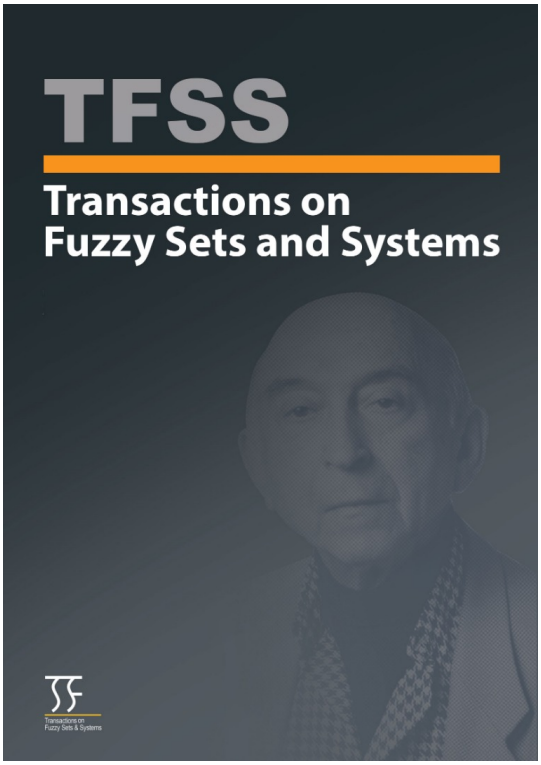
Transactions on Fuzzy Sets and Systems



Transactions on Fuzzy Sets and Systems

ISSN: **2821-0131**

https://sanad.iau.ir/journal/tfss/

A Stochastic-Process Methodology for Detecting Anomalies at Runtime in Embedded Systems

Vol.3, No.2, (2024), 142-171. DOI: https://doi.org/10.71602/tfss.2024.1183368

Author(s):

Alfredo Cuzzocrea, iDEA Lab, University of Calabria, Rende, Italy, &, Department of Computer Science, University of Paris City, Paris, France.
E-mail: alfredo.cuzzocrea@unical.it
Enzo Mumolo, Department of Engineering, University of Trieste, Trieste, Italy.
E-mail: mumolo@units.it
Islam Belmerabet, iDEA Lab, University of Calabria, Rende, Italy.
E-mail: ibelmerabet.idealab.unical@gmail.com
Abderraouf Hafsaoui, iDEA Lab, University of Calabria, Rende, Italy.
E-mail: ahafsaoui.idealab.unical@gmail.com

Transactions on Fuzzy Sets and Systems (TFSS)
URL: https://sanad.iau.ir/journal/tfss/
Online ISSN: 2821-0131
Vol.3, No.2, (2024), 142-171
DOI: https://doi.org/10.71602/tfss.2024.1183368

Article Type: Original Research Article

A Stochastic-Process Methodology for Detecting Anomalies at Runtime in Embedded Systems

Alfredo Cuzzocrea*, Enzo Mumolo, Islam Belmerabet, Abderraouf Hafsaoui

Abstract. *Embedded computing systems* are very vulnerable to *anomalies* that can occur during execution of deployed software. Anomalies can be due, for example, to faults, bugs or deadlocks during executions. These anomalies can have very dangerous consequences on the systems controlled by embedded computing devices. Embedded systems are designed to perform autonomously, i.e., without any human intervention, and thus the possibility of debugging an application to manage the anomaly is very difficult, if not impossible. Anomaly detection algorithms are the primary means of being aware of anomalous conditions. In this paper, we describe a novel approach for detecting an anomaly during the execution of one or more applications. The algorithm exploits the differences in the behavior of *memory reference sequences* generated during executions. *Memory reference sequences* are monitored in real-time using the *PIN tracing tool*. The memory reference sequence is divided into randomly-selected blocks and spectrally described with the *Discrete Cosine Transform* (DCT) [1]. Experimental analysis performed on various benchmarks shows very low error rates for the anomalies tested.

AMS Subject Classification 2020: 62H30; 62M02
Keywords and Phrases: Anomaly detection, Embedded systems, Stochastic processes, Inference models.

1 Introduction

Nowadays, *embedded computing systems* are extensively diffused and their *uses* include automotive applications, consumer applications and particular domains such as industrial subsystems or military applications. Embedded systems share some important properties, namely the fact that their failures often result in severe consequences (whose degree of gravity depends on the specific application), and interacting with them is difficult, if not impossible, and the number of concurrent executions is limited and frequently known in advance . Embedded system failures may be caused by software errors (bugs), faults, or the injection of new applications, including those deliberately designed to cause failures (malware), possibly coming from the network to which some embedded systems could be connected [2]. All of these events could result in *runtime anomalies*. The ability to automatically detect these anomalies may prevent failures in embedded systems and, hence, avoid damage to the controlled systems.

Anomalies are events that differ from some standard or reference events. They can be detected explicitly, i.e., through pattern recognition, which aims to classify patterns using a-priori knowledge or statistical information extracted from patterns [3, 4, 5]. *Anomaly detection* is a key application of Machine Learning, focusing on identifying data points that deviate from the norm and understanding why this occurs. Its uses

*Corresponding Author: Alfredo Cuzzocrea, Email: alfredo.cuzzocrea@unical.it, ORCID: 0000-0002-7104-6415
Received: 11 August 2024; **Revised**: 24 August 2024; **Accepted**: 31 August 2024; **Available Online**: 26 October 2024; **Published Online**: 7 November 2024.

How to cite: Cuzzocrea A, Mumolo E, Belmerabet I, Hafsaoui A. A stochastic-process methodology for detecting anomalies at runtime in embedded systems. *Transactions on Fuzzy Sets and Systems.* 2024; 3(2): 142-171. DOI: https://doi.org/10.71602/tfss.2024.1183368

are numerous, ranging from noise reduction and data cleaning to security-related tasks such as fault detection, fraud prevention, predictive maintenance, and social security.

Our anomaly detection technique establishes the behavior of the normal executions under examination, compares the observed behavior with the normal behavior, and signals when the observed behavior differs significantly from its normal profile. Since anomaly detection techniques signal all anomalies, false alarms are expected when anomalies are caused by behavioral irregularities. Therefore, this realizes a methodology based on *Stochastic Processes* (e.g., [6]).

Following these considerations, in this paper we propose a technique (and related algorithm) to build a profile of program behavior and to detect deviations from this profile. The profile is based on a *statistical model* of the *memory references* generated during the execution [7]. Our technique is designed to operate, for the detection phase, on embedded devices. Its computational complexity is low, and hence the overhead on the embedded device is limited. In particular, our prototypical implementation on an embedded device currently introduces an overhead lower than 35%. However, it can easily be speeded-up.

Our approach uses the *memory address sequences* generated by the applications during their execution, since these sequences contain a lot of information about the running applications. After an initial time period where the applications perform initialization tasks, we train, for each application, a *Hidden Markov Model* (HMM) of the execution. Then, we compute the likelihood that the sequences observed during the following execution are consistent with the HMM models, and we use this figure to detect the anomalies.

This paper significantly extends our previous conference paper [8], where we have first introduced the preliminary concepts of our research. Here, with respect to the previous paper, we made the following contributions:

- we provide a extended concepts on the methodology we proposed, along with a better organization and linkage with the results introduced in our original work;
- we provide a clearer description about the paper organization;
- we extend related work analysis, as to include other emerging initiatives dealing with the anomaly detection research problem;
- we extend our proposed methodology with several algorithms that clearly describe our main proposal for anomaly detection in embedded systems;
- we extend the experimental evaluation and analysis of our proposed framework, by introducing novel experimental metrics;
- we provide an innovative case study that clearly describes runtime anomaly detection using our methodology, within the context of embedded systems.

The remaining part of this paper is organized as follows. Section 2 provides a summary of relevant work in anomaly detection for embedded systems. In Section 3, we provide the theoretical foundations of our research. Section 4 presents a detailed description of our methodology in the case of offline executions. In Section 4.5, we experimentally demonstrate that the analysis framework performs well in classification tasks. Section 5 focuses on the runtime analysis and anomaly detection in an embedded system, thus leading to our innovative algorithm, including its experimental validation. In Section 6, we introduce an innovative case study on runtime anomaly detection using our methodology, within the context of embedded systems. Finally, Section 7 provides conclusions and possible future work.

2 Related Work

In this Section, we provide an overview on research efforts that are related to our work.

Different aspects of a program execution may be used for describing its behavior and, hence, analyzing it in order to detect anomalies. *A system call trace* is a common type of measure for detecting anomalies [9]. A system call trace is an ordered sequence of system calls that a process performs during its execution. Other systems use measures based on the use of resources [10], such as CPU, memory or I/O. The traces collected during a normal execution are classified with *standard pattern matching tools* such as HMM [11, 12, 13, 14, 15], *Embedded Hidden Markov Model* (EHMM) [16], *Neural Networks* and *Genetic Programming* [17], *Support Vector Machine* (SVM) [18], and *rule-based* classifiers [19]. Anomaly detection, also called *intrusion detection* in networked systems, is a very important problem that has been widely studied in different areas and applications. Markovian techniques are one of the best methods for detecting anomalies in a sequence of discrete symbols [20]. Training a Markov model means fine-tuning the parameters of a probabilistic model of a sequence without anomalies; after training, the likelihood of unknown sequences are computed given the parameters of the trained model.

[3] presents two methods for detecting anomalies in embedded systems, namely Markov and *Sequence Time Delay Embedding* (Stide) . The Markov approach evaluates the probabilities of the transitions between events in a training set and uses these probabilities to see if they correspond to the transitions of the test set. The Stide approach builds templates of normal executions and compares the templates with unknown sequences. Other approaches, such as [21], use Markov Models of system call sequences. In some cases, enhanced models can be obtained with Hidden Markov Models, which are widely used for *sequence modeling*.

In [22], authors report a survey of *HMM-based techniques for intrusion detection*. Despite their power, there are few papers dealing with the use of HMM for anomaly detection in embedded systems . Sugaya et al. describe in [23] an anomaly detection system based on HMM modeling of resource consumption, such as CPU, memory and network. In [24], Zandrahimi et al. propose two methods, a *buffer-based* and a *probabilistic detector*. The buffer-based detector builds a cache formed with events considered as normal. During test stage, the method counts the cache misses. However, the probabilistic detector employs the probability of events to evaluate the testing sequence. The approaches are suitable for embedded systems, as they require a smaller memory size and can be easily implemented in hardware. Some authors, for example [25, 16, 26], consider the discrete sequences as signals and use *signal processing techniques* to analyze them.

[27] presents a significant contribution to the field of anomaly detection by developing a novel methodology for *acquiring reliable performance results* for *frequency-based anomaly detectors*. By identifying and characterizing key aspects of the data environment, such as the frequency distribution of data, the paper constructs a synthetic data environment specifically tailored to assess detector performance comprehensively. Through a systematic series of experiments, this approach effectively maps out the performance landscape of the anomaly detector, by highlighting its strengths and exposing areas of weakness. Furthermore, the study demonstrates the practical applicability and extensibility of the insights gained from synthetic data to real-life scenarios, providing valuable guidance for improving anomaly detection techniques.

