2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)
October 11-14, 2020. Toronto, Canada

GWAD: Greedy Workflow Graph Anomaly Detection Framework for System Traces

Wiliam Setiawan
Department of Computer Science
The University of British Columbia
BC, Canada
wiliamd.great@gmail.com

Yohen Thounaojam
Department of Computer Science
The University of British Columbia
BC, Canada
yohenthounaojam@gmail.com

Apurva Narayan
Department of Computer Science
The University of British Columbia
BC, Canada
apurva.narayan@ubc.ca

***Abstract*—System traces are a collection of time-stamped messages recorded by the operating system while the system is running. Analysis of these traces is crucial for tasks such as system fault finding. Moreover, detecting anomalies in system behavior becomes crucial in safety-critical and time-sensitive systems where delayed detections can lead to catastrophic outcomes. Therefore, we focus on developing a lightweight and explainable approach for safety-critical time-sensitive systems.**

Given a set of system traces under normal conditions and anomalous conditions, trace-based anomaly detection aims at classifying the trace as anomalous or not. In this work, we introduce GWAD, a greedy workflow graph framework for anomaly detection, a novel greedy graph construction approach for both offline and online anomaly detection in system traces. Our approach utilizes both sequence of occurrence of events and the time interval between their occurrences in learning the normal system behavior. We propose two approaches, first for offline classification of the trace as anomalous or normal using the event occurrence workflow graphs and secondly an online streaming algorithm that monitors the events as they occur in real-time for detecting anomalies increasing system resilience. Our approach also provides reasoning for the cause of anomalous behavior. We show that GWAD is better than traditional state-of-the-art models. The paper shows the technical feasibility and viability of GWAD through multiple case studies using traces from a field-tested hexacopter.

***Index Terms*—anomaly detection, graph, greedy algorithm, cyber-physical systems, workflow graphs**

I. Introduction

Most software systems including the ones on safety-critical real-time systems provide state information in the form of log files or system traces. These files reveal behavioral information about the systems if it is operating correctly or which instruction is under execution etc. System engineers analyze such information to detect anomalies in system software that can help identify potential security leaks. Moreover, most complex software systems produce huge quantities of these traces, and it becomes infeasible to analyze these traces manually. Thus, it becomes critical to develop and use automated techniques for analysis of system traces. Analysis can reveal important insights about system behavior and help with fault diagnosis and root cause analysis.

Specifically, in the context of embedded systems, there are a large number of heterogeneous subcomponents that work together to ensure safe and secure operations [1]. These systems are provided with requirements and standards such as the ISO-26262 in automotive systems or DO178C for airborne systems. Analysis of system traces enables engineers to provide assurance of conformance with such standards. Automated analysis of these requirements from system traces helps enhance the credibility of post-mortem analysis in embedded systems where most of these systems generate traces as a part of their execution.

Anomaly detection or outlier detection is a well-studied topic in the statistics community and dates back to the 19^{th} century [2]. Recently, a variety of anomaly detection techniques have been developed in different domains and research communities. Anomaly detection primarily focuses on finding patterns in data that allow one to identify if the system is under normal operation or abnormal. These patterns allow one to identify points in system operation that lead to anomalies, discordant behavior, exceptions, etc. Anomaly detection finds extensive use in a wide variety of applications such as intrusion detection for cyber-security, fault detection in safety-critical systems, insurance or health care, fraud detection for credit cards and military surveillance for enemy activities.

Workflow graph-based anomaly detection techniques aims to detect attacks as a deviation from the normal behavior. As opposed to machine learning-based approaches for anomaly detection, graph-based approaches require handcrafted or semi-automated approach to system behavioral modeling that captures legitimate system behavior in some formal way. Quite often a workflow graph is used. This approach minimizes the rate of false alarms caused by the legitimate but previously unseen behavior. The downside of such approaches is the development of workflow graphs that can be time-consuming and extremely tedious. Our approach tries to minimize both the rate of false alarms and the complexity of defining the workflow graphs for the system.

