Signal Processing for Trace-based Anomaly Detection in Embedded Software

by

Mohammad Mehdi Zeinali Zadeh Ranjbar

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

Abstract

Embedded operating systems generate a log of operating system function calls which we refer to as *traces*. Trace-based anomaly detection deals with the problem of determining whether or not an instance of traces represents a normal execution scenario. Most current approaches focus on application areas outside of the embedded systems domain and thus do not take advantage of the intrinsic properties of this domain.

This work introduces *Signal Processing for Trace Based Anomaly Detection* (SiPTA): a novel technique for offline trace-based anomaly detection that utilizes the intrinsic feature of periodicity present in embedded systems. SiPTA uses *discrete-time Fourier transform* which is a crucial tool of signal processing theory as an underlying method. This Thesis describes a generic framework for mapping execution traces to channels and signals for further processing. The classification stage of SiPTA uses a comprehensive set of metrics. As this thesis demonstrates, SiPTA is particularly useful for embedded systems. More specifically, we will compare SiPTA with state-of-the-art approaches to trace-based anomaly detection based on the Markov Model and Neural Networks. This thesis also shows the technical feasibility and viability of SiPTA through multiple case studies using traces from a field-tested hexacopter, a mobile phone platform, and a car infotainment unit. In the experiments, our approach outperformed every other tested method.

Acknowledgements

I would like to firstly thank my supervisor, Professor Sebastian Fischmeister, for his continuous guidance and support during my MASc. Also I would also like to extend special thanks to my parents for their unconditional support throughout my life, particularly during my various academic pursuits.

Dedication

This is dedicated to my "little diamond": my niece, *Elina*.

Table of Contents

List of Tables

List of Figures

List of Illustrations

Chapter 1

Introduction

In this chapter we explore the reasons why anomaly detection is necessary as a layer for error detection. Also, we will examine the structure of the rest of my thesis.

1.1 Motivation

The only way to ensure that the system theoretically behaves according to its specifications is complete formal model checking. Formal model checking is not practical on large systems as it requires exponential resources on the size of the system [16]. Even if we formally verify the software there is no guarantee that hardware misbehavior would not occur. Consequently, there is no guarantee that systems will not misbehave.

These errors can lead to catastrophic failures, especially in safety critical systems such as Therac-25 [27] and Ariane 5 flight [28] These systems are typically used in medical, automotive, and energy domains. For safety issues, these systems must comply with standards that require the use of software monitors to detect anomalies in the development and production phases, e.g., ISO-26262 [9] for automotive functional safety and DO-178C [23] for airborne systems.

Safety-critical system usually have a facility to produce logs of operating system calls. These logs are referred to as *traces*. In other words, a trace is a timed log of operating system calls. In such systems, *trace-based anomaly detection* can act as a monitoring mechanism and invoke modules responsible for prevention and recovery from failures. In essence, trace-based anomaly detection aims at detecting execution patterns that do not conform to the normal system functionality. Anomalous patterns can indicate software bugs or malfunctions, which is convenient as such an analysis treats the system under scrutiny as a black box. In other words trace-based anomaly detection does not require knowledge of system internals. This approach leverages the computational power of computer processors to distinguish anomalous traces from normal traces, and the likelihood of catastrophic failures [45].

The majority of safety-critical systems are embedded systems. Also, we know that the behavior of embedded software is periodic [35]. On the other hand, the many signal processing methods work on the periodic features of signals. Hence these methods and algorithms can (and as we will show, will) be well-suited for anomaly detection in embedded systems.

1.2 Contributions

The main contribution of this thesis is the introduction of SiPTA, which realizes a novel technique for offline program trace-based anomaly detection utilizing the intrinsic feature of periodicity found in embedded systems. As we will later explain, SiPTA uses signal processing algorithms to identify periodic features in an embedded system. Other contributions are outlined as follows:

- Introducing the concept of using the intrinsic system periodicity for trace analysis.
- Demonstrating the feasibility and viability of using signal processing algorithms for trace analysis.
- Formalizing a generic framework for modeling traces and mapping them to signals and channels.
- Specifying a comprehensive set of metrics based on frequency spectra useful for the classifying traces.

In our study we restrict our attention to traces obtained from embedded devices. We evaluate SiPTA through the analysis of sets of application-specific traces generated from QNX RTOS [6] running on deployed commercial platforms (i.e., a hexacopter platform, a car infotainment system, a phone OS, and an embedded development test kit) covering a broad range of execution scenarios. We chose to create our dataset, since the established datasets [7, 1] are ill-suited to embedded systems and are under criticism [31, 17].

1.3 State of the Art

There are several state-of-the-art approaches to trace-based anomaly detection. As we will be using these methods as a bar to compare our method to, we will briefly examine them.

Firstly, however, it is necessary to secure knowledge of anomaly detection and trace-based anomaly detection. Following these descriptions, we will examine the limitations of and challenges to anomaly detection and trace-based anomaly detection.

1.3.1 Anomaly Detection

Anomaly detection is the problem of detecting whether or not system specifications define properties of the system. In other words, anomaly detection observes whether or not patterns in system behavior deviate from expected system behavioral patterns [13]. In the case that system specifications do not define system properties, we say that the system is *anomalous*. Also, we say that the data on which we detected the anomalies is *dirty*. Otherwise, we refer to the system state as *normal* and the data used for this detection is *clean*.

Figure 1.1 illustrates this point. Let $\mathbb{R}^2$ be the domain of system properties. And also, let the regions $N_1 \subset \mathbb{R}^2$ and $N_2 \subset \mathbb{R}^2$ be the sets of normal regions. For every point $n \in N_1 \cup N_2$, n reflects a normal state of the system. Points as a_1 and a_2, which are outside of the normal regions, are anomalous system states.

Anomaly detection has a vast range of applications. For example *intrusion detection*, which is an application of anomaly detection, is based on detecting anomalies in network traffic of a website or server [26]. *Mammographic computer-aided diagnosis*, which is the diagnosis of breast cancer, can be done by anomaly detection in MRI images of people [39]. The applications even extend to *credit card fraud detection* [11] and *space craft sensor reading anomaly detection* [20] and many more.

1.3.2 Trace-based Anomaly Detection

The focus of this work is *trace-based anomaly detection*. In other words, *traces* of the target system are the means of extracting the target system specification and properties.

To have a grasp on how system specifications and properties relate to traces, we should define what traces are. Informally speaking, a trace is the sequence of system calls which

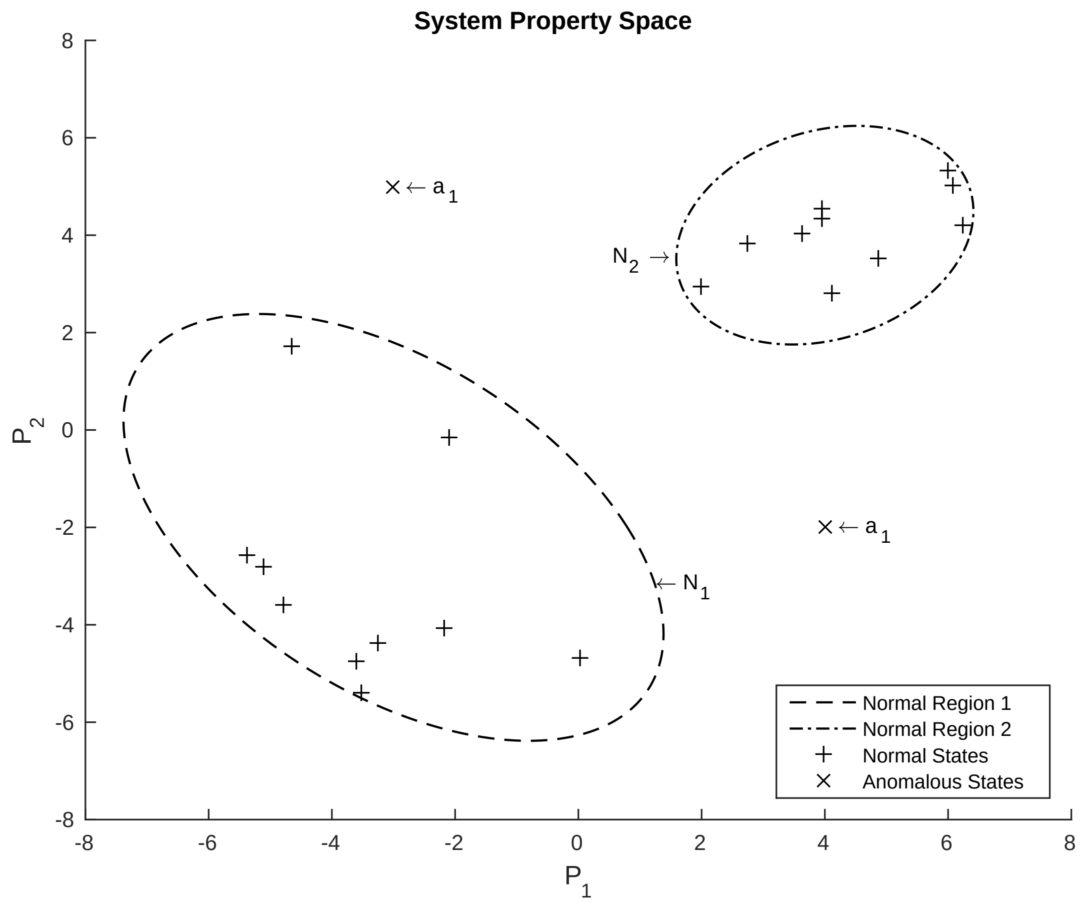


Figure 1.1: System properties

are generated by the operating system trace logger. Each entry of the system call contains such information as the time of the system call, the class of the system call, the system call name, the processor number that processed the system call, the process name and the process ID of the system call.