In [28], authors introduce an *innovative network transmission model* and localization algorithm designed to detect and rank anomalies using only *coarse-grained information* from *network endpoints*. The research addresses the critical challenge of anomaly detection in distributed systems (e.g. [29, 30, 31]), where the lack of sufficient sensors impedes monitoring and timely detection of traffic flow irregularities across interconnected nodes. By developing a novel metric to accurately rank anomalies, the study surpasses traditional statistical models that rely on *standard deviation measures*. The experimental results demonstrate that the proposed algorithm effectively identifies and ranks anomalies, and aligns well with transportation events reported on social media, thereby improving overall system reliability and performance.

[32] provides a *formal runtime security model* that enhances *anomaly-based malware detection* in network-

connected embedded systems. By defining normal system behavior, including execution sequences and timing, and leveraging on-chip hardware to non-intrusively monitor system execution via the *processor trace port*. This approach addresses significant limitations of existing anomaly-based methods, which often suffer from performance overheads and susceptibility to mimicry attacks. The detection method is evaluated on a *network-connected pacemaker* benchmark, which is prototyped in *FPGA*, and simulated in *SystemC*, which highlights its effectiveness against various impression attacks at different.

In [33], authors advance the field of complex system design and analysis by introducing *wrappings integration infrastructure*, a novel *knowledge-based* approach that enhances *system-level* verification beyond *component-level* analysis. This research demonstrates how the integration infrastructure utilizes *domain-specific knowledge* to effectively manage system resources, detect anomalies, and monitor behavior, thereby improving the reliability and robustness of complex systems. The infrastructure provides a flexible framework for incorporating anomaly detection algorithms originally developed for verification and validation of knowledge-based systems, facilitating both offline and online evaluation studies. This contribution not only bridges the gap between component-level and system-level verification but also empowers system developers with tools for better anomaly detection and system monitoring, which leads to more dependable and efficient system designs.

With respect to classical state-of-the-art proposals, our method has the specific merit of addressing embedded systems, which is very relevant at the moment . With respect to similar approaches that make use of HMM for anomaly detection, our main contribution consists in specifically pointing memory references as the input of our analysis, contrary to others that make use of other parameters like CPU and network flows.

3 Preliminaries

In this Section, we summarize the fundamental concepts used through this paper, namely *spectral description of the virtual memory sequences* and program representation by means of HMM. Here, we consider a spectral representation of memory sequences using *Discrete Cosine Transform* (DCT) as described in [16].

3.1 Spectral Description of Memory References

The *Short-Time Fourier Transform* (STFT) is a *Fourier-related transform* used to determine the *sinusoidal frequency* and phase content of local sections of a signal as it changes over time. It describes how the energy is distributed over a spectral range.

We show hereafter that memory references can be described with spectral parameters. In fact, important parts of a program are composed of loops that become peaks in the spectral domain, as we will point out shortly. Let us consider, for example, a simple cycle of this type:

```
i=0;
while(i<N) {
    i++;
}
```

The virtual memory reference sequence generated during the execution of this loop can be modeled with a *sawtooth signal*, as shown in Figure 1 . Calling $F(\omega)$ the *amplitude spectrum* of a single ramp, the analytic form of the *sawtooth spectrum* is defined as follows:

$$F(\omega) = \sum_{n} \delta(n - N) \tag{1}$$

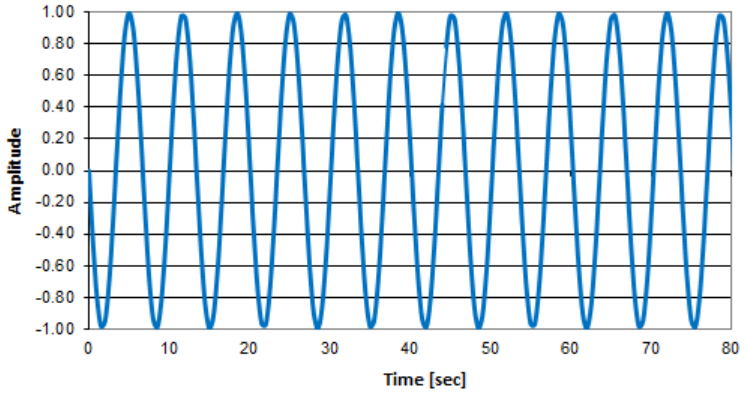


Figure 1: Sawtooth model of a loop

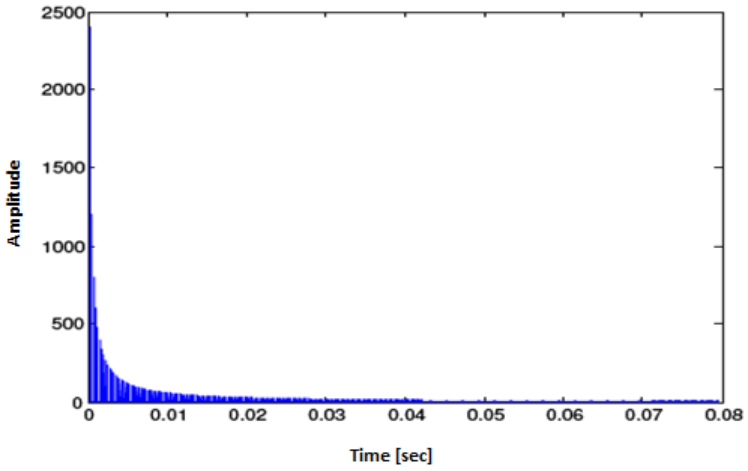


Figure 2: Sawtooth in the amplitude spectrum domain

where N is the number of iterations of the loop. The spectrum is therefore composed by a periodic series of peaks with decreasing amplitudes whose period is related to the loop width N (see Figure 2) .

As a more practical example, let us consider the code fragment reported in Figure 3, which represents a *bubble sort algorithm* . After acquiring the virtual memory sequence and performing its spectral analysis, the STFT described in Equation (2) is applied:

$$X(n) = \sum_{-\infty}^{\infty} x(n)w(n-m)e^{-j\omega n} \tag{2}$$

The sequence of memory addresses is divided into *chunks* or frames, (which usually overlap each other, to reduce artifacts at the boundary). Each chunk is Fourier transformed, and the amplitude spectrum over time is reported in Figure 4. The spectral patterns can be used to characterize the executions. We obtain spectral information with *Fast Discrete Cosine Transform*.

Algorithm 1 `Bubble Sort Algorithm`

Input: Integer vet, Integer N
Output: Integer vet

```
Begin
  i ← 0;
  j ← 0;
  temp ← 0;
  for (i = 0; i < (N − 1); i + +) do
    for (j = i + 1; j > 0; j − −) do
      if (vet[j] < vet[j − 1]) then
        temp ← vet[j];
        vet[j] ← vet[j − 1];
        vet[j − 1] ← temp;
      end if
    end for
  end for
  return vet;
End
```

Figure 3: Bubble sort algorithm

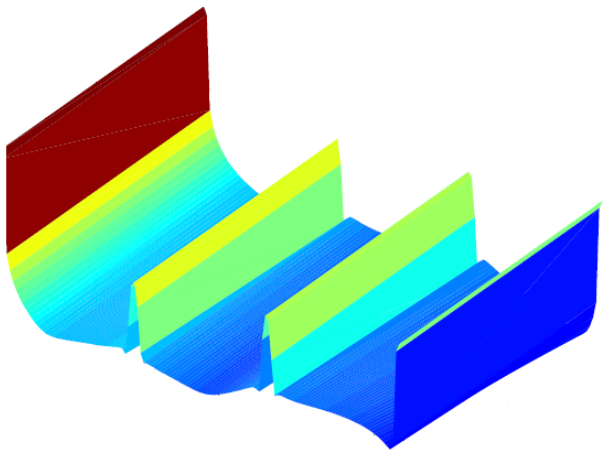


Figure 4: STFT spectrum of the bubble sort routine

3.2 Discrete Cosine Transform Representation

Discrete Cosine Transform is a method to obtain spectral information, and it is used in this work instead of STFT because it is fast and has a good *energy compaction* capability. Energy compaction means the capability of the transform to redistribute *signal energy* into a small number of transform coefficients. It can be characterized by the fraction of the total number of signal transform coefficients that carry a certain (substantial) percentage of the signal energy. The lower this fraction is for a given energy percentage, the better the transform energy compaction capability is.

$$X(n) = \sum_{i=0}^{N-1} x_i \cos\left(\frac{k\pi}{N}\left(i + \frac{1}{2}\right)\right),\ n = 0, \ldots, N-1 \tag{3}$$

The principle advantage of DCT transformation is the removal of redundancy between neighboring addresses. This leads to uncorrelated transform coefficients, which can be processed independently. The effectiveness of a transformation scheme can be directly gauged by its ability to pack input data into as few coefficients as possible. This allows the quantizer to discard coefficients with relatively small amplitudes without introducing visual distortion in the reconstructed image. DCT exhibits excellent energy compaction.

3.3 Hidden Markov Modeling

A standard Hidden Markov model with N states and M possible observation symbols can be denoted as follows:

$$\lambda = (A, B, \pi) \tag{4}$$

such that, A matrix gives the probability of each transition from one state to another, B matrix gives the probability of observing each symbol in each state, and π vector specifies the initial state distribution.

The *Baum-Welch algorithm* [34] is typically used to learn the state transition (i.e., A matrix) and observation symbol probability distributions (i.e., B matrix) of an HMM. The well-known backward and forward procedures can be used to iteratively estimate the model parameters with a space and time complexity of $O(N^2T)$, where T is the length of the sequence of events. The quality of a model can be evaluated using the forward procedure to find the probability that a sequence O was generated by the model λ for all possible paths, namely the likelihood $P(O|\lambda)$.

4 Anomaly Detection Methodology: The Offline Case

In this Section, we describe in details our proposed methodology in the offline case, including its preliminary experimental validation. Later, we introduce our algorithm for supporting the detection at runtime, particularly in embedded systems.

We define the formulation of our problem as follows. Here, we assume that we have a group of N *concurrent processes* running on behalf of a given user. Our objective is to detect anomalies during the execution of the processes due, for example, to a malware attack or a bug in the code. We considered the following two situations.

In the first considered situation, we assume that all but one of the N processes are prone to anomalies because there are well-known services that we know for sure cannot generate any anomalies. In this case, there is only one process that can introduce an anomaly. Our algorithm exploits the effects that the anomaly introduces by measuring the distance of the execution that can contain anomaly with all the other executions.

In the other considered situation, we use the same concept of measuring distances among executions and detect anomalies when all the distances change. The execution that introduces anomalies is found using an *argmax* criterion. The current limit of our approach is that we detect only one anomaly at a time, or, in other words, we detect the execution that introduces the biggest anomaly, in the sense that we will explain shortly.

The methodology applies the following steps:

- employing a `pseudo2D-HMM` model [35] to characterize the workloads of the target system;
- applying HMM to analyze the executions within the target system;

- classifying the target executions by means of the previous HMM analysis.

In the next Sections, we describe all these steps in details.