In this work, we propose a novel greedy approach to workflow graph construction from system traces and present an approach for offline anomaly detection and an online approach to monitoring system behavior with a focus on safety-critical real-time systems. The key contributions of this paper include:

- A novel greedy approach to workflow graph construction from system traces with consideration to sequence and inter-arrival time between events.
- An approach for offline anomaly detection by comparing the workflow graphs generated from normal and anomalous systems traces.
- An approach for online monitoring of system traces for online anomaly detection.

The rest of the paper is organized as follows: Section II will be discussing similar work in related topics, such as anomaly detection from system traces, Section III will explain the steps taken to generate our workflow graph, and also explain the details on both our offline and online anomaly detection framework. Section IV lays in detail the experimental setup. This section will also explain the QNX operating system data set, and Section V will provide results, conclusions, and direction for future work.

II. Related Work

System trace-based anomaly detection is an important area of study in a variety of domains ranging from business process mining to embedded systems. There has been comprehensive reviews of various anomaly detection techniques that have been employed in diverse domains [3].

Moreover, people have explored a variety of approaches for anomaly detection from system traces ranging from information-theoretic approaches [4] where authors use information-theoretic measures such as entropy, conditional entropy, relative conditional entropy, information gain, and information cost for anomaly detection. However, some interesting approaches similar to ours have been explored in [5] where they introduced two methods for graph-based anomaly detection. In [5] they introduce two methods for calculating the regularity of the graph, with application to anomaly detection. They even hypothesize that their method will be useful for finding anomalies and finding irregularities in graph-based data. However, their approach expects to receive data in the form of graphs. Quite often obtaining graphs from system traces is not trivial, therefore one needs to develop novel approaches to identifying graphs that depict the workflow of the system under consideration.

Anomaly detection from log files is an important and dynamic field of study for networks. A comprehensive review of various facets of network anomaly detection and/or network intrusion detection techniques has been presented in [6]. They provide a crisp classification based on the computational technique used in the work.

Another interesting application for anomaly detection is in the recovery of disk storage metrics from low-level trace events. In [7] authors introduce a novel framework that can calculate meaningful storage performance metrics from low-level trace events generated by LTTng. Moreover, they adopt a stateful approach to analyze the storage subsystem. They also provide a visualization system that provides different graphical views that represent the collected information in a convenient manner. In Linux based systems, console log analysis plays a significant role in analyzing system health and behaviours. This can be seen in [8] where they develop a reachability graph that is capable of computing the reachable relations of log files. They employ information retrieval techniques to analyze the text in the log files. They propose a novel anomaly detection algorithm that considers traces as sequence data and uses a probabilistic suffix tree-based method to organize and differentiate significant statistical properties possessed by the sequences.

In anomaly detection techniques, it is important to evaluate techniques that aid the decision-maker to perform root cause analysis. An example of this is [9] where they utilize an approach that localizes anomalous invoked methods and their physical locations by leveraging request trace logs. In their approach, they use a two-step process that first clusters and then reduces the dimensionality. This is followed by similarity check using Jensen-Shannon divergence. As mentioned before, these system trace share similar qualities with business process log files, and as such detecting anomalies within these files are also similar in nature. In [10] they propose a new model that utilizes dynamic bayesian networks. This model also includes numerical attributes and functional dependencies and is used to model the log's behavior. One feature that stood out in this approach was the decomposition score which indicates causes for anomalies.

Graph-based anomaly detection approaches have also been explored, for instance, in [11] authors propose an approach to mine causality and to diagnose root cause analysis of a system. They utilized a causal search algorithm and also knowledge graph technology in their method. They verified their approach on a native cloud application. A close-knit work related to runtime anomaly detection for embedded systems has been proposed in [12] where they monitor the system non-intrusively by using the on-chip hardware. More specifically, they use the trace port of the processor which detects any malicious activity at runtime. They also monitor the timing distribution for the control flow of events. Numerous solution approaches use dependency graphs as solution techniques [13]. A comprehensive survey of graph-based anomaly detection techniques has been presented in [14].

Frequent pattern mining is a well-researched area and is utilized in a variety of domains for system behavior identification, automated reasoning, data mining from system traces, or business process logs. This pattern mining approach can also help in analyzing system behavior and to develop robust and reliable anomaly detection frameworks. In [15] authors there outlines an overview in current frequent pattern mining works. They also provide insight in promising research directions that can utilize frequent pattern mining. Similarly, temporal sequence mining approaches have also been explored for application in anomaly detection such as [16]–[19].

