CAML: Machine Learning-based Predictable, System-Level Anomaly Detection *

Jiguo Song*, Gerald Fry†, Curt Wu†, Gabriel Parmer*
*The George Washington University
{jiguos, gparmer}@gwu.edu
† Charles River Analytics
{gfry, cwu}@cra.com

***Abstract*—Security challenges are increasing in distributed cyber-physical systems (CPSs), which integrate computation and physical processes. System security is complicated by both the temporal and safety constraints of CPSs. In this paper, we investigate the potential for using system-level anomaly detection in a component-based RTOS to detect system compromises and aberrant behavior. We investigate a machine learning-based anomaly detection framework, CAML, which monitors for and identifies cyber attacks in system-level services within bounded time. We leverage past work in system fault recovery to predictably recover the system to an uncompromised state. We also evaluate the effectiveness of CAML in an avionics simulator-based CPS environment with injected cyber attacks. Our results and analysis indicate that CAML has promise to effectively enhance CPS robustness by securing the underlying RTOS against system-level cyber attacks with only small performance degradation.**

I. Introduction

Cyber-physical systems (CPSs) are integral to many domains such as health care, military, and industrial control. Due to strict timing constraints and limited resources, they must carefully balance between cost, size, weight, energy and safety. Threat and vulnerability mitigation is emerging as an extremely important concern regarding CPS design, as these systems often interact with the physical world through the integration of computation, actuation, sensing, and physical processes. The threats in CPSs are not only from the design errors or software bugs in the increasing complexity of system, but can also be from malicious attacks against the system by attackers who intentionally want to create chaos. Much research [1][2] has been done in the past to address different kinds of cyber-attacks, including deception attacks, *DoS* attacks, and even physical attacks against the actuator or plant. It is important for safety critical CPSs to continue to function even under the presence of malicious attacks. CPSs also must often meet *temporal* constraints and defend against cyber-attacks that can cause deadline misses to maintain the system correctness (i.e., avoid unbounded priority inversion). An intrusion detection system (IDS) is an effective way to detect the malicious attacks in the embedded system and respond without missing deadlines, before the control system is affected leading to physical system failure.

Unfortunately, the security of the system can also be compromised due to vulnerabilities within the underlying real-time operating system (RTOS), on which the application-specific components of the control system typically rely. For example, if a malicious attacker compromises the RTOS scheduler, erroneous behavior in the control system might lead to physical system failure (e.g., a vehicle could lose control of its brakes and fail to stop before an accident occurs). In this paper, we focus on how to detect and recover from cyber-attacks in low-level system services by the malicious adversary, while continuing to maintain operationally correct system behavior within bounded time. The contributions in this work include:

- a machine learning-based, predictable anomaly detection framework for cyber-attacks in system-level services, and
- an evaluation of the effectiveness of fault detection and recovery using a flight simulator under the presence of injected attacks in a component-based RTOS.

II. Threat Model

A typical cyber-physical system (CPS) consists of physical, control, actuation and sensing subsystems. The computing stage in the control system performs processing on the collected data and calculates the control commands that can be sent to the actuators to manipulate the physical system. A Real-Time Operating System (RTOS), on which the application (e.g., a PID controller) relies, is often used in the computing stage. Figure 1 shows a CPS that uses a component-based RTOS in its computing stage, which contains system-level components such as the scheduler, memory management, file system (FS), synchronization, event management, and timer. The PID controller, as an application component, communicates with the physical system through the network component (e.g., processes sensor data and sends control signals to a remote avionics system).

A malicious adversary can use various techniques to attack the CPS at any stage. For example, the adversary might intercept the information over the communication channel, compromise the key, forge messages to the operator, or prevent normal requests from being processed, causing Denial of Service (*DoS*). The adversary can also attack services within the RTOS. For example, in Figure 1, a PID controller periodically receives incoming data from the sensing subsystem and needs to access the FS to log the information, before it can send the computed control signal to the remote avionics system. A malicious attacker, *Noise* (shown in the figure as a red demon), could compromise the FS, resulting in corrupted information, or could delay the delivery of actuation control commands. For example, *Noise* could delay the execution in the FS (e.g., by executing an infinite loop) to prevent other

*This material is based upon work supported by the National Science Foundation under Grant No. CNS 1149675, ONR Award No. N00014-14-1-0386, and ONR STTR N00014-15-P-1182. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or ONR.

