DaSAD: Distributed and Scalable Anomaly Detection Framework for Embedded IoT Devices

Anonymous Author(s)

ABSTRACT

Internet of Things (IoT) connects multiple devices to the internet and allows these *connected devices* to exchange information; thereby generating a massive amount of data within a short interval of time. The applications running on the *connected devices* generate system call events as a consequence of their execution. These kernel events contain information that reflects the state of the system and when adequately harnessed, can be a suitable tool for system monitoring from the kernel layer.

However, with the vast volume of the kernel events within a short window of time, considering that these devices' primary job is not anomaly detection, the use of online anomaly models are limited because of competition for resources with the default programs running on these devices. Hence, the widespread use of signature-based tools by practitioners. In this paper, we introduce an online distributed and scalable anomaly detection (DaSAD) framework that uses a novel offloading technique that leverage edge/cloud resources to complement the resources of the leaf devices in an IoT environment when the anomaly detection model is active.

KEYWORDS

Internet of Things; Embedded System; Distributed System; Anomaly Detection

ACM Reference Format:

Anonymous Author(s). 2019. DaSAD: Distributed and Scalable Anomaly Detection Framework for Embedded IoT Devices. In *LCTES 2019: Languages, Compilers, Tools, and Theory of Embedded Systems.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

LCTES 2019, June 22-28, 2019, Phoenix, Arizona, USA

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

1 INTRODUCTION

From IoT to the Internet of Everything (IoE), the common denominator is *connectivity* driven by ubiquitous embedded systems. These devices and infrastructures create a cyber-physical framework that predicts and automates the mundane, enabling people to concentrate on more productive things [18]. This smart 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, make predictions based on the available information and if it detects an anomaly, may take corrective, preventive or stabilizing action on the system [7]. In this way, it produces a new utility in the form of time which is the positive aspect. Nevertheless, this increased connectivity era implies that an automated system handles both our safety and non-safety critical data and actions. Therefore, a breach in the expected performance 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 specification, they are expected to run other security and safety control applications 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 detector application. 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 framework 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 DaSAD framework on the performance of the device to its primary objective, we introduce a partitioning 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 DaSAD framework by creating a *decentralized* and *federated* operation for the DaSAD framework.

If the operation of the IoT devices and their applications are well-defined, then the state transition graphs advocated by the authors in [13, 17] can be applied to detect any deviations in the IoT process behavior. However, in this era of data and connectivity explosion, this approach becomes taxing because: **a)** it is daunting to define all the possible states of the tuples associated with a particular process running in the IoT device 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 DaSAD framework that utilizes the temporal information in the system call execution sequences to detect varying degrees of anomalies or faults.

The anomaly detection model component of the DaSAD framework[1] processes the traces as streams of tuples and takes into account the temporal drift by building a profile that captures both the short and long-term contexts of the sequences. The anomaly model operates as *federated* agents to suit the working of the offloading mechanism. Since kernel events are time-series data, we build the core of the anomaly model using LSTM cells and context-based attention layer that learn the profile of the application/process in an *unsupervised* manner. This dynamic approach enables us to target both known and unknown anomalies. The overall design of the anomaly detection model employs the concept of transfer learning as we start with an unsupervised *prediction* model and use the knowledge from the unsupervised learning phase to create a fast supervised anomaly *detector* that detects when an anomaly has occurred. The training employs a closed-world approach; hence we use only the kernel events from the standard operating profiles for training the unsupervised model.

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 context-aware attention mechanism 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.