Our work focuses on workflow-based graph generation and utilizes them to detect anomalies within a system and to do runtime monitoring for a wide variety of systems. The next section will provide further details of our proposed framework GWAD.

III. Greedy Workflow Graph Anomaly Detection Framework

We formalize the problem of workflow graph inference in terms of the inference of a weighted undirected graph which captures the sequential relationship between events in the system trace and the inter-arrival time between the events. Before we explain our algorithm, let us define some preliminaries.

Definition 1 (Trace and Event): The alphabet of events is a finite alphabet of strings. A timed sequence of events is the trace. A single unique event will serve as nodes of the graph. The trace will serve as the graph where each subsequent events are connected by an edge. The end of the trace will serve as the terminal node of the graph.

The sequences of events in the trace are ordered by time stamps. The alphabet of events is defined by the system generating the traces. The events have associated meaning pertaining to the functionality of the system.

We formulate the problem of learning the workflow graph topology as the inference of an undirected graph $G = (V, E)$, where the vertices $V = v_i$ represent the unique events that occurs in the system trace, and the edges $E = e_{i,j}$ represent the relationship between these events; an edge $e_{i,j}$ denotes a path from the event v_i to an event v_j. The goal is to find the most simple representation of a graph G that represents the system behavior as observed by the input system trace

A. Workflow Graph Construction

The core concept in our approach is to find the minimum-sized graph that is still able to fully explain the observed data. Ignoring the implementation details for later, let us first consider this idea in more depth by proposing that an algorithm exist which takes system traces as input, and then returns a minimum-sized graph that still retains all the data in the observation as an output.

Our algorithm considers all the possible combination of trajectories, given the observational sequence. The algorithm will then return a graph that only has edges that correspond to the inter-vertex traversals found within this trajectory set and also uses the inter-event arrival time as the weight of the edges.

The concept where choosing the simplest solution to explain a data is likely the correct solution not a new concept. In fact, it has been utilized successfully in different types of the topology inference problem [20]. This principle, which is known as Occam's razor states that, "if presented with a choice between indifferent alternatives, then one ought to select the simplest one," and has been utilized successfully numerous times. The concept is a common theme in computer science and underlies several approaches in AI; e.g. hypothesis selection in decision trees and Bayesian classifiers.

The greedy workflow graph construction algorithm is detailed in Algorithm 1.

Algorithm 1: Greedy Approach for Workflow Graph Construction

```
Input:
A set of $n$ normal traces and threshold $\delta$
Output:
Graph $G$ with vertices as events and weighted edges as
inter-arrival times
Method: GConstruct for each trace $T$
Generate a dictionary matrix of all edges $V \times V$ where
  $V$ is all the vertices/unique events;
Process the trace & update the counter for each edge
  $e \in v_i \rightarrow v_j$ where $e \in E$ and $v_i, v_j \in V$;
For each edge $e \in v_i \rightarrow v_j$ update the time $min$, $min$,
  $average$, $variance$;
Return the normal workflow Graph $G = (V, E)$;
```

To illustrate the process outlined in Algorithm 1. Let us assume our trace has six unique events A, B, C, D, E, F. Each unique event is considered as a node and a set of 2 consecutive events is considered to form an edge. For example given a trace file with event A, B, C, D, E, F, then 'A', 'B', 'C', 'D', 'E', 'F' are considered nodes. Edges on the other hand will be [A,B], [B,C], [C,D], [D,E], [E,F]. The algorithm traverses through the entire trace file, taking note of each unique event and their subsequent timing information in a list. There are two dictionaries formed, one for nodes and one for each edge. Each node is assigned as a dictionary key and their corresponding dictionary values contain all possible edges connecting to the node with their timing information. The edge's corresponding value will be an array that stores all the timing information, namely their count (number of times the edge occurs in the trace) and statistical information for the inter-arrival time such as average, variance, minimum, and maximum. After finishing the trace, the result is further processed and the graph is pruned of some edges. The pruning method relies on a "threshold" value δ that can be tuned. This threshold will be the minimum amount of time that an edge needs to have occurred in the trace file to be considered in the final graph. It avoids having rare occurrences of event edges in the workflow graphs.