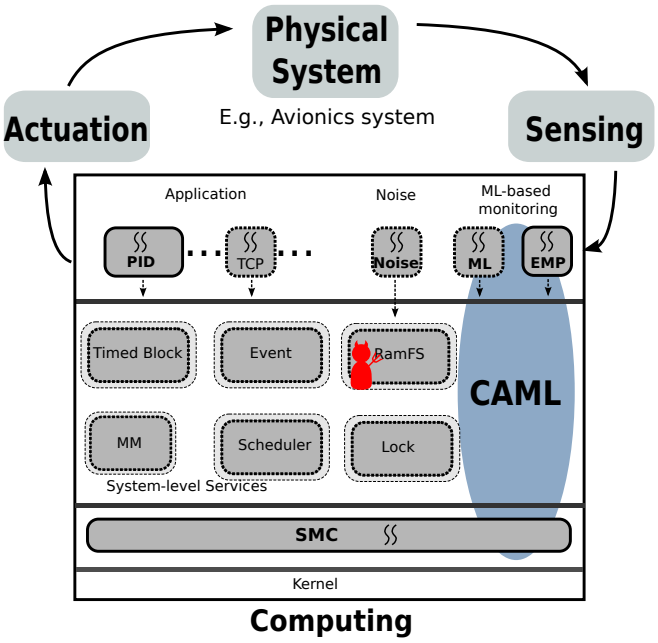

Fig. 1: A CPS control system that consists of a component-based RTOS and a PID controller application. CAML infrastructure is shown in the blue oval area and a red demon represents the malicious adversary.

applications from accessing the FS service, causing deadline misses. The attacks in system-level services could eventually cause the physical system to deviate from its expected behavior, which could result in a catastrophic failure of the safety-critical system. To address this problem, we propose CAML, a machine learning-based intrusion detection infrastructure that focuses on predictably detecting and recovering from system-level anomalies due to cyber-attacks in a component-based RTOS. CAML monitors the system control flow and timing behavior using extensions to our previous work on C'MON [3] (Section IV), uses machine learning algorithms to detect anomalies (Section III), and uses C^3 [4] techniques to predictably recover system-level services (Section IV). Though our initial analysis focuses on attacks that affect timing behavior, potentially leading to physical system failure (e.g., an aircraft crash), future work will expand the detection capability of CAML for other types of security threats.

III. MACHINE LEARNING ALGORITHMS

In this section, we describe two unsupervised learning algorithms used for CAML to identify system anomalous behavior: algorithmic sequence learning (ASL) and a density-based clustering algorithm, density-based spatial clustering of applications with noise (DBSCAN).

Algorithmic Sequence Learning. The ASL algorithm is based on the episode mining technique in [5]. ASL can be applied to temporal, sequentially ordered datasets to predict events or describe frequent patterns. ASL identifies patterns in sequences based on frequency and consistency by examining all sub-sequences of events within a specified window and enumerating associative IF-THEN rules that map *antecedents* to *consequents* (i.e., sub-sequences of events that are likely to co-occur). The algorithm calculates the *confidence* (probability) that the sequence identified by the THEN-clause will follow the observation of the sub-sequence identified by the IF-clause. For example, in a component-based RTOS, if the sequence of components that a thread invokes follows a pattern, then ASL generates rules that predict that sequence, given an observed subsequence. If a component is compromised (e.g., an attacker causes the FS component in the sequence to invoke an additional component before returning), then these rules will be violated. Parameters to the rule generation algorithm include minimum support and minimum confidence thresholds. These parameters affect the number of resulting rules and their confidence. Minimum support specifies the minimum number of times a pattern must occur for it to be of interest, and minimum confidence specifies the minimum percentage of the time that the consequent is observed given that the antecedent is observed. ASL generates association rules which can then be used as conditions that incoming events are tested against. If an event sequence is recorded that matches an antecedent, but the events that follow do not match the rule's consequent, then we flag this as a potential anomaly. The likelihood that this sequence of events is actually anomalous is proportional to the confidence level associated with the rule: the greater the confidence of the rule, the stronger the expectation that it will hold, and the greater the likelihood that a deviation represents a significant anomaly. The frequency of false positives can be controlled by only selecting rules that meet a certain minimum confidence level; this avoids problematic situations where, for example, a rule that holds only 60% of the time (has relatively low confidence) would identify anomalies the remaining 40% of the time. We extended the original ASL algorithm to detect novel events. Our updated ASL algorithm adds a rule whenever a new event is detected that includes a novel event. For example, a thread invokes the FS component for the first time. These novel rules are somewhat different from the standard rules because they are violated immediately; however they are still subject to the same thresholds (i.e., frequency and support), so multiple violations may be required to trigger an anomaly detection. We call them "null rules" because the rule would take the form of $\{X\}$ implies $\{null\}$, where X is a novel event. The "null rules" ensure that we have a way to track novel events and trigger notifications when the frequency and support of novel events exceeds thresholds.