We used *trace-driven simulations* [36] to test the proposed approach. The traces were a subset of the *SPEC2000* benchmark suite [37]. In particular, we consider the following six workloads: `bzip2` and `gzip`, which are two popular *compression programs*; `eon`, which is a *ray traces program*; `gcc`, `perl` and `vpr`, which are *FPGA place & route programs*. CPU address traces have been obtained by running the applications with different input data. As a result, several executions of each application have been considered. The programs reported above run on an *Intel(R) Core(TM) i*7 − 10700CPU @ 2.90GHz, 16GB of RAM with Windows 10 Pro as operating system . The benchmarks were downloaded from *traces.byu.edu*.

4.1 Pseudo-2D Hidden Markov Models for Workloads Classification

The basic Markov model is the Markov chain, which is represented with a graph composed by a set of N states. This graph describes the fact that the probability of the next event depends on the previous event. The current state is temporally linked to k states in the past via a set of N^k transition probabilities.

In our approach, we use Hidden Markov Models (e.g. [38, 39, 39, 40]) to describe the dynamic behavior of workloads. However, since a program exhibits different behaviors during its execution, different HMMs should be used to model memory references for each behavior. This led us to a `pseudo2D-HMM` structure [35], in which each state in the model, called a *superstate*, represents another HMM. Specifically, `pseudo2D-HMM` is a machine learning approach that requires a learning phase to estimate its parameters. In particular, the goal of the parameter re-estimation is to estimate the parameters of the `pseudo2D-HMM` λ that maximize $P(O|\lambda)$, the probability that the observed sequence O is produced by the model λ. Thus, in the training phase, the memory reference sequence related to a given workload is uniformly divided into segments, on which a DCT is applied. The Discrete Cosine Transform is a well-known signal processing operation with important properties [41]. For example, it is useful in reducing signal redundancy since it places as much energy as possible in as few coefficients as possible (energy compaction). The greatest DCT coefficients are given as input to the `pseudo2D-HMM`.

The `pseudo2D-HMM` is incrementally trained on the segments pertaining to a single memory reference workload type. Each different workload type is modeled using a different `pseudo2D-HMM`.

4.2 HMM Analysis of Executions

In this Section, we state the validity of the analysis methodology we use in the anomaly detection algorithm. Given N executions running in our system, we have N HMM models of the form $\lambda = (A, B, \pi)$, which we call λ_1, λ_2, ..., λ_N. Together with the HMM models, we also have N observations, O_1, O_2, ..., O_N, which are the sequences of symbols estimated from the memory reference sequence with *vector quantization* and from which we estimate the HMM models. The final goal of the proposed methodology is the computation of likelihood matrices computed with these models and observations.

Our basic assumption is that with a HMM model, we capture a high-level description of each observation. As a result, we perform the HMM training with blocks of memory references randomly extracted from an initial part of the address sequence. For the same reason, HMM testing, i.e., the computation of $P(O|\lambda)$, is performed by randomly selecting blocks of data and averaging the results. In the following, we describe how we divide the sequence of memory references.

The data used for the HMM analysis is taken as follows: the sequence of memory references is divided into 1024 address blocks ; on these blocks, a spectral vector is computed with Discrete Cosine Transform and from each vector, the first sixteen coefficients are extracted. The following step is to vector quantize the 16-dimension vectors into 64 centroids; the final result is that each address block is represented by a discrete

symbol from 0 to 63. The symbols are used to train a discrete Hidden Markov Model. It is worth noting that the data for HMM training and testing is different. As regards the HMM training, we consider nine thousand blocks randomly chosen within the first 20000 blocks (each block is of 1024 addresses), and we use the Baum-Welch algorithm. In conclusion, we use 20 million addresses for each execution to train a HMM. It is worth noting that the first million addresses are omitted from the training procedure because they are generally related to an initialization phase.

As regards HMM testing, namely the procedure to obtain the $P(O|\lambda)$ likelihoods, we use the forward algorithm. The data used for testing is chosen into subsequent 10^9 virtual addresses in the following way: each test is performed on 100 blocks chosen randomly into the section of 10^9 addresses. The final value of likelihood is computed by averaging all the computed likelihoods.

Finally, the sequence of memory references is divided into sections of one billion addresses, which we call *EPOCHS*. For each epoch, we compute the likelihoods by averaging the values obtained on 100 blocks of memory reference, each block is composed of 1024 addresses, chosen randomly. The analysis algorithm does not work continuously: as said before, during each epoch, the algorithm acquires addresses, computes DCT coefficients and vector quantization, and computes likelihoods by randomly sampling portions of data.

While the models are obtained from the first 20 million address references, a block of about three million references is extracted every million from the sequence of address references. On these blocks, the likelihoods $P(O|\lambda)$ are computed.

In this way, during the execution of the programs, a series of likelihood matrices , one for each epoch, are computed as follows:

$$\begin{vmatrix} P(O_1^1|\lambda_1) & P(O_2^1|\lambda_1) & \dots & P(O_N^1|\lambda_1) \\ P(O_1^1|\lambda_2) & P(O_2^1|\lambda_2) & \dots & P(O_N^1|\lambda_2) \\ \dots & & & \\ P(O_1^1|\lambda_N) & P(O_2^1|\lambda_N) & \dots & P(O_N^1|\lambda_N) \end{vmatrix}$$
$$\dots$$
$$\begin{vmatrix} P(O_1^N|\lambda_1) & P(O_2^N|\lambda_1) & \dots & P(O_N^N|\lambda_1) \\ P(O_1^N|\lambda_2) & P(O_2^N|\lambda_2) & \dots & P(O_N^N|\lambda_2) \\ \dots & & & \\ P(O_1^N|\lambda_N) & P(O_2^N|\lambda_N) & \dots & P(O_N^N|\lambda_N) \end{vmatrix}$$

where $P(O_i^k|\lambda_j)$ is the likelihood that the j-th model generates the j-th observation, at the k-th time epoch.

Our monitoring algorithm is based upon the differences between HMM models. Usually, the distance is a measure of how well model λ_1 matches observations generated by model λ_2, relative to how well model λ_2 matches observations generated by itself. In other words, this distance is computed by means of Equation (5) :

$$d(\lambda_1, \lambda_2) = |P(O_2|\lambda_1) - P(O_2|\lambda_2)| \tag{5}$$

This definition, however, does not take into account how the other observations behave with reference to λ_1 and λ_2. We therefore consider the two vectors $\Lambda_1 = |P(O_1|\lambda_1), P(O_2|\lambda_1), \dots, P(O_N|\lambda_1)|$ and $\Lambda_2 = |P(O_1|\lambda_2), P(O_2|\lambda_2), \dots, P(O_N|\lambda_2)|$ to better represent the models λ_1 and λ_2. We extend the distance measure between models reported in Equation (5) in the following way:

$$d(\lambda_1, \lambda_2) = \sum_{i=1}^{N} |P(O_i|\lambda_1) - P(O_i|\lambda_2)| \tag{6}$$

The distance measure reported in Equation (6) allows us to better separate the different benchmarks than the distance reported in Equation (5).

4.3 Tools Developed for Offline Analysis: *Valgrind* Plugin

Before developing the detection algorithm, we studied the validity of the described analysis framework by performing several experiments, as we will report shortly. It was impossible to perform this study using stored address traces, because of the extremely large amount of data required to perform the experiments, and the high processing time due to the reading operations. It is worth noting that an experiment may require analyzing several hundred of GBs. For this reason, we developed a tool based on *Valgrind*, which we called *Tracehmm*, to perform all the described processing online.

Valgrind tool [42] is a tracing framework designed to give the possibility to perform dynamic analysis of software, i.e., analysis performed during the code execution. *Valgrind* is distributed with several tools designed to perform common analysis of memory, threading, etc. *Valgrind* is available under the GNU GPLv2 license; it is possible to modify the available tools and to modify also the code of the framework itself, i.e. Coregrind. Coregrind [43] has been designed to analyze already-developed execution code. When the name of the executable file is given as input to *Valgrind*, the code itself is loaded in memory together with the related libraries. The instructions are translated into instructions of a RISC-like language, called VEX IR, and then executed on a virtual CPU.

The *Tracehmm* tool performs the DCT analysis and the HMM training directly using the memory addresses generated during execution. The direct porting of the off-line analysis tools cannot be performed because Coregrind cannot use any library. For this reason, we rewrite all the writing/reading, memory allocation and memory copy functions of the standard library. Moreover, we rewrite also some necessary mathematical functions, namely $cos()$, $sqrt()$ and $log()$. The tool is called from the command line as follows:

```
valgrind --tool=tracehmm [opt] prog & args
```

The *Tracehmm* tool's structure is reported in Figure 5. With this tool, it is possible to perform the training of a new HMM model or the re-estimation of an already computed HMM model. It also performs the *Viterbi test* on a previously trained HMM model for computing the likelihood that the model λ may generate the execution sequence O.

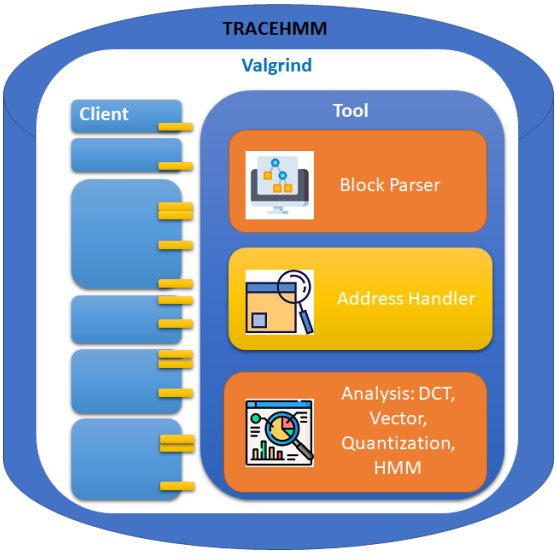


Figure 5: Structure of the *Tracehmm* tool

4.4 Classification of Executions

Before extensively using *Tracehmm*, we perform a limited classification experiment with only four SPEC benchmarks to explore the performance of Neural Networks using the following standard configuration: *feed-forward topology* , weights computation with the *back-propagation algorithm*, six hundred inputs and one output, and 256 elements in the hidden layer. Then we used HMM for the same classification task. We first stored the address memory sequences for the four benchmarks (`bzip`, `gcc`, `go`, and `perl`), we compute the DCT coefficients and vector quantization, and perform classification with Neural Networks and HMM. The classification results are reported in Figure 6 and Figure 7.