[1]Anomaly framework is made up of the partitioning scheme and anomaly detection models

1.1 Contributions and Objectives

We can generate system calls via the regularly time-scheduled tasks, and sometimes, these system calls occur as a result of interrupts which are event-driven. Moreover, when it is event-driven, *fast* and *slow* profiles can be observed making it difficult to use one of the known types of distribution to model the behavior of the process. Also, for a real-time process, the constraint on response and execution time can best be modeled by observing the timestamp property of the system call and making use of it in creating the model. The fact that some processes may be time-driven, event-driven or a mixture of both makes a one-size-fits-all solution difficult and complicates the use of system call information for context modeling. Hence, in this paper, we introduce a hybrid anomaly model that uses RNN to profile the behavior of time-driven and event-driven processes based on system call information. The temporal nature of the system traces motivates the use of the RNN architecture to capture the temporal relationships because the *frequency*, *order* and *timestamp* information of the traces play a significant role in understanding the behavior of the process or application. We use the vanilla long short-term memory (LSTM) [11] variant of RNN because of its proven performance in time-series data prediction [14] and ability to capture long temporal dependency. Therefore, our contributions in this paper are as follows;

- **a)** design and development of a novel partitioning algorithm for IoT devices.
- **b)** extraction of fine-grained features from system calls to capture the broad representation of process behavior, and increase the scope of the anomalies we can detect in order to match the ever-increasing sophistication of threats.
- **c)** design and implementation of a federated anomaly detection model for profile monitoring in IoT devices.
- **d)** development of an experiential testbed for creating a dataset in order to facilitate research in this area.

Based on the contributions stated above, we will be investigating the following as our objectives in this paper; **a)** the capability of the anomaly detection model to *generalize* on the typical operating profile of the process/application. **b)** the ability of the model to detect deviations from the standard profile and quantify the anomaly based on the impact of the variation on the system profile. **c)** the residual effects of an anomaly on the system behavior. **d)** , and the efficiency of

the partitioning scheme as measured by the throughput. 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 partitioning algorithm and anomaly models while Section 4 details our experimental 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 [3], 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 [9]. 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 [5, 6, 20, 22] 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 [6], 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 [21] 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 [6] because it requires a long window of observation before it can make a decision. The authors of [5] 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 [10] 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 [15], 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 [20] designed a vector space model to mine console logs for anomalies and [13] used an optimization method of minimum debugging frontier sets to create a model for detection of errors/faults in software execution. In [8], 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, [8] creates a centralized anomaly model that places a considerable strain on embedded system resources, thereby limiting its use for online anomaly detection.

3 DaSAD FRAMEWORK DESIGN

3.1 Anomaly Model

The anomaly model as a component of DaSAD is the application to be partitioned by the partitioning scheme we discuss in Section 3.2. We discuss this component first because parts of it will be used in deriving the partition scheme equations. Here, we discuss how we have designed the anomaly model to take advantage of the decentralization provided by the partitioning scheme. The high-level overview is that we stream the system call traces from the instrumented kernel and we feed the streams to the model via the *data processing* block. The two channels are *concurrent* but the *event-driven* network lags the *time-driven* network because they target different kinds of anomalies. While the *time-driven* pool targets the temporal ordering of the system calls, denial of service or clock manipulation attacks, and injection attacks like buffer overflow which changes the system call arguments, the *event-driven* lane targets anomalies that tend to cause a *burst* consisting of *seen/unseen* system calls in the system. The two pools work on the same principle of sequence replication but while the *time-driven* network regenerates the system calls, and their expected instruction cycle (IC) count, the *event-driven* network replicates the *system call relative frequency distribution* of the input within a window of observation. The errors incurred as a result of this replication is fed to the anomaly detector to enable it to set an upper bound for

each of the networks. We temporarily store the inputs to the *event-driven* network in a buffer pending onward processing (conversion to **SRF**) and transmission to the *event-driven* LSTM network. The buffer is necessary because $F_{TD}/F_{ED} \geq 1$ where F_{TD} and F_{ED} are the operational frequencies of the *time-driven* and *event-driven* LSTM networks respectively. In summary, while *time-driven* network targets anomalous sequence calls with *local temporal profile*, *event-driven* network targets anomalous system call sequences that has *long temporal profile*.