To model the data structure that defines traces, we use a bottom-up scheme. In other words, we respectively model system calls, its attributes, trace entries which include the real time, and then traces.

Definition 1 (Parameters) *Let $\mathbb{P}_0$ be the set of possible events. These events represent the type of operating system calls invoked by processes, user, or operating system itself. Also, Let the events have $m-1$ possible attributes and let $\mathbb{P}_i$ be the set of possible values* *for i^{th} attribute for $1 \leq i \leq m-1$. With this, a parameter is an array of length m where its i^{th} member is selected from $\mathbb{P}_i$, i.e, $P = \langle \rho_0, \ldots, \rho_{m-1} \rangle$, where $\forall 0 \leq i \leq m-1, \rho_i \in \mathbb{P}_i$, or equivalently $\forall 0 \leq i \leq m-1, P(i) \in \mathbb{P}_i$.*

As previously mentioned, events occur at a specific and measurable time. In order to complete the information about an event, we need to include the time information as well. We refer to the collection of the parameter and its corresponding time as *trace entry.*

Definition 2 (Trace Entry) *A trace entry E is a structure of two members, real time t, and parameter list P. We say that trace entry E is defined over $\{\mathbb{P}_0, \ldots, \mathbb{P}_{m-1}\}$, if and only if $|E.P| = m$, and $\forall 0 \leq i \leq m-1, E.P(i) \in \mathbb{P}_i$. Also $E.t$ is the real time associated with the trace entry, i.e., the real time in which $E.P(0)$ with attributes $E.P(1 \ldots m-1)$ occurred.*

A sequence or complete record of an operating system run generates many trace entries. Definition 3 defines *trace* as this sequence.

Definition 3 (Trace) *A trace T over parameter sets $\{\mathbb{P}_0, \ldots, \mathbb{P}_{m-1}\}$ of length n over parameter sets $\{\mathbb{P}_0, \ldots, \mathbb{P}_{m-1}\}$ is a sequence of trace entries E_i for $i = 1 \ldots n$ over parameter sets $\{\mathbb{P}_0, \ldots, \mathbb{P}_{m-1}\}$.*

$$T = \langle E_1 \ldots E_n \rangle$$

or equivalently,

$$\forall 1 \leq i \leq n, T(i) = E_i$$

Also, as the trace is recorded during time, the time values of its entries are non-decreasing, i.e.:

$$\forall 1 \leq i < j \leq n : E_i.t \leq E_j.t$$

Example 1 provides a better grasp on trace-related definitions.

Example 1 (Trace) *Consider the usage log of some read-only file. The events that this log file can contain are* `open`*,* `read`*, and* `close`*. Each event is also invoked by some process, which can be defined by its process-id. In this scenario* $\mathbb{P}_0 = \{\texttt{open}, \texttt{read}, \texttt{close}\}$ *and* $\mathbb{P}_1$ *is the set of all present process-ids.*

Table 1.1 shows a sample trace on this simple scenario, with $\mathbb{P}_1 = \{3000, 4000\}$*.*

In this case, $T(1).P = T(7).P = \langle \texttt{open}, 3000 \rangle$*,* $T(5).t = 1110$*.*

Index	*Time (μS)*	*Event*	*PID*
1	1100	`open`	3000
2	1103	`read`	3000
3	1105	`read`	4000
4	1109	`read`	3000
5	1110	`read`	3000
6	1112	`close`	3000
7	1113	`open`	3000
8	1118	`read`	4000
9	1121	`close`	4000

Table 1.1: Trace example

Now let's assume that $\mathbb{T}_c$ is a set of clean traces. In other words, $\mathbb{T}_c$ represents a system that had properties according to its specifications. If we have such a set, we can conceptually estimate the system specifications. Considering the fact that in this work we assume that we do not have any prior knowledge of system specifications, $\mathbb{T}_c$ is the only source for knowing system specification in trace-based anomaly detection. A system can use this knowledge to determine whether or not a new trace like T_a is clean.

1.3.3 Trace-based Anomaly detection using Markov Models

In the context of detection of anomalies from sequential traces, most of research is focused on the anomaly detection of operating system events. A recent survey paper [14] summarized the progress in that research area. They discussed two major approaches. The first method uses Markovian modeling [44] to study the probabilistic characteristics of event transitions. This approach extends from first-order to higher-order Markov Models and some equivalent methods as probabilistic suffix trees (PST) and sparse Markov trees

(SMT). The second approach models the event transition states through Finite State Automata (FSA) and Hidden Markov Models (HMM) methods. Example 2 illustrates how Markov chains model the trace in Table 1.1.

Example 2 (Markov Models) *Assume the possible states of trace entries in Table 1.1. If we create these states via concatenation of* event *and* PID*, we will have the following five states in S:*

$$S = \{\texttt{open::3000}, \texttt{read::3000}, \texttt{read::4000}, \texttt{close::3000}, \texttt{close::4000}\}$$

During the training phase, the system estimates the transition probability between each of these states, and it will construct a Markov chain accordingly. Figure 1.2 depicts this particular Markov chain, which gives the anomaly detection a reference of how the transitions should behave. For example, high frequency of transition $\texttt{close::3000} \rightarrow \texttt{read::3000}$ *in a system, considering the fact that the probability of this transition is* 0*, could signify a dirty trace.*

Different utilization of Markov chains as explained in [14] is so far one of the most promising approaches to trace-based anomaly detection. That is why we will use these approaches as a benchmark for analyzing the performance of SiPTA.

1.3.4 Limitations

System traces are not a perfect representation of system properties. Additionally, a set of clean traces is not representative of clean traces as a whole. These issues and also the nature of anomaly detection impose the following limitations to trace-based anomaly detection:

- As system traces are not the perfect representation of system properties, meaning that they give only partial information about system properties, it is possible that some anomalies do not have an impact on dimensions of system traces and instead have an effect on the dimensions that are not represented by system traces. In other words, the anomaly shows the information about system properties that is not represented by traces.

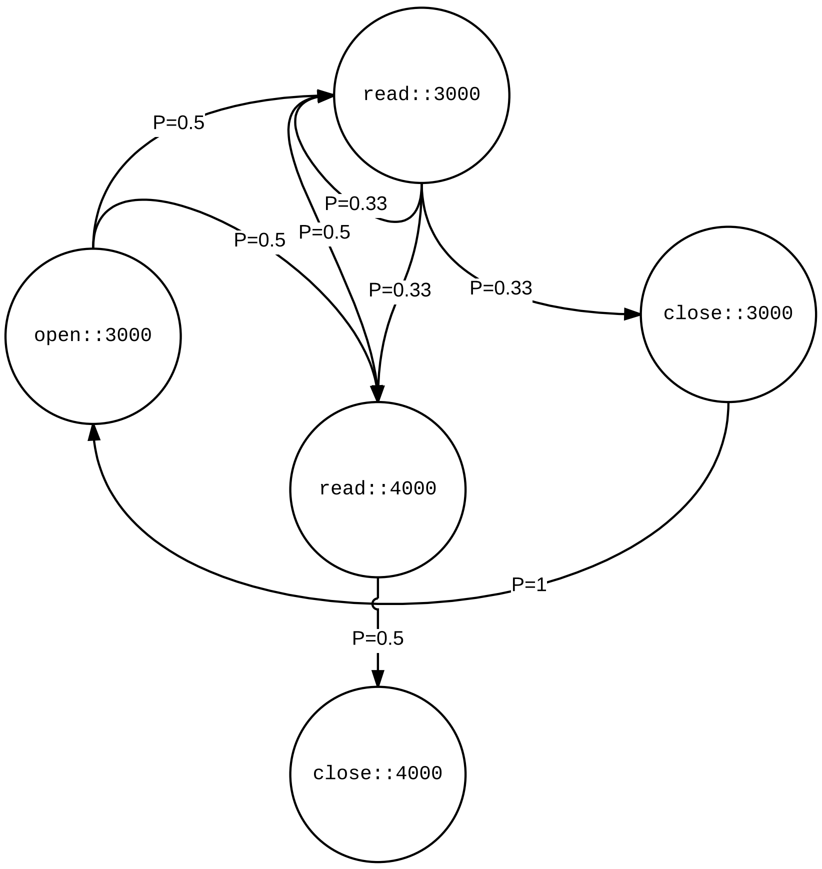


Figure 1.2: Markov chain example

For example, the two regions N_1 and N_2 in Figure 1.1 are not perfectly normal regions. P_3 could be an additional dimension in system properties that is not equivalent to P_1 or P_2. P_3 could be an indicator whether or not system behavior is normal. Put differently, the clean traces that belong to one of the two regions, merely have a low probability of representing an anomalous system.

