Detecting Anomalies in Embedded Computing Systems via a Novel HMM-based Machine Learning Approach

Alfredo Cuzzocrea[1], Eric Medvet[2], Enzo Mumolo[2], and Riccardo Cecolin[2]

[1] ICAR-CNR and University of Calabria Rende, Cosenza, Italy
[2] DIA Department, University of Trieste, Italy

Abstract. Computing systems are vulnerable to anomalies that might occur during execution of deployed software: e.g., faults, bugs or deadlocks. When occurring on embedded computing systems, these anomalies may severely hamper the corresponding devices; on the other hand, embedded systems are designed to perform autonomously, i.e., without any human intervention, and thus it is difficult to debug an application to manage the anomaly. Runtime anomaly detection techniques are the primary means of being aware of anomalous conditions. In this paper, we describe a novel approach to detect an anomaly during the execution of one or more applications. Our approach describes the behaviour of the applications using the sequences of memory references generated during runtime. The memory references are seen as signals: they are divided in overlapping frames, then parametrized and finally described with Hidden Markov Models (HMM) for detecting anomalies. The motivations of using such methodology for embedded systems are the following: first, the memory references could be extracted with very low overhead with software or architectural tools. Second, the device HMM analysis framework, while being very powerful in gathering high level information, has low computational complexity and thus is suitable to the rather low memory and computational capabilities of embedded systems. We experimentally evaluated our proposal on a ARM9, Linux based, embedded system using the SPEC 2006 CPU benchmark suite and found that it shows very low error rates for some artificially injected anomalies, namely a malware, an infinite loop and random errors during execution.

1 Introduction

Embedded computing systems are extensively used in every-day life. Their use 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), the fact that it is hard or even impossible to interact with them, and the fact that the number of concurrent executions is limited and very often well known a-priori. Embedded system failures may be caused by software errors (bugs),

faults, or by 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. All of these events could result in run-time anomalies: the ability to automatically detect these anomalies may allow preventing failures in embedded systems, and hence avoiding damages 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 on statistical information extracted from the patterns [1]. Our anomaly detection technique establishes a behavior of the normal executions under examination, compare 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.

In this paper, we propose a technique to build a profile of the behavior of a program and to detect deviations from this profile. The profile is based on a statistical model of the memory references generated during the execution. 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 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 learn, for each application, a Hidden Markov Model of the execution. Then, we compute the likelihood that the sequences observed during the following execution is consistent with the HMM models and we use this figure to detect the anomalies.

This paper is organized as follows. In Sec. 2 we summarize some work done in embedded systems, and in Sec. 3 we report preliminary basic concepts used through the paper. Then, Sec. 4 describes the analysis algorithm and in Sec. 5 we show experimentally that the analysis framework gives good performances when applied to classification. In Sec. 6 we describe some detection experiments with artificial anomalies. In Sec. 7 some final remarks are reported.

2 Related Work

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 [2]. Training a Markov model means learning 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 learned model. In [3] Maxion and Tan present two methods for

detecting anomalies in embedded systems, namely Markov and Stide (Sequence Time Delay Embedding). 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 [4], use Markov Models of system call sequences. In some cases, better models can be obtained with Hidden Markov Models, which are widely used for sequence modeling. Wang et al. report in [5] 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 system. Sugaya et al. describe in [6] an anomaly detection system based on HMM modeling of resource consumption such as CPU, memory and network. In [7], 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. The probabilistic detector employs the probability of events to evaluate the testing sequence. The approaches are suitable for embedded systems because require lower memory size and can be easily implemented in hardware. Some authors, for example [8–10], consider the discrete sequences as signals, and use signal processing techniques to analyze them.

3 Preliminaries

3.1 Spectral Description of Memory References

The short-term Fourier transform, 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 describe how the energy is distributed on a spectral range.

We show hereafter that memory references can be described with spectral parameters. In fact, important parts of a program are composed by loops, that becomes peaks in the spectral domain, as we point out shortly.