In the next section, we shall outline how our greedy graph is used for anomaly detection in system traces taking into consideration a sequence of event occurrence and inter-arrival time between the events. We first present our approach for offline anomaly detection followed by an online approach. Our graph construction technique has a time complexity of $O(n)$ where n is the length of the trace.

B. *Offline Anomaly Detection*

Anomalies are unavoidable in real-world systems, however, if they go undetected it may lead to catastrophic outcomes. In Figure 2 we present an overall workflow of our anomaly detection framework.

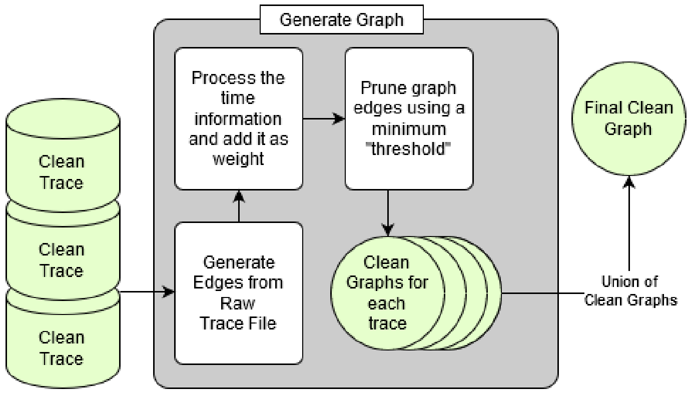
Generate Graph
Clean Trace
Clean Trace
Clean Trace
Process the time information and add it as weight
Prune graph edges using a minimum "threshold"
Generate Edges from Raw Trace File
Clean Graphs for each trace
Final Clean Graph
Union of Clean Graphs

Fig. 1. Workflow Graph Construction for System Traces

The anomaly detection framework requires trace as an input. It generates the workflow graph using the clean traces as input as outlined in Algorithm 1. The nodes denote the events in the trace and edge are weighted on the frequency of co-occurrence of the two connecting nodes and the inter-arrival times between the co-occurrence of the two associated events.

The graphs obtained for both clean and anomalous traces are used for anomaly detection. There are numerous measures that one could use to compare the two graphs to conclude if the trace belongs to the anomalous class or the clean class. In the offline approach, we use two metrics to compare the graph from traces to detect deviation from the normal. The first being a simple isomorphism check. It is assumed that a workflow graph from anomalous trace should not be isomorphic with clean graphs. However, as this is a fairly basic comparison, a more robust test is proposed in the form of Jaccard's similarity index. This index is a value that shows the similarities between two sets (graphs in this case) and will be a value between 0 and 1. The closer the index is to 1, the closer and more similar are the two graphs being compared. To obtain the Jaccard similarity index of the two graphs, first, take the intersection of the two graphs. Then divide the intersection with the union of the two graphs. Any graph with a Jaccard similarity index greater than the threshold $\Gamma = 0.9$ will be considered as a clean and anything else will be considered anomalous in the tests. The parameter Γ is a tunable parameter that can be tuned during the training process.

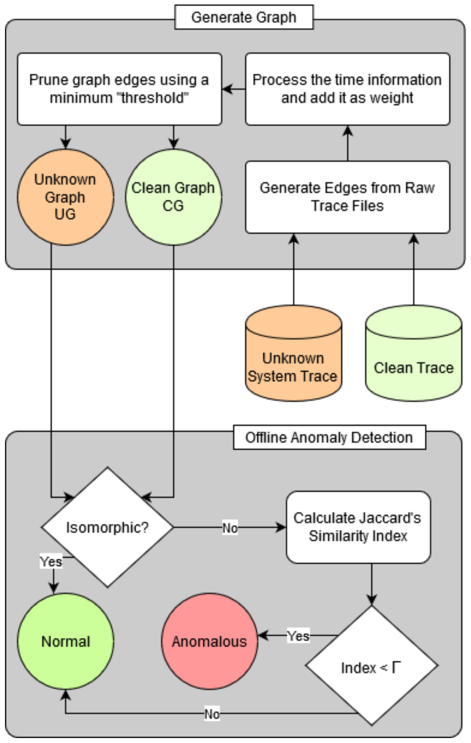
Generate Graph
Prune graph edges using a minimum "threshold"
Process the time information and add it as weight
Unknown Graph UG
Clean Graph CG
Generate Edges from Raw Trace Files
Unknown System Trace
Clean Trace
Offline Anomaly Detection
Isomorphic?
No
Calculate Jaccard's Similarity Index
Yes
Normal
Anomalous
Yes
Index < Γ
No