- Because we do not have prior knowledge of system specifications, our knowledge about system specification is limited to training involvingthe clean trace set. In other words, we cannot have a perfect knowledge of the subsets of property space that represent clean traces. If we wanted to do a similar thing conceptually, we should have a set like $\mathbb{T}_{\mathbb{A}}$ which is the set of all clean traces [13].

In addition to these limitations, [13] introduces a list of challenges. The following challenges from this reference are relevant to trace-based anomaly detection:

- Anomalies can be different in different domains. This fact conceptually affects the size of the normal regions [13]. In other words, a more lenient notion of anomaly results in bigger normal regions and more restrictive assumptions mean smaller normal regions.
- This method requires available sets of traces labeled as clean or dirty. these sets are not always accessible and can become problematic [13].

1.4 Thesis Structure

So far we have talked about our aspirations and contributions. Furthermore, we detail leading approaches to trace-based anomaly detection as well as their limitations.

In Chapter 2 we review the theoretical background knowledge that SiPTA utilizes to approach trace-based anomaly detection. Subsequently, Chapter 3 thoroughly examines how SiPTA approaches this problem. In Chapter 4 we explain the setup for experimenting. The remaining of the Chapters are dedicated to synthesizing the results from our experiments and its implications.

Appendix A provides a brief overview of how to setup our implementation of SiPTA in MATLAB.

Chapter 2

Background

There are several concepts utilized in SiPTA: Fourier transform, fast Fourier transform, major frequency component detection, minimum weight perfect bipartite matching and hierarchical clustering. This chapter aims to give an in-depth examination of how these methods work. Should you possess prior knowledge of these concepts, you can proceed to only read the notations and skip the thorough explanations.

2.1 Fast Fourier Transform

In order to explain *fast Fourier transform*, we need a brief explanation on *time series* and *discrete-time Fourier transform*.

Definition 4 (Series) *Let S be a sequence of complex numbers of length n. It means that $\forall 0 \leq j \leq n-1, S(j) = x + yi$ where $x, y \in \mathbb{R}$ and $i = \sqrt{-1}$. In this case S is a* series*.*

Also, if $\forall 0 \leq j \leq n-1, imag(S(j)) = 0$, where $imag(a)$ is imaginary part of a, S is a real series*.*

If the series S is timed, that is where $\forall 0 \leq j \leq \text{length}(S) - 1, S(j)$ is associated with time j in some time domain, we refer to S as time series.

Now, we can define *discrete Fourier transform* (DFT).

Definition 5 (Discrete Fourier Transform) *Let S be a time series of length n. And let $\mathcal{F}()$ be* discrete Fourier transform *which takes a time series as input and returns the* *frequency series of the input time series. If $F = \mathcal{F}(S)$, then the following conditions will hold.*

- length$(F) = n$, *or in other words, f has the same length is S.*
- $\forall 0 \leq j \leq n-1, F(j) = \sum_{k=0}^{n-1} S(k) e^{-i\frac{2\pi}{n}jk}$, *or according to Euler's formula*[1] $\forall 0 \leq j \leq n-1, f(j) = \sum_{k=0}^{n-1} S(k)(\cos(\frac{2\pi}{n}jk) - i\sin(\frac{2\pi}{n}jk))$.

Note that $F(j)$ is multiplication of the time series with $\cos(\frac{2\pi}{n}jk) - i\sin(\frac{2\pi}{n}jk)$ which is a sinusoidal expression with frequency of $\frac{j}{n}$.

Theorem 1 (Inverse Discrete Fourier Transform) *DFT is invertible and if inverse DFT is denoted by $\mathcal{F}^{-1}$ and $S = \mathcal{F}^{-1}(F)$, the following will hold.*

- length$(S) =$ length(F).
- $\forall 0 \leq j \leq n-1, S(j) = \sum_{k=0}^{n-1} F(k) e^{i\frac{2\pi}{n}jk}$, *or according to Euler's formula* $\forall 0 \leq j \leq n-1, S(j) = \sum_{k=0}^{n-1} F(k)(\cos(\frac{2\pi}{n}jk) + i\sin(\frac{2\pi}{n}jk))$.

You can find the proof of Theorem 1 in [32]

Definition 6 (Series Orthogonality) *We say that series g_1 and g_2 are* orthogonal, *if and only if the following holds.*

- length$(g_1) =$ length(g_2)
- $\sum_{j=0}^{\text{length}(g_1)-1} g_1(j) g_2(j) = 0$

Theorem 2 (Different Sinusoidal Orthogonality) *Let, $0 \leq j_1 < j_2 \leq n-1 \in \mathbb{Z}$. Also, let g_1, g_2, g_3, and g_4 be series of length n according to the following definition.*

$$
\begin{aligned}
g_1(k) = &\ \cos(\tfrac{2\pi}{n}j_1k) \\
g_2(k) = &\ \sin(\tfrac{2\pi}{n}j_1k) \\
g_3(k) = &\ \cos(\tfrac{2\pi}{n}j_2k) \\
g_4(k) = &\ \sin(\tfrac{2\pi}{n}j_2k)
\end{aligned}
$$

In any combination of j_1 and j_2, any two series from $\{g_1, g_2, g_3, g_4\}$ are orthogonal

[1] $e^{i\phi} = \cos\phi + i\sin\phi$

You can find a similar proof of Theorem 2 in [40]. Theorem 2 implies that any two different frequencies used in DFT and inverse DFT formula are orthogonal. It means that a time series of length n, can be uniquely rewritten as weighted summation of orthogonal sinusoidal time series of the same length with frequencies of $0, \frac{1}{n}, \frac{2}{n}, ..., \frac{n-1}{n}$. As it is evident in Theorem 1, the coefficients of these sinusoids are determined by frequency series. In other words, the i^{th} element of f determines the magnitude and phase of frequency component i/n in the corresponding time series S of length n.

It is obvious that direct calculation of DFT and inverse DFT is $O(n^2)$. In spite of that, it is possible to calculate DFT and inverse DFT using in $O(n \log(n))$ time using Fast Fourier Transform (FFT) algorithm [32].

If the time series of the signal is a real series, then DFT of the time series is symmetric [32].

Theorem 3 (DFT Symmetry of Real Time-Series) *Let S be a real time series of length n, and let $f = \mathcal{F}(S)$. In this case, $\forall 0 \leq j \leq n-1, F(j) = F(n-j)^*$, where for complex number a, a^* is the conjugate of a, i.e., if $a = x + iy$, then $a^* = x - iy$.*

This theorem implies that if f is the frequency series of a real time series of length n, we only need $f(0...\lfloor\frac{n}{2}\rfloor)$ to have all the information representing f and $\mathcal{F}^{-1}(f)$. As SiPTA deals with real time series, Theorem 3 becomes necessary as it will reduce the calculations and memory usage, and also simplify the algorithms. So, in this thesis the notation $\mathcal{F}(S)$ refers to $F(0...\lfloor\frac{n}{2}\rfloor)$.

2.2 Major Frequency Component Detection

As mentioned in Section 2.1 on Page 10, FFT gives the magnitude and phase of different frequency components of a time series. In other words, if the time series S has a length of n, and $F = \mathcal{F}(s)$, then $|F(i)|$ represents the magnitude of frequency component $\frac{i}{n}$ in S. As we later demonstrate, SiPTA will extract the most significant frequency components of some time series; However, not all frequency components are essential. In fact, SiPTA will omit some of the frequency components in order to extract more significant ones. To achieve this goal, SiPTA uses the approach proposed in [41]. In this section we will briefly describe this approach.