Density-based Spatial Clustering of Applications w/ Noise. Another algorithm, DBSCAN, finds clusters based on how densely packed the data points (i.e., events like component invocations with time stamps) are [6], and we have successfully applied these algorithms to detect anomalous behavior in sensor networks. One key advantage of density-based algorithms over a more traditional k-means clustering approach is that the number of clusters does not need to be defined a priori. A cluster is also defined by local relationships between data points, so its overall shape can be quite irregular provided its constituent data points satisfy the density requirements. Density-based clustering can be applied to continuous multi-dimensional data and is especially well-suited for data that

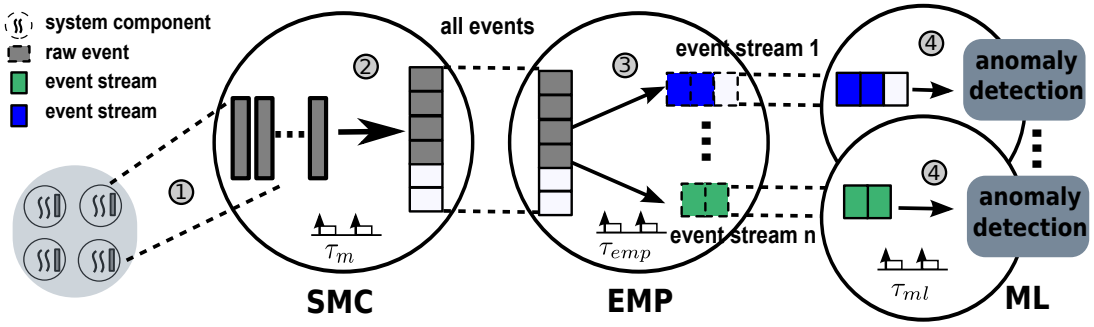

Fig. 2: CAML system-level anomaly detection infrastructure. SMC and τ_m are the component and task for logging events. EMP and τ_{emp} are the component and task for multiplexing events. ML and τ_{ml} are the component and task that run anomaly detection machine learning algorithms.

does not conform to a regular distribution. For example, the execution time of different threads in a specific component can form a cluster even if the time does not follow any known distribution. There are two primary parameters in the algorithm can be defined by the user: ε, which defines the neighborhood radius around a point for grouping points into a cluster, and $minPts$, which defines the minimum number of points to form a cluster. In the detection model, data points that cannot be classified into any existing cluster are labeled as anomalies. DBSCAN checks each event (data point), and if more than $minPts$ exist within a ε-neighborhood, then a normal cluster can grow by collecting reachable points. After all data points are considered, the events that could not be assigned to any cluster will be treated as anomalous events.

IV. System Design

In this section, we will first briefly review Composite and C'Mon [7], upon which CAML is built. Then we will discuss the CAML infrastructure.

A. Composite *and* C'Mon *background*

CAML is built on top of the Composite component-based OS. Components in Composite are user-level, hardware isolated (via page-tables) code and data that implement some functionality and export an interface of functions through which other components can harness that functionality. Components implement system policy and low-level services such as scheduling, physical memory management and mapping, synchronization, and I/O management, as shown in Figure 1. C'Mon [7] is a system-level latent fault monitor located between the kernel and rest of system components. It tracks system-level communication events and timing, such as invocations (IPC) between components, interrupts and thread dispatching. By harnessing the event logging interface code, C'Mon enables monitoring of all interactions between each component and the rest of the system beyond.

B. CAML *infrastructure*

CAML extends the C'Mon event logging infrastructure and focuses on more general system-level anomaly detection by incorporating additional event pre-processing and machine learning techniques. As Figure 2 shows, CAML consists of three components: the event logging component, SMC, the event multiplexing component, EMP, and the machine learning component(s), ML.