Let us consider for example a simple cycle of the type:

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

The virtual memory references sequence generated during the execution of this loop can be modeled with a sawtooth signal. Calling $F(\omega)$ the amplitude spectrum of a single ramp, the analytic form of the sawtooth spectrum is $F(\omega)\sum_n \delta(n-N)$ where N is the wideness of the cycle. The spectrum is therefore composed by a periodic series of peaks with decreasing amplitudes whose period is related to the loop width N.

As a more practical example, let us consider the code of a simple bubble sort, which is basically formed by inner cycles. After acquisition of the virtual

memory sequence, a Short Time Fourier Transform (STFT) described in (1) is performed.

$$X(n) = \sum_{-\infty}^{\infty} x(n)w(n-m)e^{-j\omega n} \tag{1}$$

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 resulting amplitude spectrum over time is reported in Fig. 1. It is worth observing that the spectra peaks represent the wideness of the cycles in the code. Thus, the spectral patterns well characterize the ex-

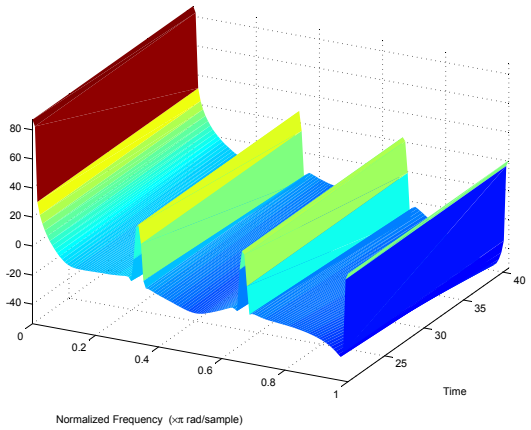

Fig. 1. Spectral representation of a Bubble sort.

ecutions. Indeed, spectral representation is obtained with Fast Discrete Cosine Transform.

3.2 Discrete Cosine Transform Representation

DCT [11] 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.

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. Efficacy 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.

4 The Proposed Algorithmic Anomaly Detection Framework and Its Implementation

4.1 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 addresses blocks; on these blocks a spectral vector is computed with Discrete Cosine Transform (DCT) and from each vector the first sixteen coefficients are extracted. The following step is to perform Vector Quantization with 64 centroids[12] 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 in 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 (HMM) 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 [13], 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.

4.2 Anomaly Detection Algorithm

According to the previous discussion, during the execution of the programs, a series of likelihoods matrices, one for each epoch, are computed:

$$\dots$$

$$L_{k-1} = \begin{vmatrix} P(O_1^{k-1}|\lambda_1) & P(O_2^{k-1}|\lambda_1) & \dots & P(O_N^{k-1}|\lambda_1) \\ P(O_1^{k-1}|\lambda_2) & P(O_2^{k-1}|\lambda_2) & \dots & P(O_N^{k-1}|\lambda_2) \\ \dots & & & \\ P(O_1^{k-1}|\lambda_N) & P(O_2^{k-1}|\lambda_N) & \dots & P(O_N^{k-1}|\lambda_N) \end{vmatrix} \tag{2}$$

$$L_k = \begin{vmatrix} P(O_1^{k}|\lambda_1) & P(O_2^{k}|\lambda_1) & \dots & P(O_N^{k}|\lambda_1) \\ P(O_1^{k}|\lambda_2) & P(O_2^{k}|\lambda_2) & \dots & P(O_N^{k}|\lambda_2) \\ \dots & & & \\ P(O_1^{k}|\lambda_N) & P(O_2^{k}|\lambda_N) & \dots & P(O_N^{k}|\lambda_N) \end{vmatrix} \tag{3}$$

$$\dots$$

where $O_1^k, \dots, O_N^k$ are the observations of the k-th epoch for process 1 to N and $P(O_i^k|\lambda_j)$ is the likelihood that the jth model generates the ith observation, at the kth time epoch.