Definition 7 (Periodogram) *Let S be a real time series of length n and $F = \mathcal{F}(S)$. In this case, p is the periodogram of F if it is the squared of* $\text{length}(F)$*, i.e.,*

$$\forall 0 \leq i \leq \lfloor \frac{n}{2} \rfloor, p(i) = |F(i)|^2$$

or in shorter form

$$p = |F|.^2$$

In this case we say $p = \texttt{periodogram}(S)$.

Periodograms are important as they are correlative not with the magnitude, but with the energy of frequency components.

The proposed method in [41] takes a confidence interval value $0 < \texttt{ConfIntv} < 1$, number of permutations $\texttt{PermCnt}$ and time series s of length n and returns a threshold t which is a minimum value for a local maximum in p to be qualified as peak which is not created due to random processes with confidence interval of c.

This method creates $\texttt{PermCnt}$ random permutations of S, $\{S_0, ..., S_{\texttt{PermCnt}-1}\}$, i.e.,

$$\forall 0 \leq i \leq \texttt{PermCnt} - 1, S_i = \texttt{permute}(S)$$

Then it calculates their periodograms $\{P_0, ..., P_{\texttt{PermCnt}-1}\}$. i.e.,

$$\forall 0 \leq i \leq \texttt{PermCnt} - 1, P_i = \texttt{periodogram}(S_i)$$

Then it constructs a periodogram matrix, $\mathbf{P}_{\texttt{PermCnt}\times(\lfloor\frac{n}{2}\rfloor+1)}$ which is by vertical concatenation of row vectors $\{P_0, ..., P_{\texttt{PermCnt}-1}\}$:

$$\mathbf{P}_{\texttt{PermCnt}\times(\lfloor\frac{n}{2}\rfloor-1)} = \begin{bmatrix} P_0 \\ P_1 \\ \vdots \\ P_{\texttt{PermCnt}+1} \end{bmatrix} \tag{2.1}$$

After that, the columns of the matrix $\mathbf{P}_{\texttt{PermCnt}\times(\lfloor\frac{n}{2}\rfloor+1)}$ will be sorted in ascending order. Then t will be the $\lfloor c \times \texttt{PermCnt} \rfloor$ line of $\mathbf{P}$:

$$t = \mathbf{P}(\lfloor c \times \texttt{PermCnt} \rfloor, :)$$

Peaks are then defined as regions of p where $p > t$. These peaks are not the product of normal random processes with confidence interval of `ConfIntv`. The reason that this process for peak selection is important is that random processes tend to generate random and unwanted peaks in the periodogram [41].

2.3 Minimum-Weight Perfect Bipartite Matching

In this section we introduce the problem and solution of finding *minimum-weight perfect bipartite matching*. For this reason we present bipartite graphs and its different subsets, and then we go through the problem of minimum-weight perfect bipartite matching.

Definition 8 (Bipartite Graphs) *Let $G = \{V, E\}$ be a graph. This graph is* bipartite *if we can partition the vertex set V into two sets V_1 and V_2 (the bipartition) such that no edge in E has both endpoints in the same set of the bipartition. Formally,*

$$\exists V_1, V_2, V_1 \cap V_2 = \emptyset \wedge V_1 \cup V_2 = V \wedge (\forall (v_1, v_2) \in E, (v_1 \in V_1 \Rightarrow v_2 \in V_2) \wedge (v_1 \in V_2 \Rightarrow v_2 \in V_1))$$

Example 3 gives an example of bipartite graphs.

Example 3 (Bipartite Graphs) *Let $G = \{V, E\}$ depicted as in Figure 2.1. Also, let $V = \{v_i | 1 \le i \le 7\}$, $V_1 = \{v_1, v_2, v_3, v_4\}$, $V_2 = \{v_5, v_6, v_7\}$ and $E = \{(v_1, v_5), (v_1, v_6), (v_2, v_5), (v_2, v_7), (v_3, v_5), (v_3, v_7), (v_4, v_6)\}$. Under these assumptions, G is a bipartite graph.*

Definition 9 (Complete Bipartite Graph) *Let $G = \{V_1 \cup V_2, E\}$ where $V_1 \cap V_2 = \emptyset$. The graph G is a* complete bipartite graph *if $E = \{(v_1, v_2) | v_1 \in V_1 \wedge v_2 \in V_2\}$. In other words, a complete bipartite graph is a bipartite graph with every edge from its left sub-vertices to its right sub-vertices. Example 4 gives an example of a complete bipartite graph.*

Example 4 (Complete Bipartite Graph) *Let $G = \{V, E\}$ be the graph in Figure 2.2. Also, let $V = \{v_i | 1 \le i \le 5\}$, $V_1 = \{v_1, v_2, v_3\}$, $V_2 = \{v_4, v_5\}$ and $E = \{(v_1, v_4), (v_1, v_5), (v_2, v_4), (v_2, v_5), (v_3, v_4), (v_3, v_5)\}$. Under these assumptions, G is a complete bipartite graph.*

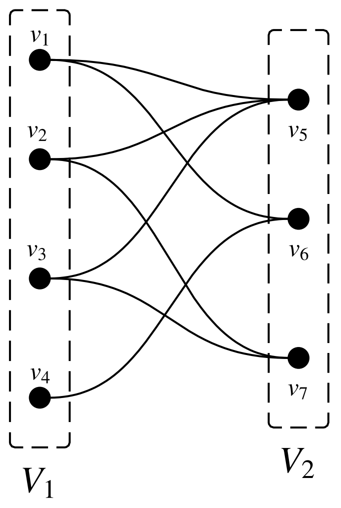


Figure 2.1: Bipartite graph example

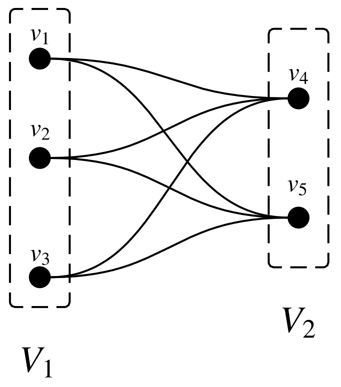


Figure 2.2: Complete bipartite graph example

Now that we have the definitions for bipartite graphs, we can define and address the problem of perfect matching. A *matching* $M \subseteq E$ is a collection of edges such that every vertex of V is incident to at most one edge of M. If a vertex v has no edge of M incident to it then v is said to be *exposed* (or unmatched). A matching is *perfect* if no vertex is exposed; in other words, a matching is perfect if its cardinality is equal to length(V_1) = length(V_2). It is obvious that a perfect matching is possible if length(V_1) = length(V_2).

In this area SiPTA needs to solve the following problem.

Definition 10 (Minimum Weight Perfect Bipartite Matching Problem) *Given a weighted complete bipartite graph* $G = \{V, E, w\}$ *with bipartition* V_1 *and* V_2 *such that* $\text{length}(V_1) = \text{length}(V_2)$, *and given the weight function* $w : E \to \mathbb{R}^+ \cup \{0\}$, *find a perfect matching of minimum cost where the cost of a matching* M, $c(M)$ *is given by the following formula.*

$$c(M) = \sum_{e \in M} w(e)$$

This problem is also known as *assignment problem*, where there are n workers and n tasks. Each worker has a cost for each task and each worker is willing to complete only one task. The goal is to every task with minimum overall cost.

SiPTA uses the solution presented in [25] which is known as *Hungarian method*. This algorithm runs in $O(n^3)$ time where $n = \text{length}(V)$

2.4 Hierarchical Clustering

Imagine that we have a set of N points in some space. Let this set be denoted as $P = \{p_0, ..., p_{N-1}\}$. Also, the function distance measure the distance of a pair of points, i.e., $\forall p, p' \in P, \text{distance}(p, p')$ return the distance between two points p and p'. This distance can be Euclidean distance or any general metric[2] for distance.

Then, distance $N \times N$ matrix D is formed via Equation 2.2.

$$\forall 0 \leq i, j \leq N - 1, D_{ij} = \text{distance}(p_i, p_j) \tag{2.2}$$

Hierarchical clustering [19] algorithm adheres to the following procedure

1. Assign each element to its own cluster. And assign cluster distance $D_C = D$. Also, assign $n = N$ which represent the number of clusters. In addition, let H be a cluster tree of height N and assign the current clusters to $H(N - 1)$.

2. Find the closest pair of clusters and merge them into a single cluster.

[2]A generic distance $d : N \times N \to \mathbb{R}$ must follow the following properties: $d(x, y) \geq 0$, $d(x, y) = 0 \Leftrightarrow x = y$ and $d(x, z) \leq d(x, y) + d(y, z)$

3. Compute D_C according to new clusters. If D_C was of size $n \times n$, now it should be $(n-1) \times (n-1)$. Assign $n = n - 1$ and assign the current cluster to $H(n-1)$.