SMC component. The System Monitor Component (SMC) tracks and logs system events. It does so by maintaining per-component shared ring buffers with all other system-level components. As ① shows, when the system executes and components communicate, the communication events are published into these buffers as *raw events* that include the event type, the cycle-accurate time stamp, and thread and component information. A task, τ_m, in the SMC then copies all *raw events* into a large buffer shared between the SMC and EMP components (as ② shows). Copying events happens either periodically or when any per-component shared buffer is full (defined as the time-trigger activation and buffer-trigger activation in [7]). The EMP component pre-processes and aggregates *raw events* into more abstract typed event streams that are fed into different machine learning algorithms. EMP-based event pre-processing logic reduces the code complexity and the memory footprint of the SMC component. It also avoids running complex computation at a high priority level.

EMP component. A task, τ_{emp}, executing in the EMP component is responsible for multiplexing the *raw events* in the large shared buffer. The task, τ_{emp}, periodically consumes and processes the raw events in the following steps: (1) examine the raw event; (2) extract the desired information (i.e., thread and component identifier, order and location of the events, etc.); and (3) aggregate the events into multiple, different *event streams*. Event streams are categorized as per-thread or per-component, and they provide an abstraction of the system behavior and its execution characteristics. For example, an event stream can be a sequence of component invocations and return actions with timing information for a particular thread. An event stream can also be specified within a time window of fixed length. For example, a stream could provide the number of times that any thread is interrupted in a specific component within a period of time. Event streams will be copied into per-stream buffers shared between the EMP component and a set of ML components, and are eventually used as the training data for machine learning algorithms (as ③ shows).

ML components. A periodic task, τ_{ml}, runs a machine learning algorithm to detect anomalies (as ④ shows). Note that different event streams can be directed to the different

ML components. To learn the pattern, τ_{ml} extracts the high-level system execution features from event streams and builds a set of rules. The violation of the rules will be identified as anomalies in the system using machine learning algorithms. The execution of the tasks, τ_{emp} and τ_{ml}, could be incorrectly scheduled or delayed if an anomaly is presented in the scheduler. Therefore, τ_m must ensure that τ_{emp} and τ_{ml} can be activated correctly and on time. CAML supports recovering the affected system service (e.g., faulty scheduler, memory manager, and/or FS) using C^3 [8]. One challenge is to accurately localize the faulty component before the proper response can be made (e.g., recover the affected service or quarantine the malicious user). The localization process must determine which system service is actually affected and the identification of the faulty service must be as accurate as possible to reduce the false positives/negatives and their impact on system-wide schedulability. A detailed approach to fault localization is a work in progress.

V. System Timing Analysis

In this section, we will present the response time analysis for CAML and show that CAML can effectively detect and recover from system-level anomalies without missing deadlines.

A. Response Time Analysis (RTA) Model

The tasks, τ_m, τ_{emp} and τ_{ml} are responsible for logging events, multiplexing events, and running the machine learning algorithm, respectively. The tasks' periodicities are denoted as p_m, p_{emp}, and p_{ml}, and the tasks' WCETs are e_m, e_{emp}, and e_{ml} respectively. Note that $p_m < p_{emp} < p_{ml} <$ periodicity of other tasks in the system. We define $B_{smc \Rightarrow emp}$ as the shared buffer between the SMC and EMP components, and $B_{emp \Rightarrow ml}$ as the shared buffer between the EMP and ML components. When a per-component event logging buffer is full, an invocation must be made to the SMC component to start consuming the *raw events*. The worst-case cost of this activation, INV^m, and the maximum number of such activations, M_m, are both defined as in [7]. The overall structure of the RTA is based on a recurrence that finds a fixed point less than the task's deadline. If no fixed point is found, the task is not schedulable:

$$R_i^{n+1} = R_i^n(RTA) + R_i^n(C^3) + R_i^n(MON) + R_i^n(EMP) + R_i^n(ML) + R_i^n(F) \quad (1)$$

where $R_i^n(RTA)$ is the traditional response-time analysis [9] and $R_i^n(C^3)$ is the contribution from the C^3 fault recovery [8]. The contribution from τ_m logging and copying events in the SMC component is given as

$$R_i^n(MON) = \left\lceil \frac{R_i^n}{p_m} \right\rceil (M_m \times INV^m + M_{emp} \times INV^m + e_{copy}^{m \rightarrow emp}) \quad (2)$$