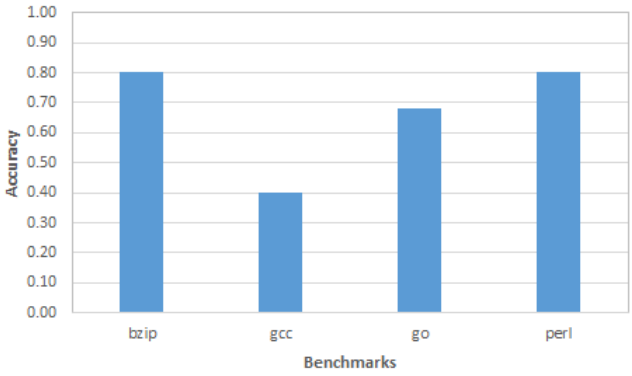


Figure 6: Classification results with neural network model

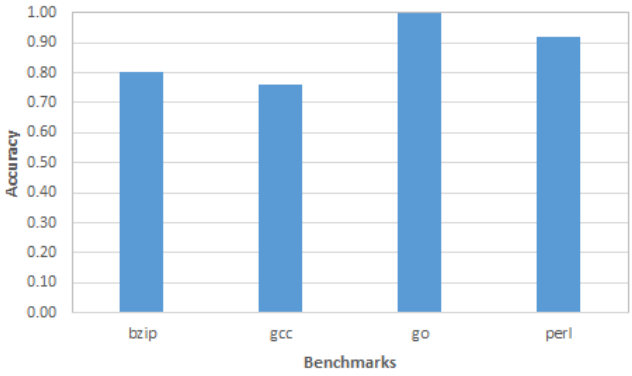


Figure 7: Classification results with HMM

From this comparison, it is clear the superiority of HMM. Then, we performed more extensive classification experiments with the *Valgrind* tool *Tracehmm*. Classification results are reported in Figure 8 for all the SPEC benchmarks: `h264`, `gcc`, `perl`, `bzip`, `go`, `mcf`, `hmmer`, `sjeng`, `quantum`, `omnet`, `astar`, and `xalanc`.

Moreover, in order to evaluate the classification results for different `pseudo2D-HMM` orders, an extensive test has been performed for different number of superstates and states. The results have been averaged for all the workloads. In Figure 9, a graphical representation of the mean classification of all the sequences over the number of states and superstates is depicted. From this data, it comes out that the best `pseudo2D-HMM` orders are six superstates and nine states. As Figure 9 represents average values, we have considered the results for each workload, i.e., averaging the results for all the traces belonging to a given workload type.

The good results reported in the previous Section say that the executions are well discriminated in the

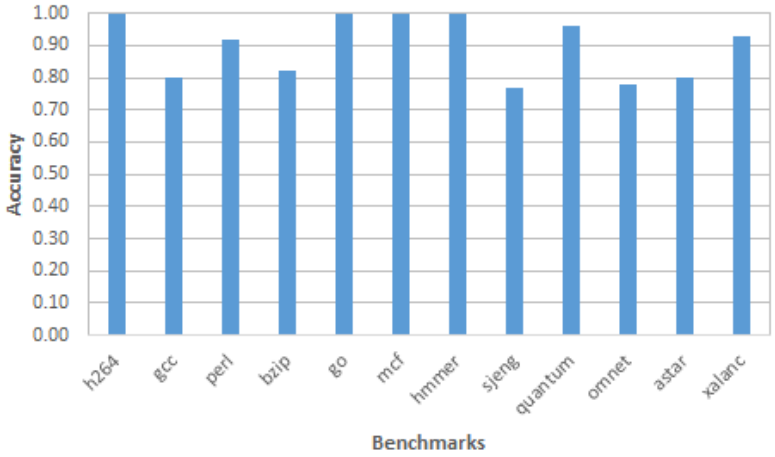


Figure 8: HMM classification result for all the twelve SPEC benchmarks

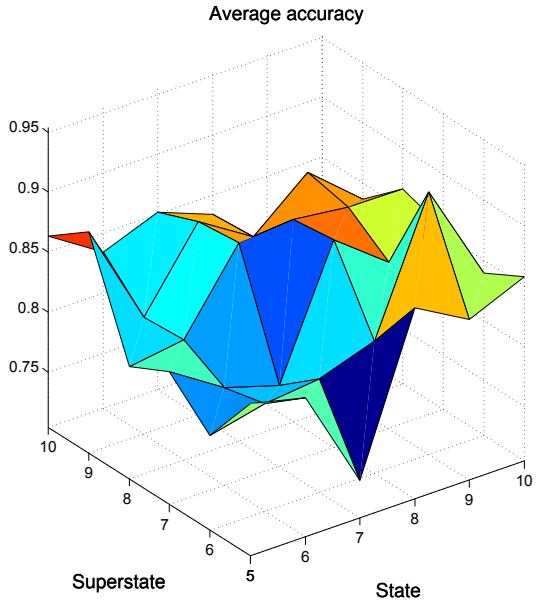


Figure 9: Average recognition rate for all the memory reference sequences over the number of states and superstates of the `pseudo2D-HMM` models

HMM likelihoods space. For this reason, we perform the following experiment: for each epoch, compute the distance matrix between models where the elements are computed according to Equation (6), and find a $3D$ distribution of points whose inter-point Euclidean distance closely resembles this distance matrix. This operation, called *Multidimensional Scaling*, may be used to represent in graphical form a distance matrix. By repeating this operation among several epochs, we obtain the $3D$ graph reported in Figure 10, which shows a good separation among the benchmarks. The line connected to each benchmark represents the dynamic over the first five epochs.

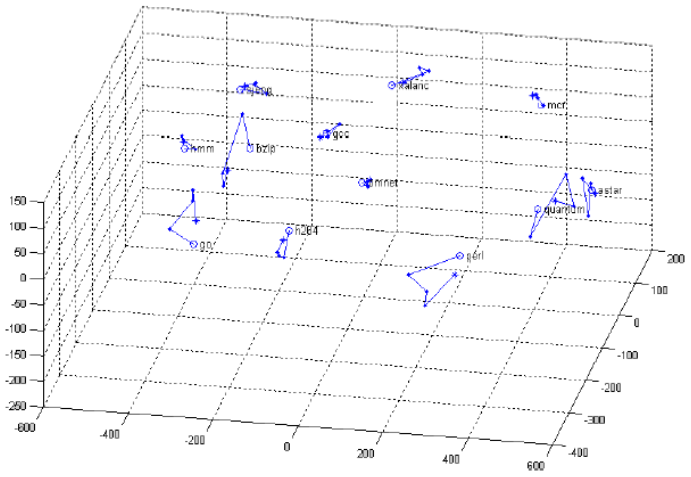


Figure 10: 3D visualization of the distance matrix between models of all the twelve SPEC benchmarks

4.5 Validation of the HMM-based Machine Learning Approach

Before performing experimental evaluations of the detection tool [44, 45], we performed extensive classification experiments to verify the described analysis framework [46]. We assume that the HMM models of the applications and the computation of the likelihood matrices are able to well describe the executions. To explore the validity of this assumption, we classify the executions.

Namely, the HMM models computed from each observation are used to see if the observations with the highest likelihood correspond to the related model. To perform this classification, we use the programs contained in the suite CINT2006 of the SpecCPU2006 benchmark. The suite is composed by twelve programs with different inputs. Of course, a program with a different input generates different observations.

In other words, if the model λ is computed from the observation O, can λ be used to verify that $P(O|\lambda)$ is the maximum for the observation used to compute that model? Of course, numerous observations are generated from the same application.

Classification results are reported in Figure 11 for all the SPEC benchmarks: `h264`, `gcc`, `perl`, `bzip`, `go`, `mcf`, `hmmer`, `sjeng`, `quantum`, `omnet`, `astar`, `xalanc`.

It was impossible to perform this analysis using stored address traces, because of the extremely large amount of data required to perform the experiments, and the high processing time due to the reading operations. In fact, a single classification experiment may require analyzing several billion bytes. For this reason, we developed a *Valgrind* tool, that we called *Tracehmm*, to perform all the described processing online .

5 Runtime Anomalies Detection Algorithm in Embedded Systems

In this Section, we demonstrate how our proposed methodology can be effectively and efficiently used in the context of runtime anomaly detection in embedded systems. Particularly, we provide an algorithm that makes use of the proposed theoretical framework, which has been described in the offline case by Section 4.

We use the programs contained in the suite *CINT2006* of the *SpecCPU2006* benchmark. The suite is composed by twelve programs with different inputs. CPU2006 [47] is *SPEC's CPU-intensive benchmark suite*, stressing a system's processor, memory subsystem and compiler. SPEC designed CPU2006 to provide a comparative measure of compute-intensive performance across the widest practical range of hardware using

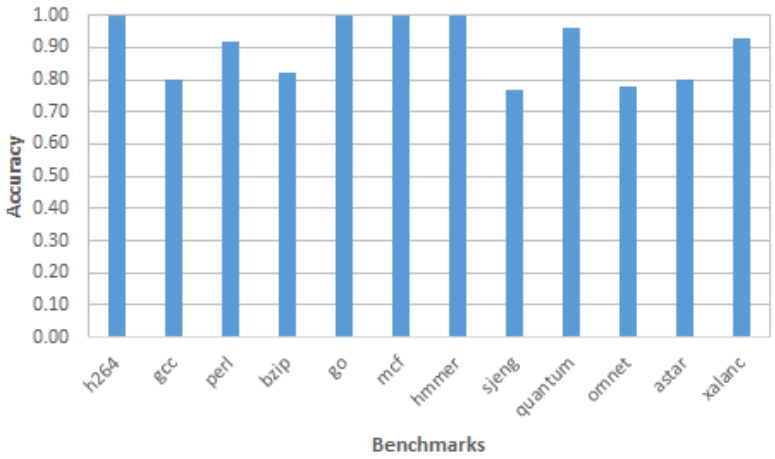


Figure 11: Average classification rate for all the considered workloads

workloads developed from real user applications [48, 49]. All the benchmarks are provided as source code. The programs included in the benchmark suite are the following:

`401.bzip2:` It is derived from the *bzip2 compressor*. The inputs are different files to compress.

`445.gobmk:` It is a version of the *Gnu GO* game. The input data is related to different positions in the board. The output is the following movement in the board.

`458.sjeng:` It is a modified version of the chess game Sjeng 11.2. The inputs is composed by different board positions and different game depths.

`403.gcc:` The program is based on the version 3.2 of GNU Gcc, and it is configured to generate assembler code for the processor $X86-64$.

`h264ref:` This program is the *Karsten Sfuhring* implementation of the H.264 compression standard. The program implements only the compression, but not the decompression.

`400.perlbench:` It is the Perl version 5.8.7.

`429.mcf:` It is derived from a scheduling software used for public transportation. The software has been modified to reduce the number of cache miss and therefore to increment the impact on the CPU than the memory. The inputs are different paths to optimize.

`462.libquantum:` It is a simulator of a quantum computer. It executes the *Peter Shor* algorithm. The inputs are different numbers and the algorithms find the numbers of their factors.

`456.hmmer:` This program analyzes *DNA sequences* using HMM. Its inputs are different reference models and it perform the searching of the correct sequence.

`471.omnetpp:` It simulates an Internet network using the *OMNeT++* simulation system. The inputs are the networks and hosts configurations.

`473.astar:` It implements an algorithm for finding the minimum cost path.

`483.xalancbmk`: It derives from *Xalan-C++*, which is a system that processes documents XML and transform them using style sheets.

Each program with a different input generates a different process. We run each process, and a sequence of virtual memory addresses is generated. So, for the program *bzip2* of the benchmark suite, for example, a different memory reference sequence is generated for each input. With each sequence of memory references, an HMM model is estimated.

Our work is based on the assumption that by iteratively re-estimating the HMM parameters with each sequence, we obtain an HMM model that describes the program itself, not the execution sequence. With the trained HMM's and other memory reference sequences, we derive the likelihood that the sequences are generated by the model. It is worth noting that the data for HMM training and testing is different.

