it requires a long window of observation before it can make a decision. The authors of [3] 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 [8] 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 [12], a real-time systems 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 [16] designed a vector space model to mine console logs for anomalies and [11] used an optimization method of minimum debugging frontier sets to create a model for detection

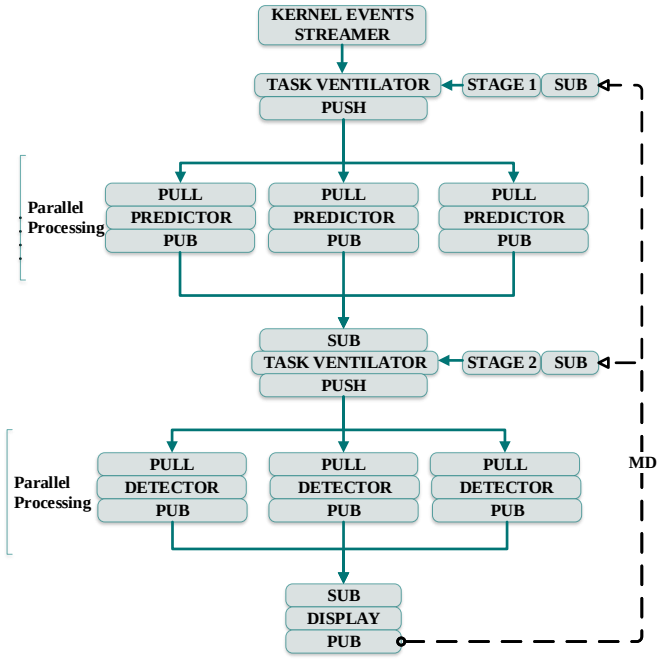

Figure 1: Dynamic, Distributed and Self-Scaling Architecture for Anomaly Model Deployment

of errors/faults in software execution. In [6], 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 model architecture and target platform. Therefore, while we introduce design mechanism to ensure a truly dynamic and distributed anomaly model, [6] creates a centralized anomaly model that places a considerable strain on embedded system resources, thereby limiting its use for online anomaly detection.

3 MODEL ARCHITECTURE

Fig. 1 is the design of our distributed model that provides the three core features of *dynamism, self-scalability* and *distribution* of the anomaly model amongst the local and cloud resources. The break down of the modules of Fig. 1 is discussed in the sub-sections below. Also, we have broadly broken down the actual anomaly detection model of Fig. 2 into two modules identified in the figure as *predictor* and *detector*. Before we proceed further, it is important that we explain some of the names in Fig. 1. *PUSH, PULL, PUB,* and *SUB* are ZeroMQ TCP sockets while *STAGES* 1 and 2 represent the points where distribution decisions are taken. We use ZeroMQ sockets in our design because unlike the traditional TCP sockets available in the operating system, ZeroMQ sockets pairs can be connected and disconnected arbitrarily in no particular order. This arbitrary connection and disconnection ability removes the need of restarting the system if one of the pairs dies and tries to reconnect. We discuss the different modules of Fig. 1 in the sub-sections below.