where $M_m \times INV^m$ is due to the buffer-trigger activation [7], and M_{emp} is defined as $\left\lceil \frac{\sum_{\forall x} B_{nosync}^x}{B_{smc \Rightarrow emp}} \right\rceil - 1$. B_{nosync}^x is the buffer for a component, c^x, for which no synchronous buffer-triggered activation could arise. The term, $M_{emp} \times INV^m$, represents when the $B_{smc \Rightarrow emp}$ is full, in which case we must switch to τ_{emp} and let it consume the events. The last term, $e_{copy}^{m \rightarrow emp}$, is the total overhead of copying events from the SMC per-component buffers to the buffer $B_{smc \Rightarrow emp}$ within p_m. This overhead is proportional to the number of events generated in p_m. The contribution due to τ_{emp} multiplexing events in the EMP component is given as

$$R_i^n(EMP) = \left\lceil \frac{R_i^n}{p_{emp}} \right\rceil e_{emp} \quad (3)$$

where e_{emp} includes the overhead of processing events and creating event streams. This is proportional to the number of events generated within p_{emp}. The contribution due to τ_{ml} running the machine learning algorithm is given as

$$R_i^n(ML) = \left\lceil \frac{R_i^n}{p_{ml}} \right\rceil e_{ml} \quad (4)$$

where e_{ml} is the total overhead of consuming event streams and detecting the anomalies. Though the the size of the $B_{emp \Rightarrow ml}$ could affect the system schedulability, to make the analysis simple we make an **assumption**: the size of the $B_{emp \Rightarrow ml}$ is sufficient for holding all streamed events. This is a reasonable assumption given the event streams contain the aggregated information such as the thread WCET in the component, the execution time since last thread activation, and component invocations. Compared to tracing system-wide events, the aggregated information requires much less memory space. The contribution due to anomaly localization and wasted computation is given as

$$R_i^n(F) = \left\lceil \frac{R_i^n}{p_{ft}} \right\rceil (e_{ml}(localization) + p_{ml} + w^f) \quad (5)$$

where $e_{ml}(localization)$ is the fault localization overhead. The sum of p_{ml} and w^f is the wasted time due to the anomaly since there is a period of computation within the faulty component which cannot be trusted. This wasted computation[1] is the period of the machine learning task, p_{ml}, plus the WCET of the system-level component, w^f.

B. Schedulability Evaluation

We conducted schedulability evaluation in a system with 25 components and 50 tasks with average periodicity of 100 ms and an attack that occurs every 500 ms. Utilization and schedulability are both represented as percentages. The memory constraint is relaxed by assuming that there is enough memory to hold all logged and streamed events.

Figure 3 shows how the CAML infrastructure affects system schedulability while varying the *event rate* (evts/ms), which is defined as the total number of events that occur in one millisecond. The green line is the reference system without CAML. The red, blue, and black lines show the schedulability of the system with event rates of 200 evts/ms, 400 evts/ms and 800 evts/ms, respectively. A higher event rate has more impact

[1]We assume that the anomaly can be detected within one p_{ml}. To relax this assumpation, Eq.5 must be modified to accommodate the situation in which multiple p_{ml} are required for detection.

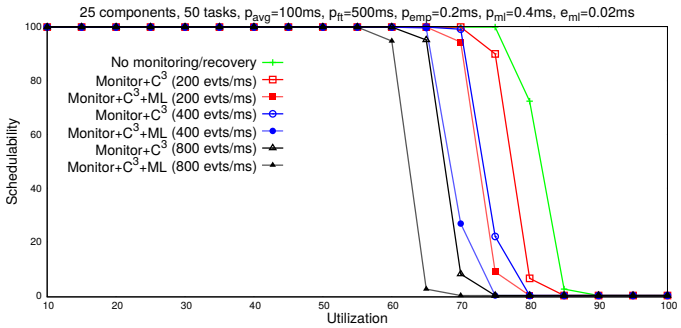

Fig. 3: Schedulability vs Utilization

on the system schedulability. The CAML infrastructure has more impact on the system schedulability compared to the system with only C'MON and C^3, due to the machine learning task, τ_{ml}.