Our monitoring algorithm is based upon the difference between the Observations. The distance between observation O_i and O_j at epoch k is given by (4):

$$d(O_i^k, O_j^k) = \sqrt{\sum_{n=1}^{N} [P(O_i^k|\lambda_n) - P(O_j^k|\lambda_n)]^2} \tag{4}$$

In the same way, we can compute the distance between observation O_i at epoch $k-1$ and O_j at epoch k. The key of the algorithm is that, in normal executions, the distance between O_i at epoch $k-1$ and O_i at epoch k should be lower than the distances between O_i at epoch $k-1$ and O_j at epoch k, for all the j different from i.

Using for simplicity a vector representation, that is calling $V_{k,i}$ the i-th column of the L_k matrix, $V_{k,i} = |P(O_i^k|\lambda_1), \dots, P(O_i^k|\lambda_N)|^T$, the above mentioned condition means that, in normal conditions the Euclidean distance between $V_{k-1,i}$ and $V_{k,i}$ is less than the distances between $V_{k-1,i}$ and $V_{k,j}$, for $j \neq i$. However, if the execution of the application i is corrupted by an anomaly the condition is not true.

In other words, the algorithm must look if

$$\sqrt{V_{k-1,i} \cdot V_{k,i}^T} = \min_{j=1 \leq N \leq n} \sqrt{V_{k-1,i} \cdot V_{k,j}^T} \tag{5}$$

4.3 Runtime Implementation using PIN

Tools capable of extracting and processing memory traces from running processes have been developed to monitor systems in real-time. We chosen Intel PIN as binary instrumentation framework, as it is freely available on both x86, x86_64 and ARM architectures under a Linux environment. Using the API of PIN [14], 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.

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.

5 Validation of the HMM-based Machine Learning Approach

Before performing experimental evaluations of the detection tool, we performed extensive classification experiments to verify the described analysis framework. Namely, the HMM models computed from each observation are used to see if the observations with the higher 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.

Classification results are reported in Fig. 2 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 to analyze several billion bytes. For this reason we developed a Valgrind tool, that we called Tracehmm, to perform on line all the described processing.

5.1 Valgrind

Valgrind [15] is an instrumentation 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

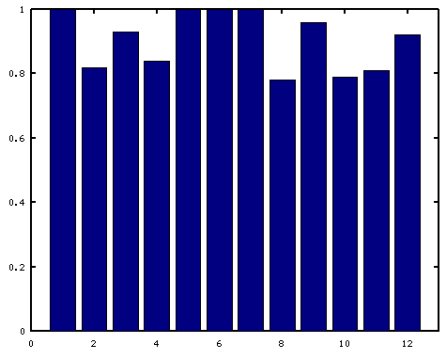

Fig. 2. The figure shows one bar for each application whose height corresponds to the classification accuracy for that application