3.1.1 Feature Processor. The system call traces contain several properties, but the features of interest to us are *timestamp*, *system call ID* and *system call arguments*. To create a relative deterministic model, we avoided using the CPU clock or wall clock as a timestamp; alternatively, we use the CPU cycle count. We assume that the process is running in a scaled-down operating system with not much-competing processes as typically obtained in embedded IoT applications. Therefore, a single trace is a multivariate variable which undergoes further processing to yield the desired features. Given *timestamp* as **t**, system call string name as **k** and system call arguments as **a**, we process these properties to yield a multi-input feature space that we feed to our anomaly models.

Timestamp and System Call Classes. During the learning phase, we store the training samples in a database, but the model runs *online* after learning. Given an observation of system call samples $\{s_1, s_2, s_3, \ldots, s_n\}$ in storage where $\boldsymbol{n}$ is the total number of the observed traces, then the function $\boldsymbol{\beta}$ of (3.1) defines the time window between system calls. We target both the *ordering* of the system calls and the relative *duration* within such a relationship. This way, for malicious code which does not alter the ordering of the system calls but creates anomalous execution times or delays which varies from the relative duration defined by $\boldsymbol{\beta}$ will be labeled a deviation.

$$\boldsymbol{\beta} : (\boldsymbol{t}_i, \boldsymbol{t}_{i+1}) \longmapsto \boldsymbol{t}_{i+1} - \boldsymbol{t}_i;\ i \in \mathbb{N} \tag{3.1}$$

To get $\boldsymbol{S}$ in Fig. 1, we use the function defined in (3.2) to encode the system call strings into their corresponding Linux tables. There are a total of 330 unique system calls in the `X_64` platform.

$$\boldsymbol{S} : \boldsymbol{k} \longmapsto \boldsymbol{w};\ \text{where } \boldsymbol{w} = \{w \in \mathbb{N} \mid 1 \leq w \leq 330\} \tag{3.2}$$

System Call Arguments. The system call argument is alphanumeric and special character strings. System call arguments are part of the features we process because it has some compelling distribution which we use to discriminate between a valid and incorrect behavior. The motivation is that while there is no consistent frequency distribution in a particular system call, a little reorganization (e.g., sorting) of the relative frequency distribution of the characters yields a thoughtful insight into its usability.

System call arguments are strings, and to maintain a relatively few vocabularies, we encode the string characters using *ASCII* values. This encoding provides us with a range of unique **256** classes. After encoding the string values, we calculate the *frequency distribution* and *relative frequency distribution* for the **256** classes in each system call argument. Our aim is not to check how consistent each *ASCII* value is across the system call but to find the *steepness* of the distribution of the characters when we sort them. Checking for consistency of each *ASCII* value detected during training amounts to assuming that the other characters not seen during learning cannot be present in the system call arguments during operation, and this is unrealistic. Therefore, we do not consider the kind of *ASCII* values present in the argument; we are somewhat concerned about the *steepness* of the distribution of the characters per system call. Although we do not expect a completely normal distribution, legitimate system calls tend to maintain a fairly regular distribution. Therefore, we posit that this will improve the model accuracy because malicious codes that target the system arguments like *buffer overflow* usually stuff the system call arguments with more characters or some unprintable characters to achieve their aim. Although tools like autocorrelation or Naive Bayes can be used to understand the relationship amongst features, these tools learn on the assumptions that each sample is independent (thereby lacking temporal relationship), and it focuses on the relationship amongst features while we focus on the distribution of the sorted *ASCII* values in a sample regardless of the features it contains.