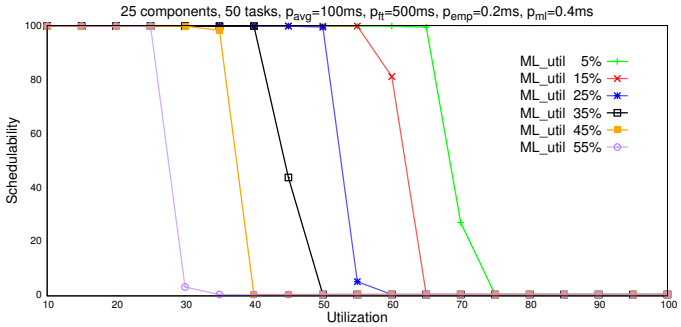

Fig. 4: ML task utilization impact on schedulability

Figure 4 presents how machine learning task τ_{ml} affects the system schedulability. The WCET of e_{ml} is proportional to the number of event streams. In Figure 4, the event rate is fixed at 400 evts/ms, and p_{ml} is fixed at 0.4 ms. It can be seen that the system running the task τ_{ml} with low utilization can achieve better schedulability even at a high task set utilization, which means that the machine learning algorithm must be efficient to ensure that the system is schedulable.

VI. Experiments

A. FlightGear Simulator

To simulate the CPS, we used a physics-based flight simulator, FlightGear (http:/www.flightgear.org/). FlightGear includes a visual interface and multiple flight dynamics models for realistic simulation of aircraft interacting with the physical environment. In our experiments, we simulate a Cessna 172P Skyhawk (1981 model) taking off from a landing strip with minimal pilot interaction. The simulation environment supports remote interaction with custom software communicating over network connections. We use this communication feature to implement an autopilot program that executes on a machine running our CAML framework on top of COMPOSITE on a Intel i7-2760QM (2.4 Ghz with only single core enabled). The autopilot program communicates over TCP connections with the remote simulator, periodically receiving messages at a rate of 3 Hz from the simulator containing data about the current status (e.g., heading) of the aircraft. Based on the information it receives, the autopilot sends commands back to the simulator to adjust the aileron positions of the aircraft to achieve and maintain a fixed target heading.

B. Experimental results

To analyze the effectiveness of the machine learning algorithms for anomaly detection, we tested them on the sequences of events that correspond to the observed sequences of component invocations by the autopilot thread. We generated rules based on event sequences under normal system behavior. We then applied these rules to the system under faulty behavior, during which we introduced a malicious call into the FS component that the component which implements the autopilot PID controller needs to access.

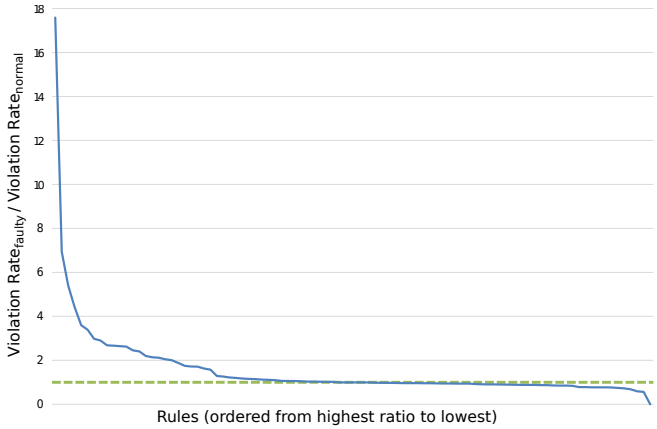

Fig. 5: Ratio of ASL rule violations

Figure 5 shows the ratio of rule violations in faulty data set versus normal data set when using ASL algorithm. The ASL algorithm was trained with a window size of 5, a minimum confidence value of 50%, and a minimum support value of 1%. With these parameters, 60 seconds of training data yielded 101 rules. An effective anomaly detection rule generates either significantly more or fewer violations in the faulty data set relative to the normal baseline. In the faulty data set, 25 of the 101 rules had violation rates more than 50% higher than the baseline. Also, missing from the figure are 8 rules were never triggered in the faulty scenario (i.e., the antecedent never appeared). Together 33 of the 101 rules can be used to detect the presence of an anomaly.

For DBSCAN, we ran the algorithm multiple times at different radius sizes to determine its effectiveness at anomaly detection. We trained it on 60 seconds of data with a minimum of 3 points required to form a cluster. We extracted two thread-based features: length of execution and time since last activation. The features were normalized across their ranges. We tested across 10 different radius values from 0.005 to 0.05 with normal and faulty test data where a fault was a failure of the file system.