Furthermore, the runtime anomaly detection algorithm based on the proposed execution model is presented. It has been tested on the memory references captured from an ARM9 Linux based embedded device. Using the PIN tracing tool [50], we have developed a runtime monitoring tool that has the structure described in Figure 12.

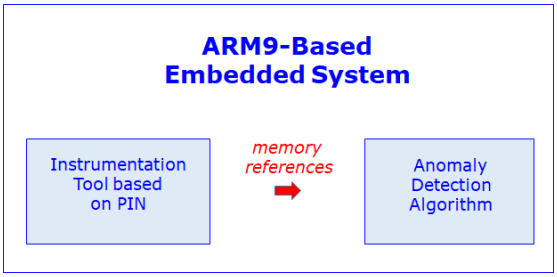


Figure 12: Runtime anomalies detection diagram

5.1 Parametrization of Memory Reference Sequences

The initial part of the executions is normally devoted to initialization tasks and is very different from the *steady-state phase* of the programs. For this reason, we simply blindly fast forward for 1 billion instructions before starting data analysis.

After that, each sequence of memory address references is divided into 1024 address blocks; on these blocks, a spectral vector is computed with Discrete Cosine Transform, and from each vector, the first sixteen coefficients are extracted. The following step is to perform vector quantization with 64 centroids [51] to reduce the 16-dimension vectors into 1 symbol; the final result is that each block of 1024 addresses is represented by a discrete symbol from 0 to 63. The sequences of memory addresses are then transformed into sequences of symbols, which are called *Observations*.

Given N programs running in our system, we have thus N observations, O_1, O_2, ..., O_N, which are the sequences of symbols estimated from the memory reference sequences with DCT analysis and vector quantization.

The N observations are used to train a Hidden Markov Model of each application, called λ_1, λ_2, ..., λ_N in the following. The training of the HMM models is performed as follows. For each observation, nine thousand blocks are randomly chosen within the first 20000 symbols (recall that each symbol corresponds to a block, which is composed of 1024 addresses) and the HMM parameters are computed with the Baum-Welch algorithm. Thus, we use 20 million addresses of each execution to train a HMM. It is worth recalling that the first billion of addresses is omitted from the training procedure because it is generally related to an initialization phase.

After HMM modeling, the observations are used to compute, with the forward-back-word algorithm [52], the $P(O|\lambda)$ likelihoods. The data used for computing the likelihoods is chosen into the subsequent billion memory addresses in the following way: twenty sub-sequences of 100 symbols are chosen randomly within the sequence corresponding to the billion addresses. The final value of likelihood is obtained by averaging all the computed likelihoods. It is worth recalling that the observations used to compute the $P(O|\lambda)$ are formed by sequences of 100 symbols.

In conclusion, the sequence of memory references is divided in sections of one billion addresses that we call *epochs*. For each epoch we compute the likelihoods by averaging the values obtained on twenty sets of 100 blocks of memory reference, each block is of 1024 addresses, chosen randomly. The analysis algorithm does not work continuously: as said before, during each epoch the algorithm acquires addresses, computes DCT coefficients and vector quantization and computes likelihoods by randomly sampling portions of data. The initial memory references are thrown away to avoid the initial transient. After that, the N observations are used to train a Hidden Markov Model of each observation, called λ_1, λ_2, ..., λ_N in the following.

A standard HMM with N states and M possible observation symbols can be denoted $\lambda = (A, B, \pi)$. The A matrix gives the probability of each transition from one state to another, the B matrix gives the probability of observing each symbol in each state, and the π vector specifies the initial state distribution. The Baum-Welch algorithm [34] is typically used to learn the state transition (A matrix) and observation symbol probability distributions (B matrix) of an HMM. The well known forward-backward procedures is used to evaluate the quality of models λ_i, with a space and time complexity of $O(N^2T)$, where T is the length of the input sequence. The quality of a model is the probability that the observation sequence O is generated by the model λ for all possible paths, and is referred to the likelihood $P(O|\lambda)$.

The ultimate goal of the proposed methodology is the computation of likelihood matrices computed with these models and observations. Our basic assumption is that with a HMM model we capture a high-level description of each observation. For this reason we perform the HMM training with blocks of 1024 memory references randomly extracted from an initial part of the address sequence. For the same reason, the HMM testing, i.e. the computation of $P(O|\lambda)$, is performed by randomly selecting blocks of data and averaging the results. While the models are obtained from the first 20 million address references, a block of about three million references is extracted every million from the sequence of address references. On these blocks the likelihoods $P(O|\lambda)$ are computed.

5.2 Anomaly Detection Algorithm

The process running on the embedded device is instrumented and the memory references are collected. The time sequence of memory references is divided in epochs used to train the corresponding Hidden Markov Model of the execution. After the training, each new epoch of the time sequence is used to compute the new matrix $|P(O_i|\lambda_j)|$, as described in Section 4.2. From this matrix, the symmetric matrix M^k of the distances between the models is obtained:

	λ_1	λ_2	λ_3	.
λ_1	0	$m_{1,2}$	$m_{1,3}$	.
λ_2	$m_{2,1}$	0	$m_{2,3}$	.
λ_3	$m_{3,1}$	$m_{3,2}$	0	.
.	.	.	.	0

(7)

where $m_{i,j}$ is the Euclidean norm of the difference between the row i and the row j of the $|P(O_i|\lambda_j)|$ matrix.

At epoch k, the difference matrix D^k is computed as follows:

$$D^k = \sqrt{|M^k - M^{k-1}|} \tag{8}$$

where k is the current epoch and $k-1$ is the previous one.

Then, the ev^k vector containing the eigenvalues of D^k is computed:

$$ev^k = eigenvalues(D^k). \tag{9}$$

Finally, to detect whether an anomaly is occurred in the current epoch, the *Nuclear Norm*, NN of the eigenvalue vector ev^k is computed as follows :

$$\mathrm{NN} = \sum_{i=1}^{N} \left| ev_i^k \right|. \tag{10}$$

It is worth recalling that NN is an excellent energy index [53]. The percentage variation of the nuclear norm is compared to a given threshold THR in order to detect if in the current epoch an anomaly is occurring.

The meaning of the described algorithm is that when an observation varies due to an anomaly, the distance between the corresponding likelihood among different epochs accordingly changes. Moreover, if an anomaly is detected, we can determine which model has produced the anomaly, i.e. which execution has changed its behavior. The anomaly detection algorithm is summarized Figure 13.

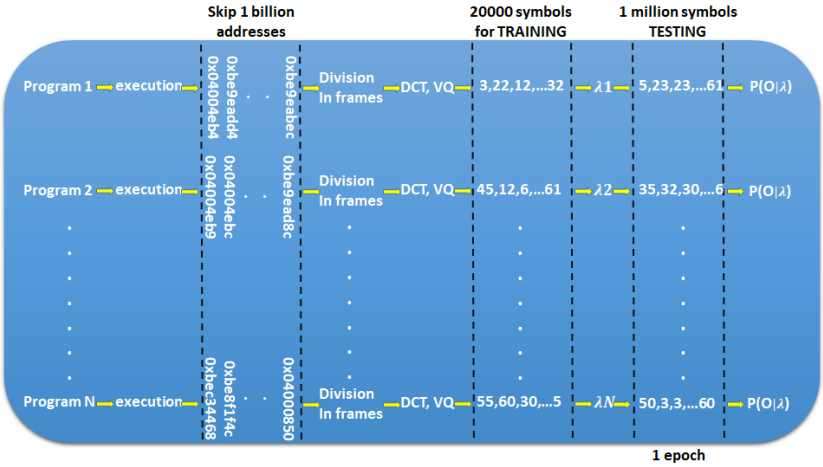


Figure 13: Schematic view of the anomalies detection algorithm

On the other hand, by focusing on more algorithmic details, Figure 14 show the pseudo code of the proposed anomaly detection algorithm presented in Figure 13.

5.3 Runtime Implementation using the PIN Tool

Tools capable of extracting and processing memory traces from running processes have been developed to monitor systems in real-time. We chose Intel Pin as tracing framework, as it is freely available on both x86, x86_64 and ARM architectures under a Linux environment. Using the API of PIN [50], we developed a tool that attach to a running process, track memory accesses until requested then detach and let the monitored application continue its execution unharmed. Provided a rule set to detect processes to monitor (e.g. processes listening on a specific port, processes running as a specific user), standard system tools can be used to find matching *Process IDs* (PID). PIDs will be used by training and tracking PIN tools to attach to each process and produce models or test workload resemblance as requested [54].

Algorithm 3 Anomalies Detection Algorithm

Input: Memory References M_r, Integer THR
Output: Integer $Process_{ID}$

Begin
$ANOMALY \leftarrow$ **FALSE**;
$Process_{ID} \leftarrow 0$;
$epoch \leftarrow divideMemory(M_r)$;
$TS \leftarrow trainHMM(epoch)$;
while (TRUE) do
$P \leftarrow computeLikelihoodMatrix(TS)$;
$M \leftarrow computeDistanceMatrix(P)$;
$D \leftarrow computeDifferenceMatrix(M)$;
$NN \leftarrow computeNuclearNorm(D)$;
if $(NN > THR)$ **then**
$ANOMALY \leftarrow$ **TRUE**;
$Process_{ID} \leftarrow findAnomalyExecution()$;
return $Process_{ID}$;
end if
end while
End

Figure 14: Pseudo code for the anomalies detection algorithm

A new group of applications that have been chosen to be monitored requires an (automated) step of tuning, where training and testing parameters are optimized to suite the group of applications chosen [55, 56]. Once parameters are defined, a PIN tool is used to generate Markov model and centroids files of each application. Multiple input sources are used to train the model in order to obtain a good generalization of the application model. Another PIN tool provides testing capabilities: it requires the PID of the process to track and the Markov model of the application it claims to belong to. A logarithmic affinity probability is then produced.

A tracking system daemon is devoted to run the testing tool at regular time intervals: the tool attaches to target process, dumps memory references and then detaches, so target process is able to keep running without any more overhead. Memory addresses are then processed and output is used to classify the tracked process as belonging to the claimed application or not. The main tasks to be performed can be summarized in the following steps:

1. find the process to track;
2. use PIN to extract memory references of the process;
3. train the DCT-HMM description of the process;
4. compute the matrix $P(O_i|\lambda_j)$ and check the difference of error matrix to detect anomalies.

In Figure 15, a detailed pseudo code of the real-time monitoring tool is reported.