Fig. 2. Offline Anomaly Detection Workflow for System Traces

C. *Online Anomaly Detection*

Detecting anomalous behavior in traces as a whole is an offline approach that is generally employed for fault diagnosis and root cause analysis. However, in real-world situations, one would like to monitor the system continuously for malfunctions or abnormal behaviors. There is a need for an online anomaly detection framework.

We develop an online anomaly detection or monitoring framework as shown in Figure 3. In this case, we traverse the composed workflow graph from that is generated from the clean traces. As events arrive in the trace we traverse and monitor the behavior in real-time on two aspects. Firstly, if there exists a path from the current event node to the next event node and secondly if the inter-arrival time constraint is violated for any event transition. In this version, the detection will be done by checking whether or not the subsequent events is an edge that exists in the clean graph. The thresholds for flagging a given sequence of events as anomalous are derived during the training process where resilience parameters γ_e and γ_t are evaluated. Here γ_e is the threshold for identifying a trace as anomalous due to failure of identifying the correct sequence of events and γ_t is the threshold that signals the traces as anomalous given

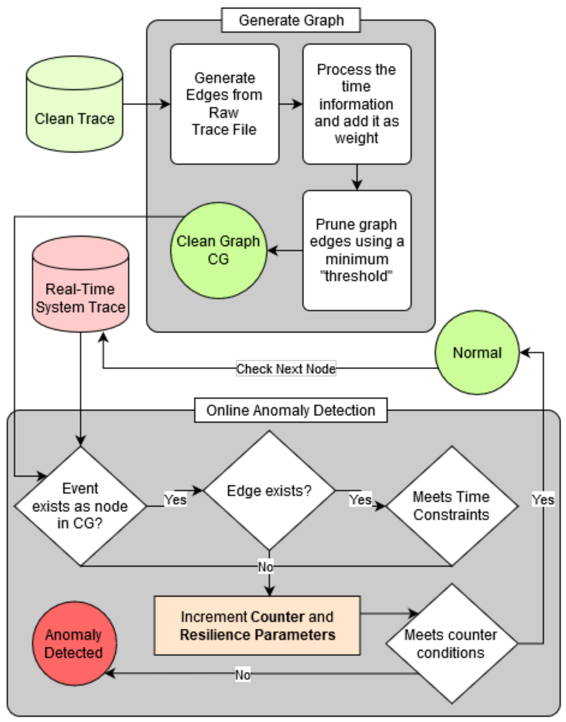
Generate Graph
Clean Trace
Generate Edges from Raw Trace File
Process the time information and add it as weight
Prune graph edges using a minimum "threshold"
Clean Graph CG
Real-Time System Trace
Normal
Check Next Node
Online Anomaly Detection
Event exists as node in CG?
Yes
Edge exists?
Yes
Meets Time Constraints
Yes
No
Increment Counter and Resilience Parameters
Meets counter conditions
No
Anomaly Detected

Fig. 3. Online Anomaly Detection Workflow for System Traces

the failure of inter-arrival times to meet the inter-arrival time bounds obtained during the training process.

In this case, we prioritize γ_e over γ_t where a failure to meet the sequence of events can have catastrophic outcomes leading to system failure whereas not able to meet the time constraint may be an indicator of precursor to system failure. This implies that by using a combination of the two parameters γ_e and γ_t, we can establish system resiliency. Continuous monitoring of these parameters can allow for preempting a system if one sees that these thresholds are being violated frequently. We show a detailed analysis for ensuring resilient systems in our case studies.

The pseudo-code for the online anomaly detection framework is shown. The information of the event is appended to an Anomaly Stack of selected size each time the two resilience parameters are exceeded. The parameter is tuned according to the situation and system.