To illustrate our point further, let us assume that a system call argument returned the following frequency distribution $\mathbf{x} = [34, 0, 56, 78, 27, 10, 34, 0, 23, 0, 0, 5, 0, 0, 75, 6, 68, 0, 90]$ for **19** *ASCII* classes monitored. Then we compute the relative frequency distribution ($\mathbf{RFD}\boldsymbol{x_i}$) of one class using $\mathbf{RFD}_{\boldsymbol{x}}^{\boldsymbol{i}} = x^i / \sum_{i=1}^{|x|}$. Therefore, the $\mathbf{RFD}_{\boldsymbol{x}}$ is given as [.06,0,.11,...,.13,0,.18]. Now the sorted system call argument relative frequency distribution (**A**) of Fig. 1 is the sorted version of **RFD**. To drive home our argument, Fig. 2 is a plot of the *sorted* **A** and *unsorted* **A**. When we present the unsorted **A** to a machine learning model, it tries to learn the relationship amongst the features at the index of the *x-axis*. Therefore, it takes only a structured input whereby the index of the incoming features are determined by the class of the feature. On the other hand, when we present the sorted **A** to a model, we are forcing it to learn how the steepness of the distribution that connects

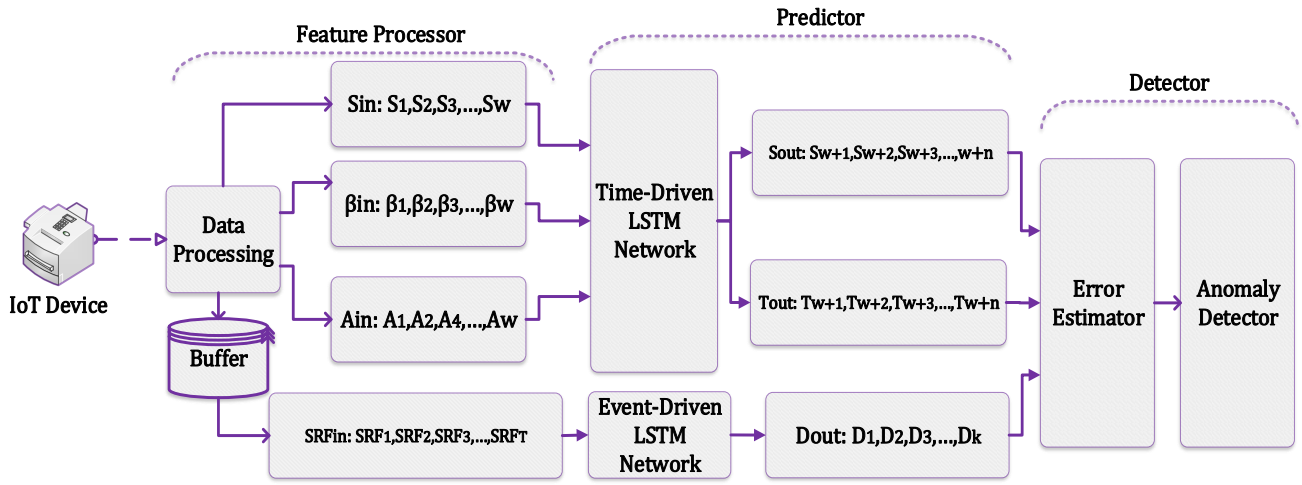

Figure 1: Architecture of the Anomaly Model for one IoT Device

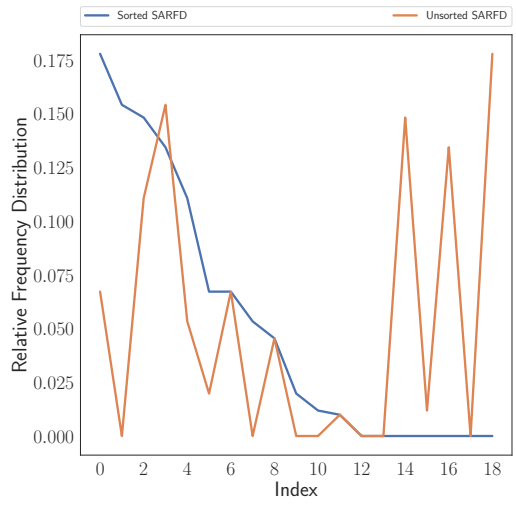

Figure 2: Sorted vs Unsorted Relative Frequency Distribution

the observed classes of the *ASCII* values in a sample without recourse to a *fixed index for a particular class*. Therefore, we impose the condition of (3.3) where $\mathbf{n}$ has a maximum value of 256.