5.4 Experimental Results

The tool based on PIN described above has been tested in three different experiments, as we will show shortly. In all the tests of this Section , an embedded device equipped with an ARM9 at 500 MHz, 128 Mb RAM and

Algorithm 2 Anomalies Detection Tools

Input: Integer THR, Integer N, Integer $EPOCHE\_SIZE$
Output: Boolean $Anomaly$

```
Begin
  P_ID ← findProcess();
  attachPIN(P_ID);

  PIN TOOL:
  COUNT ← 0;
  while (COUNT < EPOCHE_SIZE) do
    traceMemoryAccesses(N);
    writeToBuffer();
    COUNT++;
    skipMemoryAccess();
  end while

  TRACKING DAEMON:
  while (TRUE) do
    waiteForBufferToFill(N);
    COUNT++;
    spawnThread(analyzeTrace());
    if (COUNT < EPOCH_SIZE) then
        continue;
    else
        sleepUntilNextFill();
    end if
  end while

  EPOCH DAEMON:
  EveryXminutes :
  if (newEpochExists()) then
    for all (epoch in previousEpochs) do
        Δ_i ← computeMatrix(epoch);
        if (Δ_i > THR) then
            detectAnomaly();
            Anomaly ← TRUE;
            return Anomaly;
        end if
    end for
  end if
End
```

Figure 15: Pseudo code for the anomalies detection tools

a Debian Linux has been used. An important issue related to any monitoring system applyied to embedded systems is overhead. Embedded systems like the ARM based computers, in fact, have low computational power and limited power supplies, thus constraints in terms of computational and power requirements are particularly important. Our prototype implemented PIN with processor instructions, therefore only computational constraints have been experimentally evaluated. Computational constraints have been evaluated in terms of the slowdown that the embedded system suffers due to the instrumentation of the code to extract

the memory references, since most of the computational load can be assigned to a different system. Our tool each time a memory address is accessed by the instrumented application executes a callback function in order to save the memory reference, hence introducing an unavoidable overhead to the normal execution of the application. This is done for the strictly necessary number of references to provide good values of accuracy, then tool detaches from the application, which can continue its execution normally. Those memory references are shared with a tracking process, typically on a different machine on the same network, which will begin the analysis of the trace avoiding the tracing tool to slow down the application even more. In our prototype the tracing caused an about 33% slow down. However, using other instrumentation approaches the slowdown can be greatly reduced.

5.4.1 Experiment 1 - Malware Detection

In this first experimental campaign, the PIN tool has been attached to two different processes, and the anomaly detector changes artificially its input at a given epoch. This test aims at simulating a malware affecting a process that suddenly change its behavior becoming a different process. For this test 8 different models of *SPEC CPU2006* benchmark have been used, namely `sjeng`, `omnet`, `astar`, `h264`, `xalanc`, `mcf`, `perl`, `quantum` , changing this 8 execution suddenly into `gcc`, `hmm`, `bzip_0`, `bzip_1`, `bzip_2`, `bzip_3`, `go_0`, `go_1`, `go_2`, `go_3`, where for bzip and go different execution have been considered. Thus, 80 different anomalies have been tested. For example, in this test, at a given epoch, the execution of `astar` suddenly becomes `gcc` (or `bzip_0`, etc.). This behavioral change can be detected by the proposed algorithm.

In Figure 16, the corresponding False Positive versus False Negative behavior for anomaly epoch determination has been obtained.

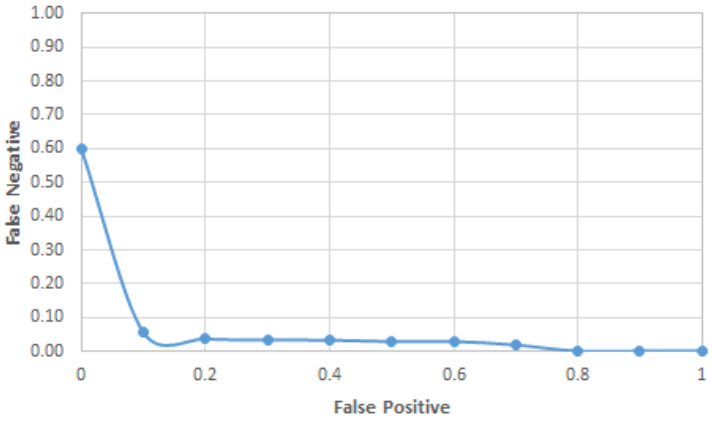


Figure 16: Performances of the detection algorithm for change behavior anomalies

The experimental results show that the algorithm can determine in an accurate way the epoch that has produced the error, with an equal error rate below 0.8%.

5.4.2 Experiment 2 - Loop Bug Detection

In this experiment, an infinite loop substitutes the normal execution of the benchmark at a given epoch. This experiment shows if the proposed algorithm is capable to detect anomalies in programs that remain blocked in loops.

Table 1: Loop bug detection results

SNR	Correct Detections
0 dB	all
5 dB	all
10 dB	all
15 dB	all
20 dB	all
25 dB	all
30 dB	all
35 dB	all
40 dB	none
45 dB	none
50 dB	none

This test has been conducted using the following benchmarks: `omnet`, `astar`, `h264`, `xalanc`, `mcf`, `perl`, `gcc`, `bzip`. In all the 8 tested cases, the epoch in which the anomaly has occurred has been always correctly determined. In this test, the execution that introduces the anomaly has also been detected with an accuracy of 89.5%.

5.4.3 Experiment 3 - Random Error in Memory References Detection

In this third experiment, the memory trace gathered using PIN has been modified by adding a *white gaussian noise*. This experiment shows what energy differences in memory reference are detected as anomaly by the proposed program.

The results have been conducted on the same 8 benchmark of Experiment 2, namely: `omnet`, `astar`, `h264`, `xalanc`, `mcf`, `perl`, `gcc`, `bzip` , and the noise has been added to one benchmark at a time, resulting in 8 tests for each value of SNR, using a threshold $THR = 100$.

This experiment shows that the proposed method is capable of detecting random modifications of the memory references when this behavior is evident, starting from 35 dB, and it becomes easier and easier as the SNR is decreasing.

6 Case Study: Runtime Anomaly Detection in Embedded Systems

In this Section, we introduce an innovative case study on runtime anomaly detection using our methodology, within the context of embedded systems.

Embedded devices have seen a remarkable surge in popularity recently, paralleling a significant increase in the variety and sophistication of anomalies that can affect their performance and security. Consequently, both industry and academia have focused extensively on addressing the unique challenges of anomaly detection in embedded systems.

Anomaly detection is crucial for these systems due to their widespread use in critical applications where reliability is paramount. To tackle this, advanced behavior-based anomaly detection systems have been developed, leveraging machine learning algorithms to identify and analyze deviations in data such as sensor readings, communication patterns, and power consumption. Hence, we provide in this Section a detailed case study demonstrating how our proposed approach can be employed for anomaly detection in embedded systems and IoT devices. By examining real-world scenarios, we illustrate the effectiveness of our method in

identifying and mitigating anomalies, thereby enhancing the reliability and security of these systems. This case study showcases how our approach can detect irregularities in various parameters, such as sensor data, communication protocols, and energy consumption, ensuring the robust operation of embedded and IoT devices.

In recent years, the significance of anomaly detection in embedded systems and IoT devices has escalated due to the increasing integration of these technologies in critical applications. Statistical data from various reports highlight the scale and impact of anomaly detection efforts.

For instance, a comprehensive survey on anomaly detection strategies for *cyber-physical systems* (CPS) indicates that advanced methods such as edge and edge-Cloud computing are becoming pivotal. These methods are designed to handle large volumes of high-dimensional data generated by IoT devices, with one study reporting a significant reduction in detection latency and an improvement in overall accuracy through the use of these techniques [57, 58].

Furthermore, the adoption of machine learning models in anomaly detection has shown promising results. A study on *industrial IoT* (IIoT) networks revealed that using a hybrid machine learning approach can enhance the detection rate of anomalies while maintaining low false-positive rates. This is particularly important in environments where real-time response is critical [57]. Additionally, [58] focused on *distributed online one-class support vector machines* (SVM) demonstrated that such approaches are effective for anomaly detection in networked embedded systems. This method allows for continuous learning and adaptation, crucial for maintaining security and operational efficiency in dynamic environments.

Monitoring data in embedded systems is crucial for assessing system health, identifying workload patterns, and defining metric spaces, which are subsequently used to detect anomalies. To ensure the efficacy of anomaly detection, various fault scenarios can be simulated and analyzed, including:

- **Sensitive Sensor Interactions**. Embedded systems often interact with sensitive sensors and actuators. Anomalies can be detected by monitoring unusual patterns in these interactions, which may indicate tampering or misuse. For instance, unauthorized access to a temperature sensor in an industrial control system could signal an attempt to disrupt normal operations.

- **CPU-Intensive Loops**. Faulty or malicious code can introduce infinite loops or heavy computational tasks that exhaust CPU resources, leading to performance degradation or system crashes. By analyzing CPU usage patterns and detecting abnormal spikes, these issues can be identified and mitigated. For example, an embedded system might exhibit abnormal CPU usage due to a spin lock fault, causing a significant slowdown.

- **Memory Leaks**. Memory leaks occur when a system continuously allocates memory without releasing it, eventually exhausting available memory and causing crashes. In embedded systems, this can be particularly problematic due to limited memory resources. Anomaly detection algorithms can monitor memory usage over time to identify and address leaks before they cause significant harm. For instance, a gradual increase in memory usage without a corresponding release could indicate a memory leak.

- **Disk I/O Errors**. Disk I/O operations in embedded systems often follow predictable patterns. Anomalies in these patterns, such as unexpected spikes in read/write operations, can indicate hardware failures or malicious activity. By continuously monitoring disk I/O performance, potential issues can be detected early. For example, a sudden increase in disk access could be a sign of a disk I/O error caused by malware.

- **Network Anomalies**. Embedded systems are increasingly networked, making them vulnerable to various network-based attacks. Unusual network traffic patterns, such as unexpected bursts of data

transmission or communication with unknown hosts, can indicate network anomalies. Monitoring network traffic for these patterns helps in identifying and responding to potential threats. For instance, abnormal outgoing network traffic could suggest a denial-of-service attack or data theft attempt.

In Figure 17, a comprehensive IoT workflow is shown where various stages are involved to ensure efficient monitoring and anomaly detection:

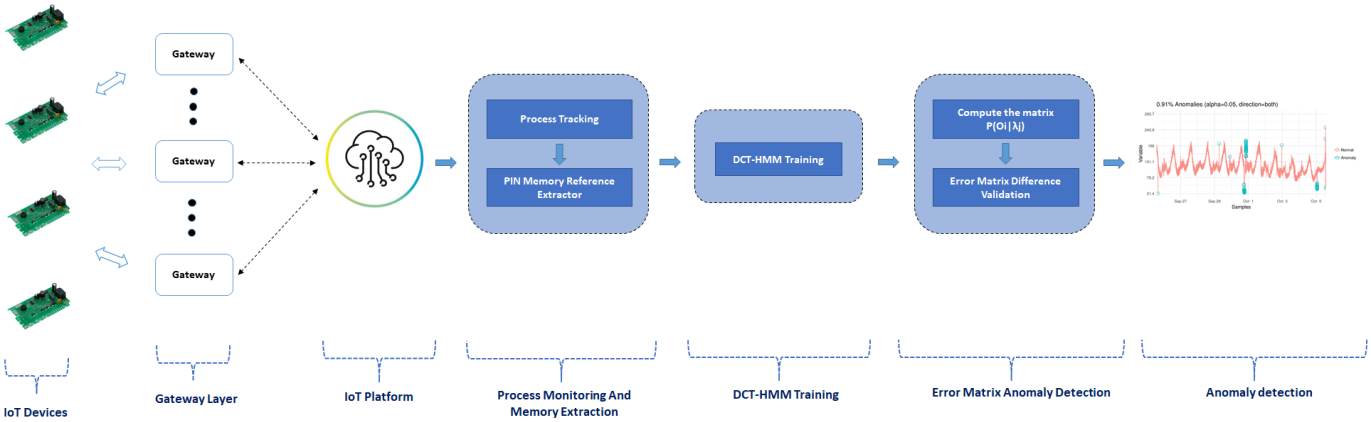


Figure 17: Workflow for monitoring and anomaly detection in IoT systems using DCT-HMM

- **IoT Platform Setup**: The workflow begins with the establishment of an IoT platform, which includes IoT gateways and devices. The platform facilitates seamless communication and data aggregation from numerous IoT devices deployed in the field.