The algorithm will iterate through pre-recorded system trace, taking each event as a "new" event logged in real-time, Simulating a real-time system. In the following pseudo-code provided, the cleanGraph is the graph obtained from clean non-anomalous traces, finalGraph is the graph returned and formed by the unknown trace, unknownTrace. To keep track of the number of times an edge did not meet the resilience parameters, a stack named anomalyStack is used.

Algorithm 2: Online Anomaly Detection.

```
Output:
cleanGraph; finalGraph;
unknownTrace[];
anomalyStack; $\gamma_1$; $\gamma_2$;
for event in unknownTrace do
    if $\gamma_1$==100 or $\gamma_2$==5 or anomalyStack == 100
     then
        Anomaly detected;
    else
        if first event and exists in cleanGraph then
            add event to cleanGraph;
            continue;
        else

        end
        if event exists in cleanGraph edge = edge of
        current and previous events. if edge in
        cleanGraph then
            if time withing min and max then
                add egde to finalGraph;
            else
                $\gamma_1 ++$;
                add edge to anomalyStack;
            end
        else
            $\gamma_2 ++$;
            add edge to anomalyStack;
        end
        add edge to anomalyStack;
    end
end
return finalGraph;
```

IV. Experimental Evaluation

In this section, we use event traces generated from deployed real-time systems of two industrial case studies to show the feasibility of the framework proposed in Section III.

UAV Case Study

We use data, that is the kernel event traces, from an unmanned aerial vehicle (UAV) that is running a real-time operating system QNX Neutrino 6.4. This UAV has received the Special Flight Operating Certificate (SFOC) after the University of Waterloo finished developing it, and has been utilized in real mapping and payload-drop missions in Ontario and Nova Scotia. The trace snippet in Figure 4 shows the sample trace used in our experiments.

The snippet is generated using the *tracelogger* and *traceprinter* utilities available in QNX Neutrino real-time oper-

```
259018352791,13,INT_ENTR::0x00000044
259018354208,14,INT_HANDLER_ENTR::0x00000044
259018357541,15,INT_HANDLER_EXIT::0x00000044
259018368666,18,COMMSND_PULSE_EXE
259018370333,18,COMMSND_PULSE_EXE
259018371583,18,COMMSND_PULSE_EXE
...
259567335041,22,COMMREC_MESSAGE
259567336541,23,KER_EXITMSG_RECEIVEV/14
259567355458,13,INT_ENTR::0x00000044
259567366416,14,INT_HANDLER_ENTR::0x00000044
259567381750,15,INT_HANDLER_EXIT::0x00000044
259567384916,16,INT_EXIT::0x00000044
259567403625,25,COMMREPLY_MESSAGE
```

Fig. 4. Trace snippet from QNX *tracelogger*

ating system. In the experiments, an event type has a unique value obtained by combining process class *pclass* and process name `pname`. The dataset has 15 UAV traces and each is composed of a stream of roughly 1 million events. There are four anomalous execution scenarios.

- One scenario is where a task is implemented where it tries to interfere with system tasks by hogging CPU resources. It does this by running while-loops in the background.
- Two scenarios is to simulate system behaviour when jobs are being executed while two different scheduling algorithms are being used to schedule the different tasks(e.g, FIFO and sporadic scheduling).
- One scenario is the behavior of a regular execution however, the regular behaviour here is different from the training traces.

We constructed individual graphs for all clean traces using the Algorithm 1. Once we obtain all the individual graphs, we perform a union of all the graphs of clean traces to form a final graph. Then, we conduct two experiments for both offline and online anomaly detection as described below:

Offline Anomaly Detection Experiments

We generated the workflow graph for each individual clean trace that was then composed together to form a single workflow graph for all the clean traces. It included all the nodes and edges. The inter-arrival times for each pair of events was the maximum interval that was found in the training instances. To classify a given trace as anomalous or not, the test trace is subjected to Algorithm 1 to obtain its workflow graph. Once the workflow graph is obtained its similarity to the composed workflow graph from the clean traces is evaluated. We compare two measures to classify a trace as anomalous or not by comparing its workflow graph. For evaluating the approach of graph creation presented in this paper, we decided to vary the graph pruning threshold, δ to see its impact on similarity with clean traces and anomalous traces. We present average and standard deviation of the Jaccard's similarity index across all clean traces as J_μ and J_σ in Table I and with anomalous traces in Table II.