4. If $n > 1$ go to step 2. Otherwise clustering is finished.

Step 2 can be done using different methods. Conventional methods calculate the centroid distance of clusters, the minimum distance of clusters, or the maximum distance of clusters.

Chapter 3

Proposed Method

This chapter aims to examine the proposed method used in SiPTA thoroughly. For this reason, we first take a look at an overview of SiPTA, and then we will review each part of SiPTA in more details.

This chapter examines methods employed by SiPTA. We begin with an overview of SiPTA and proceed to explore this framework in greater depth by in-depth examination of every SiPTA part.

3.1 Overview

Our work applies to detecting anomalies by analyzing traces from systems with recurring periodic processes, as are often found in the embedded systems domain [35].

Figure 3.1 shows a generic work-flow of SiPTA. In this workflow a set of traces, which can be training traces or analyzing traces, are passed to SiPTA as input. In the *preprocessing* phase traces are converted to *time series* and *spectra*. Then, the *feature extraction* phase, extracts different *features* from those time series and spectra. Next, that in the *training* phase *normal feature constraints* are extracted from training features that represent constraints on features for the corresponding trace to be clean. In the *analysis* phase SiPTA uses normal feature constraints and analyzing features to assign clean and dirty labels to the traces corresponding to the analyzing features. This phase does so by checking if the features of input analyzing trace fit the constraints extracted in the training phase. If it fits, the label for that trace is clean, otherwise the label is dirty.

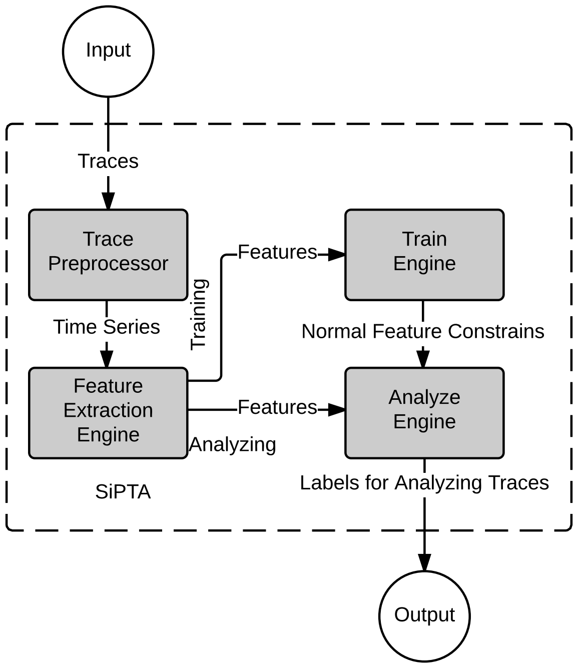


Figure 3.1: SiPTA overview

SiPTA takes advantage of the periodic behavior exhibited in the traces [29] using signal processing algorithms and methods. In the area of anomaly detection, previous works also exploit mathematical concepts that are primarily used in signal processing. For example [21, 15, 38, 36, 30] use wavelets and wavelet transform as a means for anomaly detection. Also, [46] utilizes Fourier transform as means of anomaly detection. Their work focuses on detecting anomalies in network-based systems. In contrast, SiPTA introduces the idea of using frequency characteristics of event transitions.

Given this information, we can explore different parts of SiPTA in detail.

3.2 Preprocessing

As mentioned in Section 1.3.2, traces are a sequence of *trace entries*. Also, trace entries are structured with two members: one member represents the event and another member represents the time in which that event occurred.

To apply signal processing algorithms to traces, we need to model traces with signals and channels. For this reason SiPTA needs to create multiple channels for the signal associated with the trace. Then the time series for each channel should be constructed.

Definition 11 (Channel Names) *Let $\mathbb{T}^t$ be the set of training traces of size M. In other words, $\mathbb{T}^t = \{T_0^t, ..., T_{M-1}^t\}$. Also, let all traces in $\mathbb{T}^t$ be over parameter sets $\{\mathbb{P}_0, \ldots, \mathbb{P}_{M-1}\}$.*

So, we know that the following holds:

$$\forall T \in \mathbb{T}^t, \forall 0 \leq n \leq |T| - 1, T(n).P \in \mathbb{P}_0 \times \ldots \times \mathbb{P}_{M-1}$$

In other words, any parameter of any entry of any of these traces are over $\{\mathbb{P}_0, \ldots, \mathbb{P}_{M-1}\}$.

Also, let $N \subseteq \mathbb{P}_0 \times \ldots \times \mathbb{P}_{m-1}$ be the smallest subset in which $\forall T \in \mathbb{T}^t, \forall 1 \leq n \leq |T|, T(n).P \in N$ holds, meaning, it contains every parameter from every entry of every trace in $\mathbb{T}^t$, but nothing more. In this case, we refer to N as the set of channel names associated with $\mathbb{T}^t$.

Now, for the sake of simplicity, we treat these channel names as channel objects from which SiPTA can extract various information such as their corresponding time series in some trace.

Definition 12 (Channel Objects) *Let N be the set of channel names for some trace set $\mathbb{T}^t$. We define the following fundamental members for any $n \in N$:*

- *n: A property from which returns the actual value of the channel name.*
- *$n.\texttt{exists}(T))$: A Boolean function for determining the existence of n in T. In other words,*

$$n.\texttt{exists}(T) = \begin{cases} \texttt{true} & \exists i, T(i).P = n \\ \texttt{false} & otherwise \end{cases}$$

- *$n.\texttt{timeSer}(T)$: A function that returns a vector containing time series associated with n from T.*

Now, with the foundation of channel objects, we will explain the procedure for building the time series of these channels.

Definition 13 (Channel Time Series) *Let N be the set of channel names associated with trace set $\mathbb{T}$. For some $n \in N$ and $T \in \mathbb{T}$, let $n.\mathtt{exists}(T) = \mathtt{true}$. Then, let I_n be all T indices where $\forall i \in I_n, T(i).P = n$. Let t_n be the sequence of times in which $T(i).P = n$, i.e., $t_n = T(I_n).t$. Then, time series associated with n will be defined as $n.\mathtt{timeSer}(T) = \mathit{diff}(t_n)$, where diff() is the difference function.*

Example 5 gives an example on Definitions 11, 12, and 13.

Example 5 (Time Series) *If $\mathbb{T}$ contains only the trace given in Table 1.1, and N is the set of channel names associated with $\mathbb{T}$, the following holds.* $N = \{\langle \mathtt{open}, 3000\rangle, \langle \mathtt{read}, 3000\rangle, \langle \mathtt{close}, 3000\rangle, \langle \mathtt{read}, 4000\rangle, \langle \mathtt{close}, 4000\rangle\}$

n	I_n	t_n	$n.\mathtt{timeSer}(T)$
$\langle \mathtt{open}, 3000\rangle$	$\langle 1, 7\rangle$	$\langle 0, 13\rangle$	$\langle 13\rangle$
$\langle \mathtt{read}, 3000\rangle$	$\langle 2, 4, 5\rangle$	$\langle 3, 9, 10\rangle$	$\langle 6, 1\rangle$
$\langle \mathtt{close}, 3000\rangle$	$\langle 6\rangle$	$\langle 12\rangle$	$\langle\rangle$
$\langle \mathtt{read}, 4000\rangle$	$\langle 3, 8\rangle$	$\langle 5, 18\rangle$	$\langle 13\rangle$
$\langle \mathtt{close}, 4000\rangle$	$\langle 9\rangle$	$\langle 21\rangle$	$\langle 12\rangle$

Definition 14 (Spectrum) *Let the set of channel name objects N be defined upon some set of traces $\mathbb{T}$. Let P_n be the periodogram of some channel name $n \in N$ in some trace $T \in \mathbb{T}$. Also, let $n.\mathtt{spectrum}(T)$ represent spectrum of $n.\mathtt{timeSer}(T)$. In order to extract $n.\mathtt{spectrum}(T)$,* SiPTA *needs to normalize P_n. As DC*[1] *value P_n has different information than other parts of it,* SiPTA *normalizes P_n regarding non-DC frequencies. This normalization, is done by multiplying P_n by a constant that results the area under the spectrum to be 1:*

$$S_n = P_n \times \frac{\text{length}(n.\mathtt{timeSer}(T)) - 1}{\sum_{i=1}^{\text{length}(n.\mathtt{timeSer}(T))-1} P_n(i)}$$

Also, every member of above expression has a normalized frequency *associated with. For every index $0 \leq i < \|n.\mathtt{timeSer}(T)\|$, The* normalized frequency *of i^{th} member of the above expression is $\frac{i}{\|n.\mathtt{timeSer}(T)\|}$.*

The method $n.\mathtt{spectrum}(T)(f)$ takes $0 \leq f \leq \frac{1}{2}$ as normalized frequency *and outputs the corresponding spectrum value. It also uses linear interpolation for other frequencies*

[1]DC value is the value of frequency series, periodogram, and spectrum at frequency 0.

other than the ones associated with the samples of P_n. If $i = f \times \text{length}(n.\texttt{timeSer}(T))$, then $n.\texttt{spectrum}(T)(f)$ will be defined as follows.

$$n.\texttt{spectrum}(T)(f) = \begin{cases} S_n(i) & , i \in \mathbb{Z} \\ S_n(\lceil i \rceil) \times (i - \lfloor i \rfloor) + S_n(\lfloor i \rfloor) \times (\lceil i \rceil - i) & , \textit{otherwise} \end{cases}$$

With the definitions of time series and spectra for channel name classes, we can now look into different features which classifier uses.