- **Process Monitoring**: A specific process is selected for monitoring. This process involves tracking various performance metrics and data points critical to the operation of the IoT devices.

- **Memory Reference Extraction**: Using the PIN tool, memory references of the selected process are extracted. PIN is a dynamic binary instrumentation framework that allows the collection of detailed memory access patterns and other relevant information from running applications.

- **Training the DCT-HMM Model**: The next step involves training a *Discrete Cosine Transform-Hidden Markov Model* (DCT-HMM) using the extracted memory references. This model helps in understanding the normal behavior of the process by analyzing the patterns in the data.

- **Computing Matrix** $P(O|\lambda)$: Once the model is trained, the matrix $P(O|\lambda)$, which represents the transition probabilities between different states in the HMM, is computed. This matrix is crucial for detecting deviations from normal behavior.

- **Anomaly Detection**: Finally, anomalies are detected by comparing the difference in the error matrix. This involves calculating the error matrix during real-time monitoring and checking it against the learned model. Significant differences indicate potential anomalies, prompting further investigation or corrective actions.

7 Conclusion

In this paper, we propose a technique to determine anomaly behaviors in programs based on a model built from memory reference sequences. We present a detailed modeling techniques based on spectral representation of memory reference sequences and Hidden Markov Models, and show that the execution epochs of each program can be clustered and represented using multidimensional scaling. This modeling technique is the basis of the proposed algorithm for anomaly detection, which is capable of accurately determine the epoch where an anomaly has occurred [59, 60, 61, 62, 63]. It is also capable to determine the program subject to the anomaly.

The main limitation of the technique is that it considers a single thread of execution. In a multi-threaded program, this technique should be extended to consider an aggregate model of all the different threads. In the scenario of monitoring programs running on a given embedded system, this is not a problem, as typically the processes in such environments are single-threaded.

The experimental evaluations of the algorithm reported in the paper is obviously preliminary, but, at the same, it has provided a clear vision of the potentialities offered by our proposed framework, and its reliability in effectively and efficiently supporting anomaly detection. Indeed, we obviously plan to further experimentally investigate the algorithm through extensive experiments with different faults and anomaly injections. In addition to this, we plan to extend our proposed framework to other classes of data, such as *streaming data* (e.g., [64]), and *data compression* (e.g., [65, 66, 67]) and *privacy issues* (e.g., [68]) in order to catch other advanced features that may return to be useful in *emerging Big Data environments* (e.g., [69, 70]).

Conflict of Interest: The authors declare no conflict of interest.

Funding: This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

[1] Makhoul J. A fast cosine transform in one and two dimensions. *IEEE Transactions on Acoustics, Speech, and Signal Processing*. 1980; 28(1): 27-34. DOI: https://doi.org/10.1109/TASSP.1980.1163351

[2] Paxson V, Floyd S. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking*. 1995; 3(3): 226-244. DOI: https://doi.org/10.1109/90.392383

[3] Maxion RA, Tan KMC. Anomaly detection in embedded systems. *IEEE Transactions on Computers*. 2002; 51(2): 108-120. DOI: https://doi.org/10.1109/12.980003

[4] Rabiner LR, Juang BH. *Foundamentals of Speech Recognition.* Prentice Hall Signal Processing Series; 1993.

[5] Madhyastha TM, Reed DA. Learning to classify parallel input/output access patterns. *IEEE Transactions on Parallel and Distributed Systems*. 2002; 13(8): 802-813. DOI: https://doi.org/10.1109/TPDS.2002.1028437

[6] Pavliotis GA. *Stochastic Processes and Applications.* Springer New York, NY; 2014. DOI: https://doi.org/10.1007/978-1-4939-1323-7

[7] Raghavan SV, Vasukiamaiyar D, Harign G. Hierarchical approach to building generative network-workload models. *Computer Networks and ISDN Systems*. 1995; 27(7): 1193-1206. DOI: https://doi.org/10.1016/0169-7552(94)00012-I

[8] Cuzzocrea A, Mumolo E, Cecolin R. Runtime anomaly detection in embedded systems by binary tracing and hidden markov models. In: *2015 IEEE Annual Computer Software and Applications Conference, COMPSAC 2015, 1-5 July 2015, Taichung, Taiwan.* IEEE; 2015, p.15-22. DOI: https://doi.org/10.1109/COMPSAC.2015.89

[9] Li Z, Tian JF, Yang XH. Program behavior monitoring based on system call attributes. *Journal of Computer Research and Development.* 2012; 49(8): 1676-1684.

[10] Ohno Y, Sugaya M, Van Der Zee A, Nakajima T. Anomaly detection system using resource pattern learning. In: *2009 Software Technologies for Future Dependable Distributed Systems, STFSSD 2009, 17 March 2009, Tokyo, Japan.* IEEE; 2009. p.38-42. DOI: https://doi.org/10.1109/STFSSD.2009.41

[11] Gao B, Ma H-Y, Yang Y-H. HMMs (Hidden Markov models) based on anomaly intrusion detection method. In: *Proceedings. International Conference on Machine Learning and Cybernetics, ICMLC 2002, 4-5 November 2002, Beijing, China.* IEEE; 2002. p.381-385. DOI: https://doi.org/10.1109/ICMLC.2002.1176779

[12] Qiao Y, Xin XW, Bin Y, Ge S. Anomaly intrusion detection method based on HMM. *Electronics Letters.* 2002; 38(13): 663-664. DOI: https://doi.org/10.1049/el:20020467

[13] Yin Q-B, Shen L-R, Zhang R-B, Li X-Y, Wang H-Q. Intrusion detection based on hidden Markov model. In: *Proceedings of the 2003 International Conference on Machine Learning and Cybernetics, ICMLC 2003, 5 November 2003, Xi'an, China.* IEEE; 2003. p.3115-3118. DOI: https://doi.org/10.1109/ICMLC.2003.1260114

[14] Zhang X, Fan P, Zhu Z. A new anomaly detection method based on hierarchical HMM. In: *Proceedings of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2003, 29 August 2003, Chengdu, China.* IEEE; 2003. p.249-252. DOI: https://doi.org/10.1109/PDCAT.2003.1236299

[15] Ma X, Schonfeld D, Khokhar A. A general two-dimensional hidden Markov model and its application in image classification. In: *2007 IEEE International Conference on Image Processing, ICIP 2007, 16 September - 19 October 2007, San Antonio, TX, USA.* IEEE; 2007. p.VI41-VI44. DOI: https://doi.org/10.1109/ICIP.2007.4379516

[16] Moro A, Mumolo E, Nolich M. Workload modeling using pseudo2D-HMM. In: *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, MASCOTS 2009, 21-23 September 2009, London, UK.* IEEE; 2009. p.1-2. DOI: https://doi.org/10.1109/MASCOT.2009.5366721

[17] González FA, Dasgupta D. Anomaly detection using real-valued negative selection. *Genetic Programming and Evolvable Machines.* 2003; 4: 383-403. DOI: https://doi.org/10.1023/A:1026195112518

[18] Wang M, Zhang C, Yu J. (2006, June). Native API based windows anomaly intrusion detection method using SVM. In: *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing ,SUTC'06, 5-7 June 2006.* IEEE; 2006; 1: 6. DOI: https://doi.org/10.1109/SUTC11330.2006

[19] Li X, Han J, Kim S, Gonzalez H. Anomaly detection in moving object. In:*Intelligence and Security Informatics.* 2008; 357-381. DOI: https://doi.org/10.1007/978-3-540-69209-6_19

[20] Chandola V, Banerjee A, Kumar V. Anomaly detection for discrete sequences: A survey. *IEEE Transactions on Knowledge and Data Engineering*. 2012; 24(5): 823-839. DOI: https://doi.org/10.1109/TKDE.2010.235

[21] Tan X, Wang W, Xi H, Yin B. A Markov model of system calls sequence and its application in anomaly detection. *Computer Engineering*. 2002; 43: 189-191.

[22] Wang P, Shi L, Wang B, Wu Y, Liu Y. Survey on HMM based anomaly intrusion detection using system calls. In: *2010 5th International Conference on Computer Science & Education, ICCSE 2010, 24-27 August 2010, Hefei, China*. IEEE; 2010. p.102-105. DOI: https://doi.org/10.1109/ICCSE.2010.5593839

[23] Sugaya M, Ohno Y, Van Der Zee A, Nakajima T. A lightweight anomaly detection system for information appliances. In: *2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2009, 17-20 March, 2009, Tokyo, Japan*. 2009; 257266. DOI: https://doi.org/10.1109/ISORC.2009.39

[24] Zandrahimi M, Zarandi HR, Mottaghi MH. Two effective methods to detect anomalies in embedded systems. *Microelectronics Journal*. 2012; 43(1): 77-87. DOI: https://doi.org/10.1016/j.mejo.2011.11.003

[25] Moro A, Mumolo E, Nolich M. Ergodic continuous hidden Markov models for workload characterization. In: *2009 Proceedings of 6th International Symposium on Image and Signal Processing and Analysis, ISPA 2009, 16-18 September 2009, Salzburg, Austria*. IEEE; 2009. p.99-104. DOI: https://doi.org/10.1109/ISPA.2009.5297771

[26] Zadeh MMZ, Salem M, Kumar N, Cutulenco G, Fishmeister S. SiPTA: Signal processing for trace-based anomaly detection. In: *Proceedings of the 14th International Conference on Embedded Software, EMSOFT 2014, 12-17 October 2014, Uttar Pradesh, India*. Association for Computing Machinery; 2014. p.1-10. DOI: https://doi.org/10.1145/2656045.2656071

[27] Hansen JP, Tan KMC, Maxion RA. Anomaly detector performance evaluation using a parameterized environment. In: *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection, RAID 2006, 20-22 September 2006, Hamburg, Germany*. 2006. p.106-126. DOI: https://doi.org/10.1007/11856214_6

[28] Chong F, Chua T, Lim EP, Huberman BA. Detecting flow anomalies in distributed systems. In: *2014 IEEE International Conference on Data Mining, ICDM 2014, 14-17 December, 2014, Shenzhen, China*. IEEE; 2014. p.100-109. DOI: https://doi.org/10.1109/ICDM.2014.94

[29] Wu Z, Zhou X, Xu J. A result fusion based distributed anomaly detection system for android smartphones. *Journal of Networks*. 2013; 8(2): 273-282. DOI: https://doi.org/10.4304/jnw.8.2.273-282

[30] Rajasegarar S, Leckie C, Palaniswami M. Hyperspherical cluster based distributed anomaly detection in wireless sensor networks. *Journal of Parallel and Distributed Computing*. 2014; 74(1): 1833-1847. DOI: https://doi.org/10.1016/j.jpdc.2013.09.005