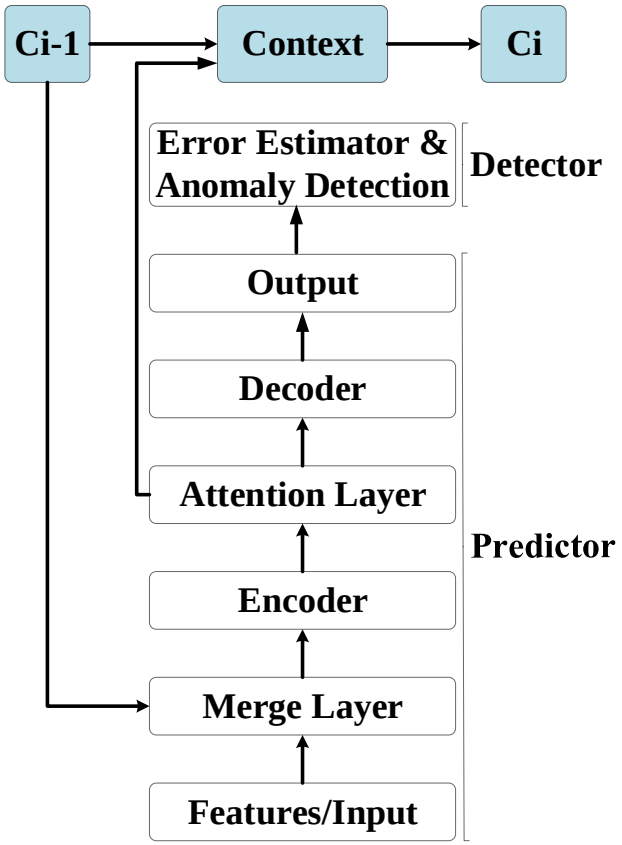

Figure 2: A Recursive Deep Context Anomaly Detection Model

3.1 Stage 1 Ventilator Module

Kernel Event Streamer. The *kernel events streamer* is a module that handles the processing of the kernel events emanating from the instrumented kernel. The raw kernel events have both useful and non-useful attributes for our model design. Therefore, the streamer module extracts the useful information from the kernel events and positions the events in the format accepted by the model. As we mentioned in Section 1, we only process the attributes which are common across the trace irrespective of the system to remove any system-induced bias on the classifier. In kernel, an example event stream of THRUNNING $\longrightarrow$ THREADY $\longrightarrow$ THRECEIVE $\longrightarrow$ THREPLY has *timestamps, PID, TID, NID*, etc. which are common across the event stream. Therefore, we formulate the characteristics of interest using these features which exist across kernel event streams irrespective of platform.

Task Ventilation. The frequency of the kernel event streamer is high, and the downstream modules have to keep up with the streams in order to avoid the sockets dropping the data. Hence, the need for parallelism at the *predictor*. As stated in Section 3, the sockets can connect and disconnect arbitrarily without restarting the network thereby providing an efficient horizontal scaling mechanism for the parallel predictors and detectors. Our dynamic allocation scheme

starts in (3.1) where we determine the number of new predictors N_p we can create locally using the free system resources (compute and memory) R_s, the energy level E_l of the embedded device and the system resources R_p required by a predictor.

$$N_p = \frac{R_s \times E_l}{R_p} \tag{3.1}$$

$$\tau_p = N_p + N_{local} \tag{3.2}$$

The total number of predictors that we can run on the embedded device is given in (3.2) pending the outcome of conditions (3.3) and (3.4). N_{local} of (3.2) refers to already running instances of predictors on the embedded device.

$$PI_{local} = \frac{\tau_p}{C_p \times F_k} \tag{3.3}$$

$$PI_{cloud} = \frac{TDD_1}{PI_{local} \times \lambda} \text{ where } \lambda \geq 1 \tag{3.4}$$

To determine the performance of the predictors locally and in the cloud, we use (3.3) and (3.4) which we refer to as the local (embedded device) and cloud performance index (PI) respectively. C_p and F_k refers to the computational time of the predictor in seconds and the kernel event streamer frequency which is measured in input/seconds. In (3.4), λ is the speed factor we get for processing in the cloud and TDD_1 is the total delay from the ventilator to the display when the predictors in the cloud are used. The metadata feedback loop of Fig. 1 provides the TDD_1 information. For every task sent to the cloud from the stage 1 ventilator, the cloud detector is equally used for anomaly detection and the result published to the display. In our model, there is no automatic system reaction to the detection of an anomaly; hence we envisage the display to be operated by a human operator. Therefore, the display is assumed to be in the monitoring room. The SUB sockets of the stage 2 task ventilator and the display provide the metadata information on the number of parallel predictors and detectors running locally via the metadata (MD) feedback loop. Now, examining (3.3) and (3.4) closely, we see that they vary inversely to one another. So, as the streaming frequency F_k increases compared to the available number of predictors, the local performance index PI_{local} decreases and the cloud performance index PI_{cloud} increases, thereby favoring more predictors to be run in the cloud and vice versa. Therefore, (3.1) to (3.4) create a *self-scalable* and *distributed* system for handling both the predictor and detector tasks. The latency requirement of the application places a bound on the viable values of TDD_1 that is permissible in the model.

3.2 Predictor

Merge Layer. Our hypothesis is based on creating a deep execution context via a recursive input generated from the attention layer. Since the attention layer has a learned weight, it means that its output which is used to create the context C contains information of multiple previous inputs, and feeding it along with the present input either reinforces a standard profile or weakens the prediction accuracy which will indicate the presence of an anomaly. Therefore, (3.5) describes our approach of merging the recursive input C_i with the present input X_i.