DBSCAN cluster radius	Anomalies in normal scenario	Anomalies in faulty scenario	Anomaly rate increase
0.015	102	123	21
0.02	42	90	114
0.025	58	73	26

TABLE I: DBSCAN anomaly results

Table I shows the aggregate anomaly scores for each radius and scenario with the anomaly rate increases in the presence of fault. 7 of the 10 radii showed no statistical difference. The radii (0.015, 0.02, 0.025) showed some differences, and we repeated experiments with these three radii two more times to determine its effectiveness at anomaly detection.

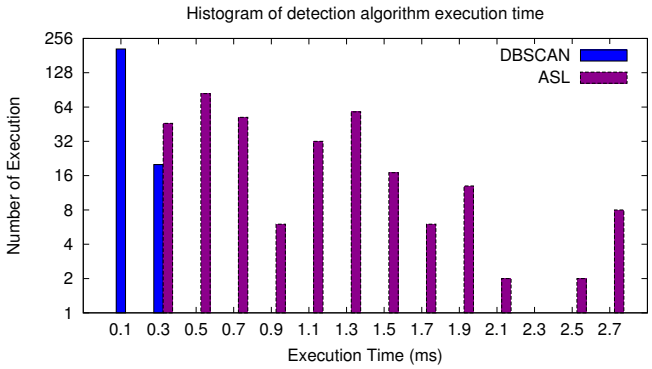

Fig. 6: Machine learning execution time comparison

To assess the use of the ML algorithms in a real-time system, **Figure 6** shows the histogram of how task τ_{ml}'s execution time varies for two different machine learning algorithms. The results show that the execution time of the DBSCAN algorithm is mostly distributed between 0.1∼0.3 ms, and the ASL algorithm is spread between 0.3∼2.7 ms.

	τ_m	τ_{emp}	τ_{ml}	τ_{noise}
period (ms)	110	130	170	610
exec time (ms)	event copy 0.00002	event process 0.0002	3 (ASL) 0.5 (DBSCAN)	500

TABLE II: Task period and measured execution time

Table II shows the periodicity of task τ_m, τ_{emp}, τ_{ml}, and τ_{noise} as configured in the experiment. We assume that $p_m < p_{emp} < p_{ml} < p_{noise}$. The table also shows the cost of copying an event into the shared buffer and the cost of processing an event. After it is activated, the attacker task, τ_{noise}, keeps spinning 500ms in the FS to cause temporal disturbance before it returns from the FS component.

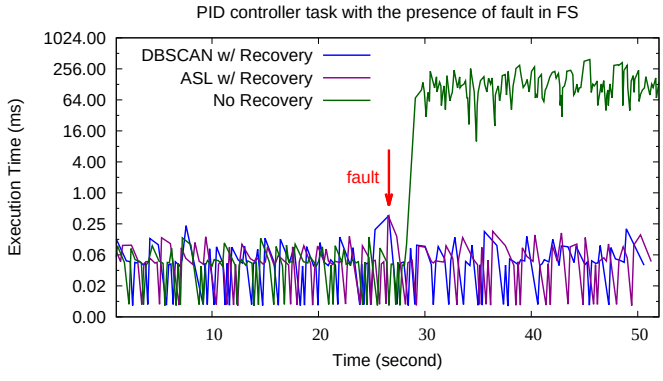

Fig. 7: PID controller with the injected attack

Figure 7 presents how the PID controller task, τ_{pid}, is affected under the attack and how CAML effectively detects and recovers the system with the setup from Section VI-A and the system parameters from Table II. The normal behavior of τ_{pid} is shown in Figure 7 from the beginning to 27s. The moment that τ_{noise} is activated and starts delaying itself by spinning 500 ms in the FS component is depicted as "fault" with a red arrow in Figure 7. The execution time of τ_{pid} increases quickly and results in unbounded priority inversion (shown as the green line) without proper detection and recovery. In contrast, the system with CAML successfully identifies the anomaly (e.g., the execution time in FS has deviated from its normal pattern and breaks the established rules). CAML recovers the FS from the attack and τ_{pid} quickly returns to its normal execution without missing deadlines.