δ	J_μ	J_σ
0.0001	0.955	0.031
0.0005	0.956	0.029
0.001	0.982	0.021
0.005	0.990	0.007
0.01	0.980	0.024

TABLE I
SUMMARY OF JACCARD SIMILARITY INDEX FOR CLEAN TRACES

δ	J_μ	J_σ
0.0001	0.704	0.140
0.0005	0.697	0.178
0.001	0.716	0.208
0.005	0.711	0.117
0.01	0.703	0.071

TABLE II
SUMMARY OF JACCARD SIMILARITY INDEX FOR ANOMALOUS TRACES

The results obtained from our industry strength experiments are assuring since the approach we employed classified anomalous traces with 100% accuracy. The workflow graph also provides a platform for further root cause analysis and fault diagnosis.

Model	Accuracy	Explainability
Markov Chains	99%	Partial
Feedforward Neural Networks	97.3%	No
Bayesian Network	94%	Partial
LSTM	96.5%	No
Random Forests	100%	Yes
GWAD	100%	Yes

TABLE III
COMPARISON WITH STATE-OF-THE-ART-TECHNIQUES

In Table III we present a comparison between GWAD and other state-of-the-art techniques on the same dataset. Here, it is clear that the approach presented in this paper behaves either at-par, or better than the state of the art with the advantage that it is simple, light-weight, and provides ground for explainability or root cause analysis that is lacking in most complex models.

Streaming Anomaly Detection Experiments

In the context of online streaming anomaly detection, it is critical that we analyze system behavior in real-time and raise flags ahead of anomalous events or during an anomalous event. Again, in safety-critical real-time systems, it is critical to observe that aberrations from normal behavior could occur due to abnormal sequence of event occurrence or failure to meet the time constraints. To encompass both these properties in our runtime anomaly detection approach we use two resiliency parameters γ_e and γ_t that allow us to monitor the behavior in real-time. It may be noted that bounds for both γ_e and γ_t are obtained during the training phase or from clean traces. γ_e denotes the maximum number of consecutive events that did not meet the pairwise event co-occurrence sequentially as learned by the workflow graph from the normal traces. γ_t denotes the maximum number of times the pairwise event co-occurrence sequentially was correct but the inter-arrival time for the pair of event co-occurrence was not met. We keep a track of the maximum sequential count for event co-occurrences that were expected but did not occur and the same for the inter-arrival time bounds of non-anomalous traces. If an abnormal trace goes beyond the γ_e and γ_t, we raise a flag denoting the system is undergoing abnormal behavior. It was observed that $\gamma_e = 184$ and $\gamma_t = 2$ were obtained from clean traces. All anomalous traces exhibited values for $\gamma_e = 204$ and $\gamma_t = 3$ respectively that have a sufficient margin to identify anomalous behavior. We also experimented with other regular expression based monitoring and it was observed that such subtle behavioral changes are missed with standard approaches. Therefore, it reinforces the need and use of our approach.

V. Conclusions and Future Work

The power of workflow graphs has been demonstrated in the context of real-time systems. We utilized workflow graph generation in system traces with discrete events to our advantage, when used with quantifiable metrics they allow us to reason the system behavior. We integrated workflow graph construction in system traces into an anomaly detection framework in both offline and online settings. The experiments validate our work using industrial case studies in a real-time safety-critical application domain. This work can be extended to extract system properties in the form of a timed regular expression that will enable a more expressive formulation for understanding system behavior. Furthermore, further work can be put into the framework itself, namely in generating the resilience parameter for our online detection framework. In order to produce a more concrete and robust result, a more detailed methodology needs to be created in order to fine tune these parameters. These parameters directly correlate with the accuracy of the framework and any further work that can improve this will greatly improve the accuracy of the framework.

Acknowledgment

The authors would like to thank Prof. Sebastian Fischmeister, Electrical and Computer Engineering, the University of Waterloo for providing datasets for use in our work.

References

[1] A. D. Pimentel, L. O. Hertzbetger, P. Lieverse, P. van der Wolf, and E. E. Deprettere, "Exploring embedded-systems architectures with artemis," *Computer*, vol. 34, no. 11, pp. 57–63, 2001.