$$v = \text{merge}(\vec{X_i}, \vec{C_i}) \tag{3.5}$$

Encoder (LSTM Layer). Our choice of LSTM cells in this layer stems from the fact that it is designed primarily for time-series data and its recursive nature helps to propagate temporal information across so many timesteps infinitely in theory. However, we recognize that in practice, there is a limit to how far behind it can propagate the errors before the vanishing gradient problem sets in. Hence, our idea to augment it with a recursive context to improve the learnability over a long span of time. We feed the output of (3.5) to the LSTM layer. Different kinds of LSTM configuration can be used, but we use the LSTM units described in [9] to create our layer. This layer's output is captured mathematically in (3.6). We omit the bias terms for brevity, and Φ is a nonlinear function like an LSTM.

$$h_i = \Phi\left(v_i, h_{i-1}\right) \tag{3.6}$$

Attention Layer. Differing from [13] that uses memory to create context, we add a query weight W_q that is learned during training to ensure that each tuple is not attended to solely based on the information in the present input sequence but also based on the knowledge gained during the learning phase. This query performs a similar role as the term-frequency inverse document frequency used to weigh the occurrence of tuples in the vector space model. The inputs to this layer are the input weights W_i and the LSTM layer output h_i of (3.6). We pass the LSTM layer output via a tanh layer after being scaled by the input weights W_i and the input bias vector b_i to generate correlation vectors m_i given in (3.7).

$$m_i = \tanh(W_i \cdot h_i + b_i) \tag{3.7}$$

The correlation vector (3.7) represents the effect of each input based on the present. Hence, we multiply it with the query vector W_q which has the *global knowledge* of each input tuple in the present input sequence to provide deep horizontally spanning inputs for the inference process as shown in (3.8). This vector is then passed through a softmax layer to generate s_i in (3.9). This normalized value is scaled by the input vectors h_i and summed to generate the attention vector Z_i in (3.10).

$$a_i = W_q \cdot m_i + b_q \tag{3.8}$$

$$s_i = \left(\frac{e^{a_i}}{\sum_j e^{a_j}}\right)_i \tag{3.9}$$

$$Z_i = \sum_{i=1}^{n} s_i \times h_i \tag{3.10}$$

Decoder (LSTM Layer). This layer performs the function of a decoder while the lower LSTM is responsible for encoding the input. This layer in conjunction with the fully connected layer tries to reconstruct the input sequence. This layer creates an intermediate output h_{di} using the previous output y_{i-1}, the context vector Z and the previous hidden state h_{di-1} of the previous unit d_{i-1}. Equation (3.11) defines this relationship where Ψ is a nonlinear function called LSTM. Again, bias vector is omitted for brevity.

$$h_{di} = \Psi\left(h_{di-1}, y_{i-1}, Z\right) \tag{3.11}$$

Output Layer. Our output layer is a simple dense layer with same number of units as there are unique features in the input sequences. The output of this layer is given in (3.12) where $\boldsymbol{h}_{di}$, $\boldsymbol{W}_z$ and $\boldsymbol{b}_z$ are the decoding LSTM layer output, the layer weight and the bias vector respectively.

$$\boldsymbol{y}_i = \boldsymbol{W}_z \cdot \boldsymbol{h}_{di} + \boldsymbol{b}_z \tag{3.12}$$

The $\boldsymbol{y}_i$ is then passed through a softmax layer to produce the predicted system call S_i.

3.3 Stage 2 Ventilator Module

We apply the stage 1 ventilator equations of Section 3.1 to determine how we distribute the detectors but $\boldsymbol{TDD_2}$ replaces $\boldsymbol{TDD_1}$ and reference symbol is now the detector $\boldsymbol{d}$ and not the predictor $\boldsymbol{p}$.

3.4 Detector

Error Estimator and Anomaly Detection. 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 $\boldsymbol{d} = \left|\boldsymbol{x} - \boldsymbol{x'}\right|$ is the difference between the ground truth and the predicted value for the given sequence. This deviation becomes the prediction error values which we use to to fit a non-parametric kernel density estimator. The output of the estimator is clustered with k-means clustering where $\boldsymbol{k} = 2$.

3.5 Display

We send the clustering decision to the display via the PUB sockets of the detector. The display extracts the metadata information and relays the same to stages 1 and 2. Information from the display can also be used for corrective actions when an anomaly is detected.

4 EXPERIMENTAL EVALUATION

4.1 Dataset

We use the publicly available kernel event dataset of [12] to test our model. According to the authors in [12], the *hilRf-InFin* scenario corresponds to the default system setting while *full-while* scenario refers to generating fictitious tasks via a while loop to waste CPU resources by competing for the same CPU resource with the standard tasks running the UAV. The *fifo-ls* and *sporadic* situations derive their names from the scheduling algorithms in the operating system and are identified by the corresponding scheduling algorithm used in each experiment.