$$\mathbf{A_i} \geq \mathbf{A_{i+1}}, \ldots, \geq \mathbf{A_n} \text{ where } \mathbf{n} \in \mathbb{N} \tag{3.3}$$

Hence, (3.4) provides the mapping from the raw string arguments to the $\mathbf{A}$ feature highlighted in the architecture of Fig. 1.

$$\mathbf{A} : \mathbf{L} \longmapsto \mathbf{M} \text{ where} \tag{3.4}$$

$$\mathbf{L} = \{l \in \mathbb{N} \mid 1 \leq l \leq 256\} \tag{3.5}$$

$$\mathbf{M} = \{m \in \mathbb{R} \mid 0 \leq m \leq 1\} \tag{3.6}$$

$$\sum_{i=1}^{256} \mathbf{A}_i = 1 \tag{3.7}$$

Also, for every system call encoded with (3.2), the *data processing* block of Fig. 1 transmits it to the *event-driven* lane via the buffer. Then, for every F_{TD}/F_{ED}, the system call relative frequency distribution (**SFR**) is computed in the *event-driven* channel as an input to the corresponding LSTM network.

3.1.2 *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, (3.8) describes our approach of merging the recursive input $\boldsymbol{C_i}$ with the present input $\boldsymbol{X_i}$.

$$\boldsymbol{v} = \text{merge}(\vec{X_i}, \vec{C_i}) \tag{3.8}$$

LSTM Layer (Encoder). 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 discussed in [19] 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 (3.8) to the LSTM layer. Different kinds of LSTM configuration can be used, but we use the LSTM units described in [11] to create our layer. This layer's output is captured mathematically in (3.9). We omit the bias terms for brevity, and Φ is a nonlinear function like an LSTM.

$$\boldsymbol{h}_i = \Phi\left(\boldsymbol{v}_i, \boldsymbol{h}_{i-1}\right) \tag{3.9}$$

Attention Layer. Differing from [16] that uses memory to create context, we add a query weight $\boldsymbol{W}_q$ 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 $\boldsymbol{W}_i$ and the LSTM layer output $\boldsymbol{h}_i$ of (3.9). We pass the LSTM layer output via a tanh layer after being scaled by the input weights $\boldsymbol{W}_i$ and the input bias vector $\boldsymbol{b}_i$ to generate correlation vectors $\boldsymbol{m}_i$ given in (3.10).

$$\boldsymbol{m}_i = \tanh(\boldsymbol{W}_i \cdot \boldsymbol{h}_i + \boldsymbol{b}_i) \tag{3.10}$$

This correlation vector ((3.10)) represents the effect of each input based on the present. Hence, we multiply it with the query vector $\boldsymbol{W}_q$ 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 (3.11). This vector is then passed through a softmax layer to generate $\boldsymbol{s}_i$ in (3.12). This normalized value is scaled by the input vectors $\boldsymbol{h}_i$ and summed to generate the attention vector $\boldsymbol{Z}_i$ in (3.13).

$$\boldsymbol{a}_i = \boldsymbol{W}_q \cdot \boldsymbol{m}_i + \boldsymbol{b}_q \tag{3.11}$$

$$\boldsymbol{s}_i = \left(\frac{e^{\boldsymbol{a}_i}}{\sum_j e^{\boldsymbol{a}_j}}\right)_i \tag{3.12}$$

$$\boldsymbol{Z}_i = \sum_{i=1}^{n} \boldsymbol{s}_i \times \boldsymbol{h}_i \tag{3.13}$$