licence; it is possible to modify the available tools and to modify also the code of the framework itself, i.e. Coregrind. Coregrind [16] 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 using directly 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
```

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 perform also the Viterbi test on a previously trained HMM model for computing the likelihood that the model λ may generate the execution sequence O.

6 Experimental Results

The tool based on PIN described above has been tested in three different experiments. In all the tests of this section, an embedded device equipped with an ARM9 at 500 MHz, 128 Mb RAM and a Debian Linux has been used.

6.1 Experiment 1 - Malware Detection

In this test, 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 (see Appendix for further details on the benchmarks) 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 has 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.

Results shows that the algorithm can determine in an accurate way the epoch that has produced the error, with an error rate below 1%.

6.2 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.

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 87.5%.

6.3 Experiment 3 - Memory Reference Random Error Detection

In this experiment, the memory trace gathered using PIN has been modified by adding a white Gaussian noise. This experiment shows that 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. From 0 dB to 35 dB, all the anomalies where correctly detected, while if the sequence shows a SNR greater that 40 dB, which means really low injected noise, no anomalies were detected.

As a final remark, we observe that some experiments were conducted looking at the values of the single likelihoods $P(O|\lambda)$. Clearly, it can be expected that in presence of an anomaly, the value of the likelihood is modified with respect to the normal executions. However, this procedure leads to a very high number of false positive because it is impossible to set that right threshold. Using the procedure depicted in (5) no thresholds must be found; the false positive are limited by the fact that (5) measures the behaviour of all the applications with respect to the HMM models.

6.4 Overhead Analysis

Most of the complexity relies on the binary instrumentation method which is needed to extract the memory references. We experimentally evaluated that the instrumentation causes a slowdown of about 30%. 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 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, that will begin the analysis of the trace avoiding the instrumenting tool to slow down the application even more.

There are many other possibilities to reduce this overhead, from statistical methods to software tools to architectural methods: in the ARM architectures, for example, there is a register which furnish the running value of the program counter. The other task which requires high computation is the Baum-Welch algorithm, but it is computed once, at the beginning of the executions, to compute the HMM models. The other operations, namely DCT, Vector quantization, forward-backward are much less computationally intensive and it was evaluated that their overhead is less than 5%.

7 Final Remarks and Conclusions

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 executions epochs of each program can be clustered and represented using multidimensional scaling. This modeling technique is the base of the proposed algorithm for anomaly detection, that is capable of accurately determine the epoch where an anomaly has occurred. It is also capable to determine the program subject to the anomaly. In a multi-thread 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 rather limited. Indeed, we obviously plan to further experimentally investigate the algorithm through extensive experiments with different faults and anomaly injections.

References

1. Maxion, R., Tan, K.: Anomaly detection in embedded systems. Computers, IEEE Transactions on **51**(2) (feb 2002) 108 –120

2. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection for discrete sequences: A survey. Knowledge and Data Engineering, IEEE Transactions on **PP**(99) (2010) 1
3. Maxion, R., Tan, K.: Benchmarking anomaly-based detection systems. In: Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on. (2000) 623 –630
4. X. Tan, W. Wang, H.X., Yin, B.: A markov model of system calls sequence and its application in anomaly detection. In: Computer Engineering. Volume 43. (sept. 2002) 189–191
5. P.Wang, B.Wang, Y.L.Y.: Survay on hmm based anomaly intrusion detection using system calls. In: The 5th International Conference on Computer Science and Education. (August 2010) 102–105
6. M. Sugaya, Y.O., Nakajiman, T.: A markov model of system calls sequence and its application in anomaly detection. **3** (July 2009) 35–54,
7. M. Zandrahimi, H. Zarandi, M.M.: Two effective methods to detect anomalies in embedded systems. **43** (Jan. 2012) 77–87
8. Moro, A., Mumolo, E., Nolich, M.: Ergodic continuous hidden markov models for workload characterization. In: Image and Signal Processing and Analysis, 2009. ISPA 2009. Proceedings of 6th International Symposium on. (sept. 2009) 99 –104
9. Moro, A., Mumolo, E., Nolich, M.: Workload modeling using pseudo2d-hmm. In: Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on. (sept. 2009) 1 –2
10. M.M.Z.Zadeh, M.Salem, N.G.S.: Sipta: Signal processing for trace-based anomaly detection. In: Proc. of the Conference on Embedded Software (EMSOFT). (October 2014) 2–10
11. Makhoul, J.: Fast cosine transform in one and two dimensions. **1** (feb. 1980) 27–34
12. Y. Linde, A.B., Gray, R.M.: An algorithm for vector quantizer design. **1** (Jan. 1980) 702–710
13. Devijver, P.A.: Baum's forward–backward algorithm revisited. **3** (Jan. 1985) 369–373
14. Intel: Pin tool. http://www.pintool.org/
15. : Valgrind instrumentation framework. http://valgrind.org/
16. : kcachegrind/coregrind. http://kcachegrind.sourceforge.net/html/Home.html