[2] F. Y. Edgeworth, "Xxii. on a new method of reducing observations relating to several quantities," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 25, no. 154, pp. 184–191, 1888.

[3] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, 2009. [Online]. Available: https://doi.org/10.1145/1541880.1541882

[4] Wenke Lee and Dong Xiang, "Information-theoretic measures for anomaly detection," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001*, 2001, pp. 130–143.

[5] C. C. Noble and D. J. Cook, "Graph-based anomaly detection," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: Association for Computing Machinery, 2003, p. 631–636. [Online]. Available: https://doi.org/10.1145/956750.956831

[6] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Network anomaly detection: Methods, systems and tools," *IEEE Communications Surveys Tutorials*, vol. 16, no. 1, pp. 303–336, 2014.

[7] H. Daoud and M. R. Dagenais, "Recovering disk storage metrics from low-level trace events," *Software: Practice and Experience*, vol. 48, no. 5, pp. 1019–1041, 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2566

[8] L. Bao, Q. Li, P. Lu, J. Lu, T. Ruan, and K. Zhang, "Execution anomaly detection in large-scale systems through console log analysis," *Journal of Systems and Software*, vol. 143, pp. 172 – 186, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121218301031

[9] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, and H. Cai, "Localizing root causes of performance anomalies in cloud computing systems by analyzing request trace logs," *Science China Information Sciences*, vol. 55, no. 12, pp. 2757–2773, Dec 2012.

[10] S. Pauwels and T. Calders, "Detecting anomalies in hybrid business process logs," *SIGAPP Appl. Comput. Rev.*, vol. 19, no. 2, p. 18–30, Aug. 2019. [Online]. Available: https://doi-org.ezproxy.library.ubc.ca/10.1145/3357385.3357387

[11] J. Qiu, Q. Du, K. Yin, S.-L. Zhang, and C. Qian, "A causality mining and knowledge graph based method of root cause diagnosis for performance anomaly in cloud applications," p. 2166, 2020.

[12] S. Lu and R. Lysecky, "Time and sequence integrated runtime anomaly detection for embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 2, Dec. 2017. [Online]. Available: https://doi-org.ezproxy.library.ubc.ca/10.1145/3122785

[13] T. B. Callo Arias, P. van der Spek, and P. Avgeriou, "A practice-driven systematic review of dependency analysis solutions," *Empirical Software Engineering*, vol. 16, no. 5, pp. 544–586, Oct 2011. [Online]. Available: https://doi.org/10.1007/s10664-011-9158-8

[14] L. Akoglu, H. Tong, and D. Koutra, "Graph based anomaly detection and description: a survey," *Data Mining and Knowledge Discovery*, vol. 29, no. 3, pp. 626–688, 05 2015, copyright - The Author(s) 2015; Last updated - 2015-04-15. [Online]. Available: http://ezproxy.library.ubc.ca/login?url=https://search-proquest-com.ezproxy.library.ubc.ca/docview/1671986357?accountid=14656

[15] J. Han, H. Cheng, X. Dong, and X. Yan, "Frequent pattern mining: current status and future directions," *Data Mining and Knowledge Discovery*, vol. 15, no. 1, pp. 55–86, 08 2007, copyright - Springer Science+Business Media, LLC 2007; Last updated - 2014-08-30.

[16] T. Lane and C. E. Brodley, "Temporal sequence learning and data reduction for anomaly detection," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 3, p. 295–331, Aug. 1999. [Online]. Available: https://doi-org.ezproxy.library.ubc.ca/10.1145/322510.322526

[17] A. Narayan, N. Benann, and S. Fischmeister, "Mining specifications using nested words," in *2017 6th International Workshop on Software Mining (SoftwareMining)*, 2017, pp. 9–16.

[18] A. Narayan and S. Fischmeister, "Mining time for timed regular specifications," in *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, 2019, pp. 63–69.

[19] A. Narayan, G. Cutulenco, Y. Joshi, and S. Fischmeister, "Mining timed regular specifications from system traces," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 2, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3147660

[20] D. Marinakis and G. Dudek, "Topological mapping through distributed, passive sensors." in *IJCAI*, 2007, pp. 2147–2152.