LSTM Layer (Decoder). This layer in conjunction with the fully connected layer reconstructs the input sequence. This layer creates an intermediate output $\boldsymbol{h}_{di}$ using the previous output $\boldsymbol{y}_{i-1}$, the context vector from the attention layer $\boldsymbol{Z}$ and the previous hidden state $\boldsymbol{h}_{di-1}$ of the previous unit d_{i-1}. Equation (3.14) defines this relationship where Ψ is a nonlinear function called LSTM. Again, bias vector is omitted for brevity.

$$\boldsymbol{h}_{di} = \Psi\left(\boldsymbol{h}_{di-1}, \boldsymbol{y}_{i-1}, \boldsymbol{Z}\right) \tag{3.14}$$

Fully Connected (Output) Layer. Our FC layer is a simple dense layer with a dimension matching the expected output format. The $\boldsymbol{S_{out}}$ and $\boldsymbol{T_{out}}$ output have the same dimension but $\boldsymbol{S_{out}}$ is the number of systems call predicted for $\boldsymbol{w}$ number of steps while the $\boldsymbol{T_{out}}$ generates the corresponding expected IC for each system call predicted by $\boldsymbol{S_{out}}$. Each of $\boldsymbol{S_{out}}$ and $\boldsymbol{T_{out}}$ is represented by (3.15) where $\boldsymbol{h}_{di}$, $\boldsymbol{W}_z$ and $\boldsymbol{b}_z$ are the decoder LSTM layer output, the layer weight and the bias vector respectively.

$$\boldsymbol{y}_i = \boldsymbol{W}_z \cdot \boldsymbol{h}_{di} + \boldsymbol{b}_z \tag{3.15}$$

The $\boldsymbol{y}_i$ is further passed through a softmax layer to produce the predicted system call S_i. The whole *predictor* network is implemented using the Keras/TensorFlow deep learning tool [4].

3.1.3 Detector.

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. The anomaly model is a *multi-input, multi-output* (MIMO) architecture. Hence, the prediction 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.

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 in order to capture 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 μ. The inequality is given in (3.16).

$$P\left(\left|X-\mu\right| \geq \Upsilon\right) \leq \frac{\sigma^2}{\Upsilon^2} \tag{3.16}$$

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 $d - \mu > 1$. Hence, lack of symmetry will not affect the threshold computation. Therefore, we interchange Υ in (3.16) with $\left|d-\mu\right|$ to derive (3.17) which we use to compute the decreasing probability for increasing deviation $\left|d-\mu\right|$ where $d > \mu$.

$$P\left(d > \mu\right) = P\left(\left|X-\mu\right| \geq \left|d-\mu\right|\right) = \frac{\sigma^2}{(d-\mu)^2} \tag{3.17}$$

One of the advantages that this probability 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, 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* [2]. Our quantized levels are called *anomaly level probability*, (AL_p) and each level depends on the value of the probability from (3.17). As (3.17) measures how the error values are clustered around the mean, the model is expected to create high fidelity predictions (reconstruction of the normal profile sequence) to ensure that (3.17) performs optimally and reduces false negatives.

3.2 Partition Scheme

Fig. 3 is the design of our distribut model that provides the three core features of *dynamism, scalability* and *decentralization* of the anomaly model of Section 3.1 amongst the local and cloud resources. Before we proceed further, it is vital that we explain some of the names in Fig. 3. *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. And this asynchronous behavior removes the need of restarting the system if one of the pairs dies and enables the partition algorithm to *birth* or *kill* connections on the fly during runtime.

The partitioning scheme works as a data streamer in which the objective is to transfer the highest amount of data possible from the *feature processor* to the *display* unit of Fig. 3. Therefore, we can represent the scheme as a graph $G = (N, P)$ where $N = \{j | j = 1, 2, 3, ..., n\}$ represents the nodes *feature processor*, *predictor*, and *detector*, while $P = \{(j,k) | j,k \in N\}$ are communication paths linking node j to node k. Given F, P and D as the *feature processor*, *predictor* and *detector* partitions respectively, then, the total time to convey a unit of data from F to the *display* is given in (3.18).