[31] Peiris M, Hill JH, Thelin J, Bykov S, Kliot G, Konig C. PAD: Performance anomaly detection in multi-server distributed systems. In: *2014 IEEE 7th International Conference on Cloud Computing, CLOUD 2014, 27 June - 2 July, 2014, Anchorage, AK, USA*. IEEE; 2014. p.769-776. DOI: https://doi.org/10.1109/CLOUD.2014.107

[32] Lu S, Lysecky R. Time and Sequence Integrated Runtime Anomaly Detection for Embedded Systems. *ACM Transactions on Embedded Computing Systems*. 2017; 17(2): 1-27. DOI: https://doi.org/10.1145/3122785

[33] Landauer C, Bellman KL. Detecting anomalies in constructed complex systems. In: *Proceedings of the 33rd IEEE Annual Hawaii International Conference on System Sciences, HICSS 2000, 4-7 January, 2000, Maui, Hawaii, USA*. IEEE; 2000. p.9. DOI: https://doi.org/10.1109/HICSS.2000.926733

[34] Rabiner LR. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*. 1989; 77(2): 257-286. DOI: https://doi.org/10.1109/5.18626

[35] Kuo S, Agazzi OE. Automatic keyword recognition using hidden markov models. *Journal of Visual Communication and Image Representation*. 1994; 5(3): 265-272. DOI: https://doi.org/10.1006/jvci.1994.1024

[36] Thiebaut D, Wolf JL, Stone HS. Synthetic traces for trace-driven simulation of cache memories. *IEEE Transaction of Computers*. 1992; 41(04): 388-410. DOI: https://doi.org/10.1109/12.135552

[37] Thornock NC, Flanagan JK. Using the BACH trace collection mechanism to characterize the SPEC2000 integer benchmarks. *Workload Characterization of Emerging Computer Applications*. 2001; 610: 121-143. DOI: https://doi.org/10.1007/978-1-4615-1613-2_6

[38] Yu SZ, Liu Z, Squillante MS, Xia C, Zhang L. A hidden semi-Markov model for web workload self-similarity. In: *Conference Proceedings of the IEEE International Performance, Computing, and Communications Conference, IPCCC 2002, 3-5 April 2002, Phoenix, AZ, USA*. IEEE; 2002. p.65-72. DOI: https://doi.org/10.1109/IPCCC.2002.995137

[39] Song B, Ernemann C, Yahyapour R. Parallel computer workload modeling with Markov chains. In: *Proceedings of 10th International Conference on Job Scheduling Strategies for Parallel Processing, JSSPP 2004, 13 June 2004, New York, NY, USA*. 2004. p.47-62. DOI: https://doi.org/10.1007/11407522_3

[40] Sapia C. PROMISE: Predicting query behavior to enable predictive caching strategies for OLAP systems. In: *Proceedings of 2nd International Conference on Data Warehousing and Knowledge Discovery, DaWaK 2000, 4-6 September 2000, London, UK*. Springer-Verlag; 2000. p.224-233. DOI: https://doi.org/10.5555/646109.679288

[41] Strang G. The discrete cosine transform. *SIAM Review*. 1999; 41(1): 135-147. DOI: https://doi.org/10.1137/S0036144598336745

[42] Valgrind. *Valgrind Instrumentation Framework*. http://valgrind.org/ [Accessed 15th June 2024].

[43] KCachegrind. *kCacheGrind/CoreGrind*. http://kcachegrind.sourceforge.net [Accessed 15th June 2024].

[44] Haring G, Lindemann C, Reiser M. *Performance Evaluation: Origins and Directions*. Berlin, Germany: Springer; 2000. DOI: https://doi.org/10.1007/3-540-46506-5

[45] Calzarossa M, Marie R, Trivedi KS. System performance with user behavior graphs. *Performance Evaluation*. 1990; 11(3): 155-164. DOI: https://doi.org/10.1016/0166-5316(90)90008-7

[46] Calzarossa MC, Massari L, Tessera D. Workload characterization: A survey revisited. *ACM Computing Surveys*. 2016; 48(3): p.1-43. DOI: https://doi.org/10.1145/2856127

[47] SPEC. *SPEC CPU2006 Benchmark*. http://www.spec.org/cpu2006/ [Accessed 15th June 2024].

[48] McDonell KJ. Benchmark frameworks and tools for modelling the workload profile. *Performance Evaluation*. 1995; 22(1): 23-41. DOI: https://doi.org/10.1016/0166-5316(94)E0036-I

[49] Hsu WW, Smith AJ, Young HC. Characteristics of production database workloads and the TPC benchmarks. *IBM Systems Journal*. 2001; 40(3): 781-802. DOI: https://doi.org/10.1147/sj.403.0781

[50] Intel. *Intel PIN Tool*. http://www.pintool.org/ [Accessed 15th June 2024].

[51] Linde Y, Buzo A, Gray R. An algorithm for vector quantizer design. *IEEE Transactions on Communications*. 1980; 28(1): 84-95. DOI: https://doi.org/10.1109/TCOM.1980.1094577

[52] Devijver PA. Baums forwardbackward algorithm revisited. *Pattern Recognition Letters*. 1985; 3(6): 369-373. DOI: https://doi.org/10.1016/0167-8655(85)90023-6

[53] Nikiforov V. The energy of graphs and matrices. *Journal of Mathematical Analysis and Applications*. 2007; 326(2): 1472-1475. DOI: https://doi.org/10.1016/j.jmaa.2006.03.072

[54] Crovella ME, Bestavros A. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*. 1997; 5(6): 835-846. DOI: https://doi.org/10.1109/90.650143

[55] Pentakalos OI, Menasce DA, Yesha Y. Automated clustering-based workload characterization. In: *NASA Conference Publication, Proceedings of the 5th NASA Goddard Mass Storage Systems and Technologies Conference*. 1996. p.253-263.

[56] Luthi J. Histogram-based characterization of workload parameters and its consequences on model analysis. In: *Proceedings of 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 1998, 19-24 July 1998, Montreal, Canada.* 1998. p.1.

[57] Jeffrey N, Tan Q, Villar JR. A review of anomaly detection strategies to detect threats to cyber-physical systems. *Electronics*. 2023; 12(15): 3283. DOI: https://doi.org/10.3390/electronics12153283

[58] Wang Z, Wei Z, Gao C, Chen Y, Wang F. A framework for data anomaly detection based on iterative optimization in IoT systems. *Computing*. 2023; 105: 23372362. DOI: https://doi.org/10.1007/s00607-023-01186-6

[59] Nikolaou C, Labrindis A, Bohn V, Ferguson D, Artavanis M, Kloukinas C, Marazakis M. The impact of workload clustering on transaction routing. *FORTH, Institute of Computer Science, Technical Report.* 1998; p.238.

[60] Li N, Yu SZ. Periodic hidden Markov model-based workload clustering and characterization. In: *2008 8th IEEE International Conference on Computer and Information Technology, CIT 2008, 8-11 July 2008, Sydney, NSW, Australia*. IEEE; 2008. p.378-383. DOI: https://doi.org/10.1109/CIT.2008.4594705

[61] Cuzzocrea A, Darmont J, Mahboubi H. Fragmenting very large XML data warehouses via K-means clustering algorithm. *International Journal of Business Intelligence and Data Mining*. 2009; 4(3-4): 301-328. DOI: https://doi.org/10.1504/IJBIDM.2009.029076

[62] Bonifati A, Cuzzocrea A. Efficient fragmentation of large XML documents. In: *International Conference on Database and Expert Systems Applications, DEXA 2007, 3-7 September, 2007, Regensburg, Germany.* 2007. p.539-550. DOI: https://doi.org/10.1007/978-3-540-74469-6_53

[63] Bonifati A, Cuzzocrea A. Storing and retrieving XPath fragments in structured P2P networks. *Data & Knowledge Engineering*. 2006; 59(2): 247-269. DOI: https://doi.org/10.1016/j.datak.2006.01.011

[64] Cuzzocrea A, Furfaro F, Masciari E, Saccà D, Sirangelo C. Approximate Query Answering on Sensor Network Data Streams. In: *Stefanidis A, Nittel S. (eds.) GeoSensor Networks*. Boca Raton, FL, USA: CRC Press; 2004; p.53-72. DOI: https://doi.org/10.1201/9780203356869.ch4

[65] Cuzzocrea A, Furfaro F, Saccà D. Hand-OLAP: A system for delivering OLAP services on handheld devices. In: *The Sixth International Symposium on Autonomous Decentralized Systems, ISADS 2003, 9-11 April 2003, Pisa, Italy.* IEEE; 2003. p. 80-87. DOI: https://doi.org/10.1109/ISADS.2003.1193935

[66] Cuzzocrea A, Matrangolo U. Analytical synopses for approximate query answering in OLAP environments. In: *International Conference on Database and Expert Systems Applications, DEXA 2004, August 30-September 3, 2004, Zaragoza, Spain.* 2004. p.359-370. DOI: https://doi.org/10.1007/978-3-540-30075-5_35

[67] Cuzzocrea A, Saccà D, Serafino P. A hierarchy-driven compression technique for advanced OLAP visualization of multidimensional data cubes. In: *Data Warehousing and Knowledge Discovery: 8th International Conference, DaWaK 2006, Krakow, Poland, September 4-8, 2006, Krakow, Poland.* 2006. p.106-119. DOI: https://doi.org/10.1007/11823728_11

[68] Cuzzocrea A, Saccà D. Balancing accuracy and privacy of OLAP aggregations on data cubes. In: *Proceedings of the ACM 13th International Workshop on Data Warehousing and OLAP, DOLAP 2010, 30 October 2010, Toronto, Ontario, Canada.* New York, NY, United States: Association for Computing Machinery; 2010. p.93-98. DOI: https://doi.org/10.1145/1871940.1871960

[69] Cuzzocrea A, Saccà D, Ullman JD. Big data: A research agenda. In: *Proceedings of the 17th International Database Engineering & Applications Symposium, IDEAS 2013, 9-11 October 2013, Barcelona, Spain.* New York, NY, United States: Association for Computing Machinery; 2013. p.198-203. DOI: https://doi.org/10.1145/2513591.2527071

[70] Yousuf S, Kadri MB. A ubiquitous architecture for wheelchair fall anomaly detection using low-cost embedded sensors and isolation forest algorithm. *Computers and Electrical Engineering.* 2023; 105: 108518. DOI: https://doi.org/10.1016/j.compeleceng.2022.108518

Alfredo Cuzzocrea
iDEA Lab
University of Calabria
Rende, Italy
&
Department of Computer Science
University of Paris City
Paris, France
E-mail: alfredo.cuzzocrea@unical.it

Enzo Mumolo
Department of Engineering
University of Trieste
Trieste, Italy
E-mail: mumolo@units.it

Islam Belmerabet
iDEA Lab
University of Calabria
Rende, Italy
E-mail: ibelmerabet.idealab.unical@gmail.com

Abderraouf Hafsaoui

iDEA Lab
University of Calabria
Rende, Italy
E-mail: ahafsaoui.idealab.unical@gmail.com