4.2 Experimental Setup and Networking

Our testbed constitutes of Raspberry PI 3 platform which serves as our local embedded device and an Intel-i5 laptop which serves as our cloud. Each of the scenarios in the dataset has a separate model, and our experimental objectives are: **a)** demonstrate the offloading mechanism which creates a self-scaling and a distributed model. **b)** demonstrate the improvement brought by the attention layer to the anomaly model by comparing with a *base* (no attention layer) model. We implement our model of Fig. 1 using ZeroMQ [1]. First, the predictor and detector model weights are loaded in both the Raspberry PI and the laptop computer which serves as our cloud. Our sockets does not have restrictions on which end *binds* and which end *connects*. Therefore, it our design decision that any *one-to-many* connection like that of *stage 1 task ventilator PUSH* socket and *predictor PULL* sockets, *stage 2 task ventilator PUSH* and *detector PULL* sockets, we bind the *one* and connect the *many*. This way, the PULL socket processes in the predictor and detector can be scaled up and down without affecting the network functionality since ZeroMQ permits arbitrary *connect* and *disconnect*. For the *many-to-one* connection of the *predictor PUB* and the *stage 2 task ventilator SUB*, *detector PUB* and *display SUB*, we bind the *one* and connect the *many* as well. The *PUSH-PULL* sockets provide *parallelism* while the *PUB-SUB* sockets act as *source-sink* connection. The metadata *MD* feedback loop implemented with a *PUB-SUB* connection carries information about the active number of predictors and detectors as well as the $\boldsymbol{TDD_1}$ and $\boldsymbol{TDD_2}$ delay.

Table 1: Accuracy of Attention-Based Model vs the Base Model with Varying Φ_i

Experiment	Models	Lookahead Window Accuracy		
		Φ_1	Φ_5	Φ_{10}
fifo	Base	0.794	0.746	0.724
	Att-Model	0.981	0.972	0.947
full	Base	0.836	0.791	0.767
	Att-Model	0.99	0.942	0.916
hilrf	Base	0.870	0.815	0.778
	Att-Model	0.968	0.937	0.908
sporadic	Base	0.790	0.772	0.728
	Att-Model	0.916	0.872	0.857

4.3 Results

4.3.1 Offloading Mechanism Results. To test the offloading mechanism, we vary the number of running processes in the Raspberry PI from nearly idle to near full utilization of the CPU and memory. We fixed the maximum number of predictors and detectors that can be run both in the cloud and the embedded device using (3.2). Then, we allow the offloading mechanism to use the interaction between the $\boldsymbol{PI_{local}}$ and $\boldsymbol{PI_{cloud}}$ to generate the cloud and local performance index as the system gets busier. The number of local detectors and predictors running in a particular platform is directly proportional to its performance index $\boldsymbol{PI}$. In Fig. 3, we plot the $\boldsymbol{PI}$ of the *local* (embedded system) versus the *cloud* platform. We place the Raspberry PI in the same network as the laptop and also experiment with both of the devices in different networks. In both cases, the patterns of the $\boldsymbol{PI}$ plot is the same except for a slow rise time of the $\boldsymbol{PI_{cloud}}$ when the $\boldsymbol{TDD}$ is high due to network congestion. We determine the maximum $\boldsymbol{TDD}$ based on heuristic when there is mild traffic in the network as $\boldsymbol{TDD} = \boldsymbol{p_d} + \boldsymbol{t_d} + \boldsymbol{p_t}$ where $\boldsymbol{p_d}$ is the *propagation delay*, $\boldsymbol{t_d}$ is the *transmission delay* and $\boldsymbol{p_t}$ is the *processing time* at the cloud.

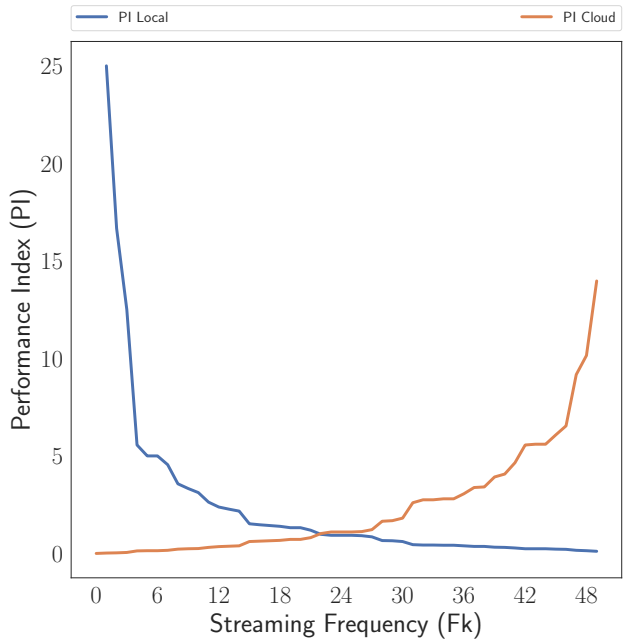

Figure 3: Performance Index *PI* of the Cloud and Embedded (Local) Platform versus Streaming Frequency F_k