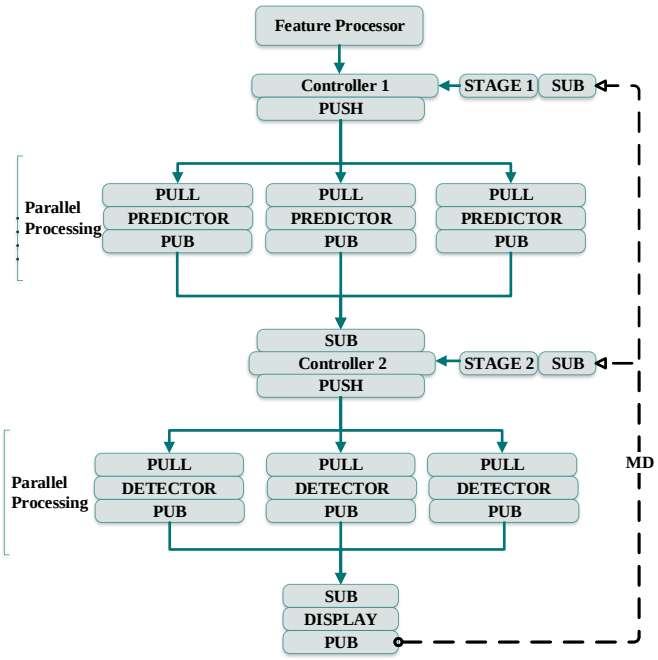

Figure 3: DaSAD Architecture showing the Decentralization and Parallelism

$$T_{F,display} = \sum_{j}^{n} T_{j,j+1} \; \forall j, n \in N \tag{3.18}$$

Note that n denotes the detector node in (3.18).

The constraints in our partitioning scheme modeling are: *communication bandwidth* between nodes B, free *local system resource* R_s, the *energy level* E_l of the IoT device and the system resource needed by the node application at that partition Ra. Therefore, $T_{j,j+1}$ = With so many frivolous properties logged during the kernel tracing, and a constrained transmission bandwidth, compute and memory resources as well as energy, the partition scheme determines the

Feature Processor. The *feature processor* handles the processing of the kernel events emanating from the instrumented kernel as described in Section 3.1.1. 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 open ⟶ mmap ⟶ close ⟶ read has *timestamps, PID, TID, NID*, etc. which are common across the

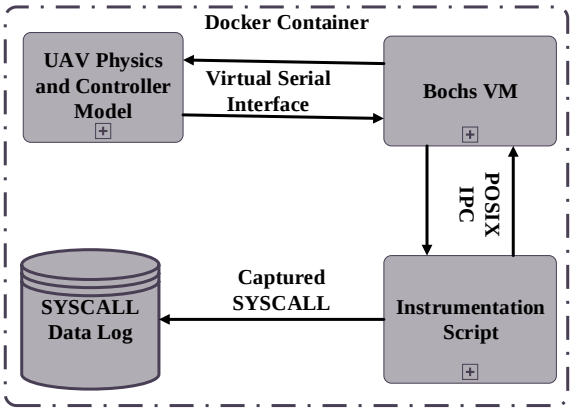

Figure 4: Setup of the Experiment for Dataset Generation

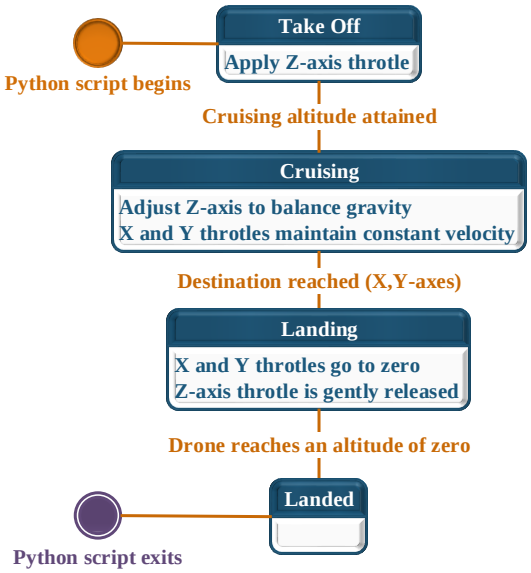

Figure 5: State Machine Diagram of the UAV Controller

event stream. Therefore, we formulate the characteristics of interest using these features which exist across kernel event streams irrespective of platform.