3.3 Feature Extraction

SiPTA uses multiple metrics each of which gives a unique feature. The idea behind different metrics is to measure different properties of signals that in turn represent different periodic or non-periodic features of the channel name objects.

As is the case with the other parameters, metrics denote members of channel name objects. Let N be the set of channels over trace set $\mathbb{T}$. Also, let M represent a generic metric, which takes some parameters as input and outputs the metric, i.e., $n.M(T, \texttt{Params})$. The first parameter is the trace on which M is performing. The generic format of M includes `Params`. Some metrics might require additional parameters.

- **Channel Existence** represents existence of channel names in a trace. This metric has a boolean evaluation. In the greater context of unprocessed traces this metric seems if some $\rho \in \mathbb{P}$ has occurred in the trace or not.
- **Channel Length** represents the time series length of existing channels in a particular channel name object and a given trace. This metric is an integer metric. In the greater context of unprocessed traces, this metric sees how many times some $\rho \in \mathbb{P}$ has occurred in the trace
- **DC Significance** measures the significance of DC value regarding other values in the spectrum of a particular channel name of a particular trace. Also, an integer number represents this metric. This metric measures how regularly some $\rho \in \mathbb{P}$ occurs in the trace.

- **Multi-Peaks** measures the frequency and magnitude of significant values inside a the spectrum of some channel name of some particular trace. Depending on how many peaks this metric extracts, the metric can be an array of different lengths of pairs of frequency and magnitude. Each of these pairs represents a peak in the spectrum.

We define metrics as members of channel name objects, i.e., $n.M(T)$ is the metric M on channel name n. This metric takes T as input and returns the feature value of channel n of trace T using metric M. Also, the modular design of SiPTA facilitates the integration of additional metrics, thus supporting classification with additional features.

Let $\mathcal{M}$ be the set of m metrics. In other words, $\mathcal{M} = \{M_0, M_1, ..., M_{m-1}\}$. Figure 3.2 gives an overview of how feature extraction works. Every metric $M \in \mathcal{M}$ takes channel name object set and configurations as input and extracts the features using that metric for every channel name and every trace. Then M returns the calculated features that are passed as input to *feature integration* block. The feature integration block then integrates the extracted features with the input channel name objects.

Now that we have the overview of feature extraction, we can delve into each of the proposed metrics in SiPTA. In these sections we will examine the feature extraction metric. Then we will give a conceptual example to get a grasp on how that metric works. And after that we will give a real example of how that metric is useful in one of our datasets. The source for real examples is roughly explained in section 3.3.1.

3.3.1 Real Example

One of the trace sets that we produced for the purpose of this research was using BeagleBone and BeagleBoneBlack boards running QNX Neutrino Real-Time Operating System. There are twelve similar scenarios, each of which has over 400 different traces.

These scenarios produce and send random data to some ip address. The difference between these scenarios are the time intervals and/or some differences in random data generation functions.

For the following sections, we used these traces as a real examples to give the reader a better grasp on how feature extraction works in real examples.

3.3.2 Channel Existence

In this section we will examine the feature extraction method: *Channel Existence.*

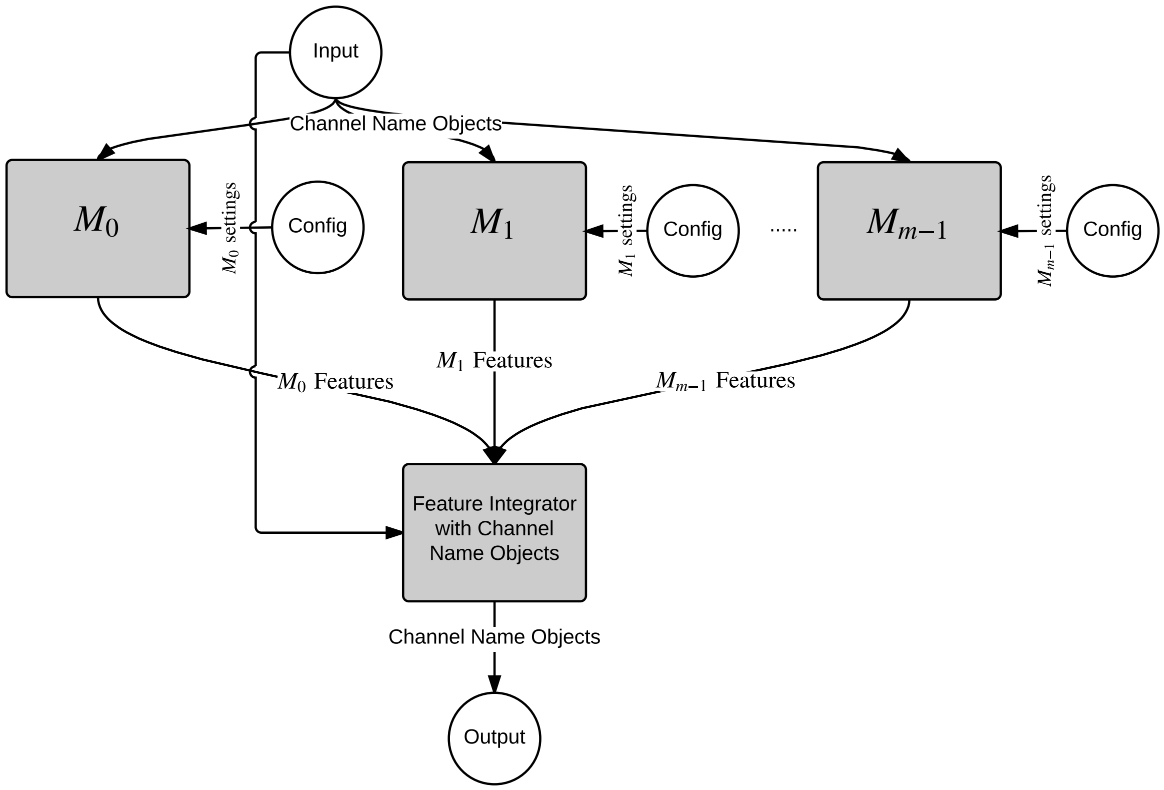


Figure 3.2: Feature Extraction overview

Definition 15 (Channel Existence) *Let N be the set of channel name objects over trace set $\mathbb{T}$. $\forall T \in \mathbb{T}, \forall n \in N$, $n.\texttt{exists}(T)$ is the Channel Existence metric*

Example 6 (Channel Existence Conceptual Example) *Let $\mathbb{T} = \{T_1, T_2\}$, be a set of traces of a read/write file access. Let T_1 be as Table 3.1 and T_2 be as trace 3.2. So, we can say that $N = \{\texttt{open::r}, \texttt{open::w}, \texttt{read::}, \texttt{write::}, \texttt{close::}\}$ is the set of channel names. In this case if $n = \texttt{open::w}$ or $n = \texttt{write::}$, then $n.\texttt{exists}(T_1) = \texttt{true}$ and $n.\texttt{exists}(T_2) = \texttt{false}$. Otherwise, $n.\texttt{exists}(T_1) = n.\texttt{exists}(T_2) = \texttt{true}$.*

Example 7 (Channel Existence Real Example) *Using the QNX Neutrino data described in Section 3.3.1, Table 3.3 will give the number of common channels names in the same scenario and between different scenarios. As you can see in Table 3.3, different scenarios tend to have more difference in the channel names than other traces from the same*

Index	*Time (μS)*	*File Access Action*	*Read/Write*
1	0	`open`	r
2	3	`open`	r
3	5	`read`	
4	9	`close`	
5	20	`read`	
6	21	`close`	
7	23	`open`	w
8	38	`write`	
9	38	`close`	

Table 3.1: Read/Write file access example trace T_1

Index	*Time (μS)*	*File Access Action*	*Read/Write*
1	0	`open`	r
2	3	`open`	r
3	5	`read`	
4	9	`close`	
5	20	`read`	
6	21	`close`	

Table 3.2: Read/Write file access example trace T_2

scenario. Specially, scenario 6 *and* 10 *with the greatest difference as compared to other scenarios.*

3.3.3 Channel Length

In this section, we will examine the feature extraction method: *Channel Length*.