4.3.2 Base Model vs Attention-based Model Results. With high input frequency rate, we varied the lookahead window to reduce the number of iterations that we run the predictor and the detector and conserve the embedded system resources. While this conserves resources, its effectiveness depends on the accuracy of our model. We represent the ratio of the input stream window to the output lookahead window as $\mathbf{\Phi} = \theta_{in}/\theta_{out}$. Therefore, $\mathbf{\Phi_i}$ is the *input-output* window ratio for lookahead length $\boldsymbol{i}$. We use early stopping and dynamic learning rate during training to control the number of training epochs and improve the performance of the *adam* [10] optimization scheme that was used during the training process. In Table 1, two patterns emerge: **a)** the attention-based model we designed consistently outperforms the base model in every $\mathbf{\Phi_i}$. **b)** there is decreasing accuracy as we increase $\boldsymbol{i}$ in $\mathbf{\Phi}$. These patterns conform with our postulation that the attention layer and the recursive input of of the model impact positively on the model performance. Also, with a fixed input window, θ_{in}, the decreasing accuracy with increasing θ_{out} is expected as the temporal information needed for longer sequence generation is restricted.

5 CONCLUSIONS AND FUTURE WORK

We propose a dynamic, distributed and self-scaling anomaly detection model targeting resource-constrained embedded devices. We detail the implementation of the model as well as the hypothesis behind the model design. Our experimental results validate our hypothesis on the offloading scheme. Also, the postulation of creating a recursive-context using the attention layer is confirmed as shown in the results. Finally, while our model can be deployed for *soft* real-time systems, we will be exploring more robust offloading schemes for *hard* real-time embedded applications so that the *TDD* bounds are based on real-time application's latency requirement.

REFERENCES

[1] Faruk Akgul. 2013. *ZeroMQ.* Packt Publishing.
[2] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM computing surveys (CSUR)* 41, 3 (2009), 15.
[3] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. 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, 1285–1298.
[4] Mellitus Ezeme, Akramul Azim, and Qusay H Mahmoud. 2017. 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, 43–52.
[5] Mellitus Okwudili Ezeme. 2015. *A Multi-domain Co-Simulator for Smart Grid: Modeling Interactions in Power, Control and Communications.* Ph.D. Dissertation. University of Toronto (Canada).
[6] Mellitus Okwudili Ezeme, Qusay H Mahmoud, and Akramul Azim. 2018. 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.
[7] Pedro Garcia-Teodoro, J Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. 2009. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security* 28, 1-2 (2009), 18–28.
[8] Yu Gu, Andrew McCallum, and Don Towsley. 2005. Detecting anomalies in network traffic using maximum entropy estimation. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement.* USENIX Association, 32–32.
[9] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[10] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[11] Feng Li, Zhiyuan Li, Wei Huo, and Xiaobing Feng. 2017. Locating software faults based on minimum debugging frontier set. *IEEE Transactions on Software Engineering* 43, 8 (2017), 760–776.
[12] Mahmoud Salem, Mark Crowley, and Sebastian Fischmeister. 2016. Anomaly detection using inter-arrival curves for real-time systems. In *Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on.* IEEE, 97–106.
[13] Giancarlo Salton, Robert Ross, and John Kelleher. 2017. Attentive Language Models. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers),* Vol. 1. 441–450.
[14] William N Sumner and Xiangyu Zhang. 2013. Comparative causality: Explaining the differences between executions. In *Proceedings of the 2013 International Conference on Software Engineering.* IEEE Press, 272–281.
[15] Marcus K Weldon. 2016. *The future X network: a Bell Labs perspective.* CRC press.
[16] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. 2009. Largescale system problem detection by mining console logs. *Proceedings of SOSP'09* (2009).
[17] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu, and Lui Sha. 2017. 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, 191–196.
[18] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. 2016. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. In *ACM SIGPLAN Notices,* Vol. 51. ACM, 489–502.