3.3 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 [15] to test our model. According to the authors in [15], 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. 3 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 TDD_1 and TDD_2 delay.

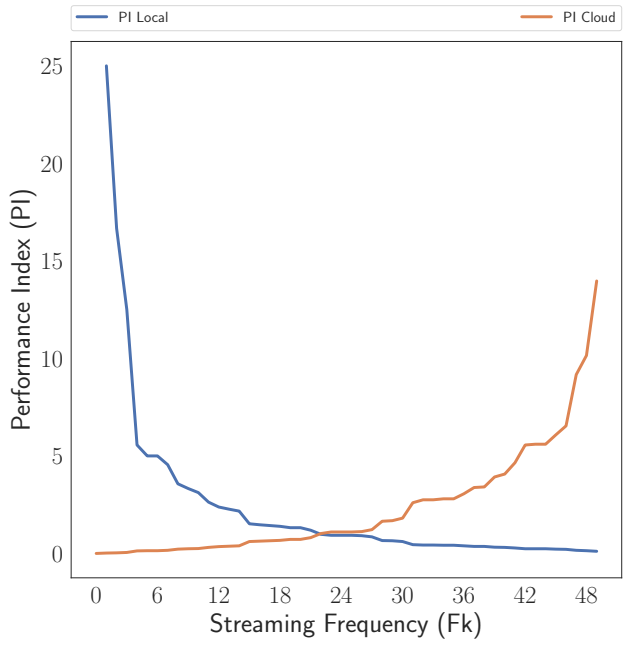

Figure 6: Performance Index *PI* of the Cloud and Embedded (Local) Platform versus Streaming Frequency F_k

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 (**??**). Then, we allow the offloading mechanism to use the interaction between the PI_{local} and 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 PI. In Fig. 6, we plot the 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 PI plot is the same except for a slow rise time of the PI_{cloud} when the TDD is high due to network congestion. We determine the maximum TDD based on heuristic when there is mild traffic in the network as $TDD = p_d + t_d + p_t$ where p_d is the *propagation delay*, t_d is the *transmission delay* and p_t is the *processing time* at the cloud.

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

4.3.2 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 $\Phi = {}^{\theta_{in}}/_{\theta_{out}}$. Therefore, Φ_i is the *input-output* window ratio for lookahead length 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* [12] 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 Φ_i. **b)** there is decreasing accuracy as we increase i in Φ. 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] Ron Bell. 2006. Introduction to IEC 61508. In *Proceedings of the 10th Australian workshop on Safety critical systems and software-Volume 55*. Australian Computer Society, Inc., 3–12.

[3] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM computing surveys (CSUR)* 41, 3 (2009), 15.

[4] François Chollet et al. 2015. Keras. https://keras.io.

[5] 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.

[6] 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.

[7] 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).

[8] 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.

[9] 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.

[10] 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.

[11] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[12] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[13] 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.

[14] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, and Puneet Agarwal. 2015. Long short term memory networks for anomaly detection in time series. In *Proceedings*. Presses universitaires de Louvain, 89.

[15] 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.

[16] 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.

[17] 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.

[18] Marcus K Weldon. 2016. *The future X network: a Bell Labs perspective*. CRC press.

[19] Paul J Werbos. 1990. Backpropagation through time: what it does and how to do it. *Proc. IEEE* 78, 10 (1990), 1550–1560.

[20] 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).

[21] 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.

[22] 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.