VII. Related Work

Operating System Monitoring and Intrusion Detection. There has been much work done on OS monitoring in the past. For example, [10] [11] [12] are the tools to trace timing of execution within monolithic operating systems. OS intrusion detection through system call patterns has been studied in [13], [14], and virtual machine introspection for intrusion detection is studied in [15]. CAML differs from these related efforts in that it applies statistical learning methods to analyze system execution behavior in a component-based RTOS and focuses on enabling the system to be resilient to cyber attacks in low-level OS services without missing deadlines.

Machine learning-based anomaly detection. There have been several efforts that have applied software-based machine learning or statistical methods to detect anomalies and/or failures in general purpose and real-time systems. In [16], a probabilistic model-driven approach is used to detect intrusions in CPSs and to develop mitigating responses to malicious attacks. A sequence matching approach is used in [17] to detect anomalous user behavior in UNIX-based systems. The authors in [18] investigated a hardware implementations of a Support Vector Machine (SVM) and clustering algorithms for anomaly detection in NoC-based systems. The work most closely related to ours is [19], in which the authors proposed Real-time Calculus [20]-based inter-arrival curves and applied a semi-supervised sliding window-based classification technique on a sequence of events for anomaly detection. In contrast, CAML detects and recovers from cyber attacks on software components that perform OS functions with real-time constraints. The algorithms (ASL and DBSCAN) used for CAML are unsupervised. CAML extracts high-level features from the event traces with cycle-accurate time stamps, which allows the events (i.e., event streams for the thread or component) to be associated with the timing information for more effective anomaly detection.

VIII. Conclusions

In this paper, we presented CAML: a machine learning-based anomaly detection framework that predictably monitors, identifies, and recovers from cyber-attacks in a component-based RTOS. We evaluated the effectiveness of CAML using two unsupervised machine learning algorithms to detect injected attacks and showed that CAML shows promise for enhancing system resilience to cyber-attacks in system-level services without missing deadlines.

REFERENCES

[1] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo, "S3a: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems," HiCoNS '13.

[2] C. Liu, C. Yang, and Y. Shen, "Leveraging microarchitectural side channel information to efficiently enhance program control flow integrity," CODES '14.

[3] J. Song and G. Parmer, "C'mon: a predictable monitoring infrastructure for system-level latent fault detection and recovery," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[4] J. Song, J. Wittrock, and G. Parmer, "Predictable, efficient system-level fault tolerance in C^3," in *Proceedings of the 2013 34th IEEE Real-Time Systems Symposium (RTSS)*, 2013, pp. 21–32.

[5] H. Mannila, H. Toivonen, and A. Inkeri Verkamo, "Discovering frequent episodes in sequences," KDD '95.

[6] C. Braune, S. Besecke, and R. Kruse, "Density based clustering: Alternatives to dbscan," 2015.

[7] J. Song and G. Parmer, "C'MON: a predictable monitoring infrastructure for system-level latent fault detection and recovery," in *RTSS*, 2013.

[8] J. Song, J. Wittrock, and G. Parmer, "Predictable, efficient system-level fault tolerance in C^3," in *RTSS*, 2013.

[9] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, 1993.

[10] B. Brandenburg and J. Anderson, "Feather-trace: A light-weight event tracing toolkit," in *OSPERT*, 2007.

[11] T. Bird, "Measuring function duration with ftrace," in *Proceedings of the Linux Symposium*, 2009.

[12] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," ATEC '04.

[13] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," S&P '03.

[14] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," S&P '01.

[15] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," NDSS '03.

[16] R. Mitchell and R. Chen, "Effect of intrusion detection and response on reliability of cyber physical systems," *IEEE Transactions on Reliability*, 2013.

[17] T. Lane, C. E. Brodley *et al.*, "Sequence matching and learning in anomaly detection for computer security," in *AAAI Workshop: AI Approaches to Fraud Detection and Risk Management*, 1997.

[18] A. Kulkarni, Y. Pino, M. French, and T. Mohsenin, "Real-time anomaly detection framework for many-core router through machine learning techniques," *ACM Journal on Emerging Technologies in Computing (JETC)*, 2016.

[19] M. Salem, M. Crowley, and S. Fischmeister, "Anomaly detection using inter-arrival curves for real-time systems," ECRTS '16.

[20] S. Chakraborty, S. Kunzli, and L. Thiele, "A general framework for analysing system properties in platform-based embedded system designs," Date '03.