Definition 16 (Channel Length) *Let N be the set of channels over trace set $\mathbb{T}$. For some $n_0 \in N$ and $T \in \mathbb{T}$ if $n_0.\texttt{exists}(T) = \texttt{true}$, we define $n_0.\texttt{length}(T)$ as Channel Length metric. In this case, $n_0.\texttt{length}(T) = len(n_0.\texttt{timeSer}(T)) / \sum_{\substack{n \in N \\ n.exists(T)=true}} \text{length}(n.\texttt{timeSer}(T))$. If $n.\texttt{exists}(T) = \texttt{false}$, $n_0.\texttt{length}(T)$ returns a value representing that it is not a number, meaning a value of* `NaN` *that stands for* not a number.

Scenario Number												
1	3											
2	4	0										
3	4	5	4									
4	4	3	4	4								
5	9	8	10	8	5							
6	43	41	44	44	47	2						
7	5	3	6	4	9	44	4					
8	5	5	6	4	10	44	6	4				
9	9	10	9	9	5	49	9	11	4			
10	45	41	46	44	49	8	44	46	47	6		
11	5	3	5	6	9	42	6	6	9	44	3	
12	4	2	5	5	8	41	5	3	10	43	3	0
Scenario Number	1	2	3	4	5	6	7	8	9	10	11	12

Table 3.3: Number of different channel names between scenarios

Example 8 (Channel Length Conceptual Example) *Let $\mathbb{T} = \{T_1, T_2\}$ as defined in Table 3.1 and Table 3.2. In this case, Table 3.4 gives the values for channel length.*

$n \in N$	$n.\texttt{length}(T_1)$	$n.\texttt{length}(T_2)$
`open::r`	1/4	1/3
`open::w`	0	`NaN`
`read::`	1/4	1/3
`write::`	0	`NaN`
`close::`	1/2	1/3

Table 3.4: Channel Length example

Example 9 (Channel Length Real Example) *Using the QNX Neutrino data described in Section 3.3.1, Table 3.5 provides the Channel Length of some channels on twelve different scenarios. As you can see the Channel Lengths of same scenarios tend to have more similar values than when compared to other scenarios.*

Scenario	$n.\texttt{length}(T)$ $(\times 10^{-5})$					
Number	n_1[2]		n_2[3]		n_3[4]	
1	3.55	3.3	4.33	3.76	354	619
2	2.57	2.73	2.91	3.23	834	463
3	2.85	3.62	3.36	4.26	462	504
4	3.85	3.3	4.54	3.76	615	619
5	3.03	3.07	3.58	3.5	369	573
6	3.74	3.72	4.57	4.55	1.41	1.41
7	6.65	6.68	7.83	7.87	537	540
8	4.4	4.3	5.19	5.07	488	476
9	4.68	4.65	5.51	5.47	797	792
10	5.81	5.23	7.28	6.59	3.72	3.57
11	7.89	7.89	9.19	9.2	620	618
12	5.94	0	7	6.99	413	412

Table 3.5: Channel Length example

3.3.4 DC Significance

In this section we will examine the feature extraction method: *DC Significance*.

Definition 17 (DC Significance) *This metric determines the significance of DC value for some spectrum. Let $\mathbb{T}$ be defined over set of channel names N, and let $T \in \mathbb{T}$. DC value is represented by the value of $n.\texttt{spectrum}(T)$ at frequency 0, i.e., $n.\texttt{spectrum}(T)(0)$. We define $n.\texttt{dcSig}(T)$ as DC Significance metric. for n and T, if $n.\texttt{exists}(()T) = \texttt{false}$ or $n.\texttt{length}(T) < 2$, then $n.\texttt{dcSig}(T) = \texttt{NaN}$. If $n.\texttt{exists}(()T) = \texttt{true}$ and $n.\texttt{length}(T) \geq 2$, we define this value as Equation 3.1 where $N = \lfloor \frac{n.\texttt{length}(T)}{2} \rfloor$.*

$$n.\texttt{dcSig}(T) = \frac{n.\texttt{spectrum}(T)(0)}{n.\texttt{length}(T) \times \frac{\sum_{i=1}^{N} n.\texttt{spectrum}(T)(i/n.\texttt{length}(T))}{N}} \tag{3.1}$$

In Equation 3.1 the numerator is the DC value of the spectrum. In the denominator, there are two parameters. The first parameter, $n.\texttt{length}(T)$, is for normalizing the DC

[2] `COMM,MSG_ERROR,`

[3] `COMM,MSG_ERROR,bin/sh`

[4] `COMM,REC_MESSAGE,proc/boot/pipe`

value according to the length of the channel. The reason for this normalization is that the FFT calculates DC by adding all the values in the time series. So, the DC value of spectrum depends on the length of the time series.

The second parameter, $\frac{\sum_{i=1}^{N} n.\texttt{spectrum}(T)(i/n.\texttt{length}(T))}{N}$, divides the DC value by the mean value of other frequency parameters in the spectrum. For example, if other values have a more significant value, the DC value in comparison to them would relatively decrease. This normalization puts it into account.

DC Significance measures how regular some event corresponding the channel name occurs in the trace. For example, if some event regularly occurs with a random interval between $9\mu s$ and $11\mu s$, it would have a higher DC Significance compared to some other event that occurs at random intervals between $5\mu s$ and $15\mu s$. Example 10 gives an conceptual example to clarify this metric.

Example 10 (DC Significance Conceptual Example) *Assume that trace T has two channel names: n_1 and n_2. Let n_1 represent an event that occur with intervals randomly between $9\mu S$ and $11\mu S$ with uniform distribution. Also, let n_2 represent an event that occur with intervals randomly between $5\mu S$ and $15\mu S$ with uniform distribution. Figure 3.3 depicts a small segment of event series of n_1 and n_2. Also, Figure 3.4 depicts their corresponding spectra. Table 3.6 depicts the DC Significance values of n_1 and n_2.*

Channel	*DC Significance value*
n_1	147.63
n_2	6.2033

Table 3.6: Example 10 DC Significance values

As depicted in Figure 3.3, n_1 is more regular than n_2. Also, as depicted in Figure 3.4, the DC values of $n_1.\texttt{spectrum}(T)$ and $n_2.\texttt{spectrum}(T)$ reside around 500. But the rest of the spectrum in n_1 is around 10^{-3} while the spectrum for n_2 is around 10^{-1} which is significantly higher than n_1. For this reason, $n_1.\texttt{dcSig}(T) > n_2.\texttt{dcSig}(T)$ which Table 3.6 confirms.

Example 11 (DC Significance Real Example) *Using the QNX Neutrino data described in Section 3.3.1, Table 3.7 gives the DC Significance of some channels of twelve different scenarios. For each scenario there is for different channel names and for each of those channel names, there are two different runs. As you can see the DC Significance values*

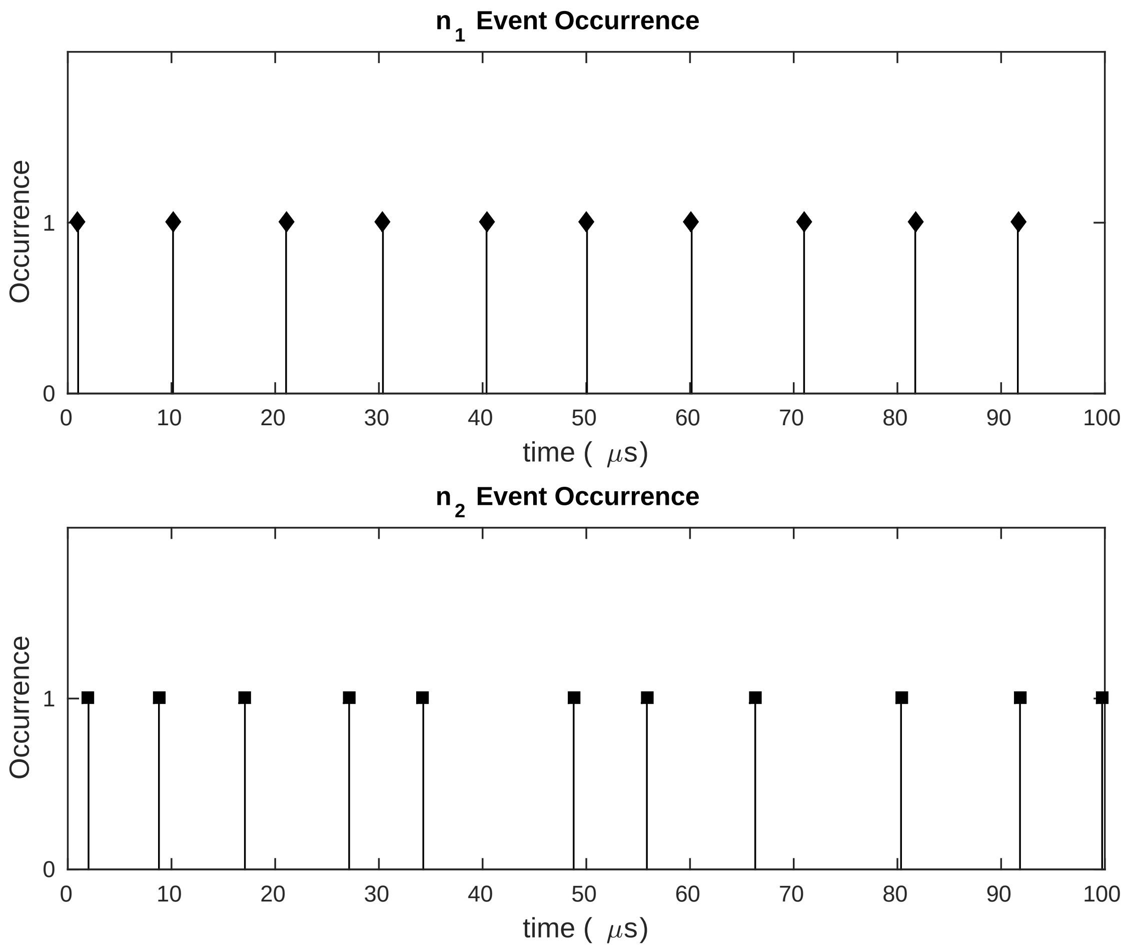
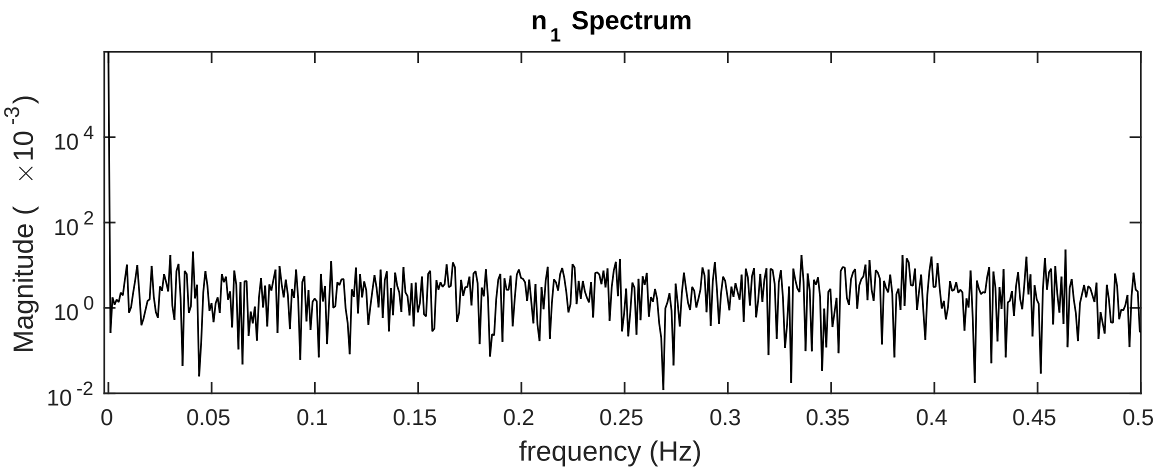


Figure 3.3: Example 10 events

of two different runs of the same scenarios and same channel names tends to have more similar values compared to other scenarios. This means that DC Significance can be good measure for system properties.



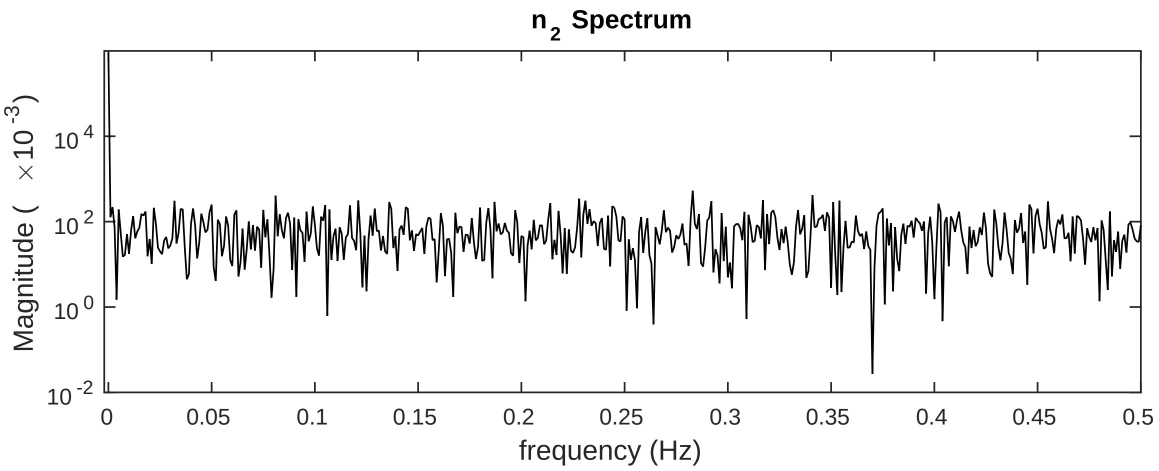


Figure 3.4: Example 10 spectra

3.3.5 Multi-Peak

In this section, we will review the functionality of *Multi-Peak* metric for feature extraction. For this reason, SiPTA uses the algorithm introduced in Section 2.2. As mentioned in Section 2.2, this method receives a time series and a confidence interval c as input and outputs a threshold for peak detection. SiPTA detects peaks by comparing spectrum to

[5] `COMM,MSG_ERROR,bin/sh`
[6] `COMM,REC_MESSAGE,proc/boot/pipe`
[7] `COMM,REC_MESSAGE,proc/boot/random`
[8] `COMM,REC_PULSE,proc/boot/io-pkt-v4-hc`

Scenario Number	$n.\texttt{dcSig}(T)$ $(\times 10^{-3})$							
	n_1[5]		n_2[6]		n_3[7]		n_3[8]	
	run 1	run 2	run 1	run 2	run 1	run 2	run 1	run 2
1	57.7	62.5	0.306	0.172	0.49	0.576	190	127
2	59.5	53.8	0.112	0.343	0.495	0.424	102	156
3	63.7	67.6	0.301	0.368	0.538	0.56	184	180
4	58.9	61.7	0.225	0.178	0.517	0.554	126	128
5	49	57.9	7.77	0.147	0.611	0.615	185	138
6	62.5	63.3	227	230	0.397	0.46	381	380
7	71.3	72.6	0.628	0.612	1.21	1.26	149	149
8	54.6	57.5	0.314	0.316	58.8	58.8	157	156
9	43	43	0.204	0.203	0.409	0.391	59.5	59.2
10	62.9	59.8	41.2	41.2	0.226	0.226	380	376
11	64.6	64.6	0.855	0.853	0.911	0.911	99	98.2
12	60.2	61	0.785	0.784	81.4	81.4	127	126

Table 3.7: DC Significance example

the derived threshold. For the sake of simplicity, we define the threshold finder algorithm in Section 2.2 as a member of channel name object. Definition 18 gives a formal definition for this member of channel name objects.

Definition 18 (Threshold Finder) *Let trace set $\mathbb{T}$ be defined over channel name set N. for any $n \in N$, $T \in \mathbb{T}$, and confidence interval $0 < c < 1$, let $n.\textit{\textbf{FindThreshold}}(T, c)$ return the peak detection threshold of channel n in trace T with confidence interval of c.*

In order to formally define Multi-Peak, we should first define peak.

Definition 19 (Peak) *Let $y : \mathbb{R} \to \mathbb{R}$ and $y_{\text{threshold}} : \mathbb{R} \to \mathbb{R}$ be some functions with domain $(x_{\min}, x_{\max})$. In this case, $p = \langle x_p, y_p \rangle \in (x_{\min}, x_{\max}) \times \mathbb{R}$ is a peak in y with regard to threshold $y_{\text{threshold}}$, if the following holds:*

- $y_p = y(x_p)$
- $y_p > y_{\text{threshold}}(x_p)$
- $\exists x_m, x_M \in (x_{\min}, x_{\max}), x_m < x_p < x_M, \forall x_m < x_0 < x_M, x_0 \neq x_p \Rightarrow y(x_0) < x_p$

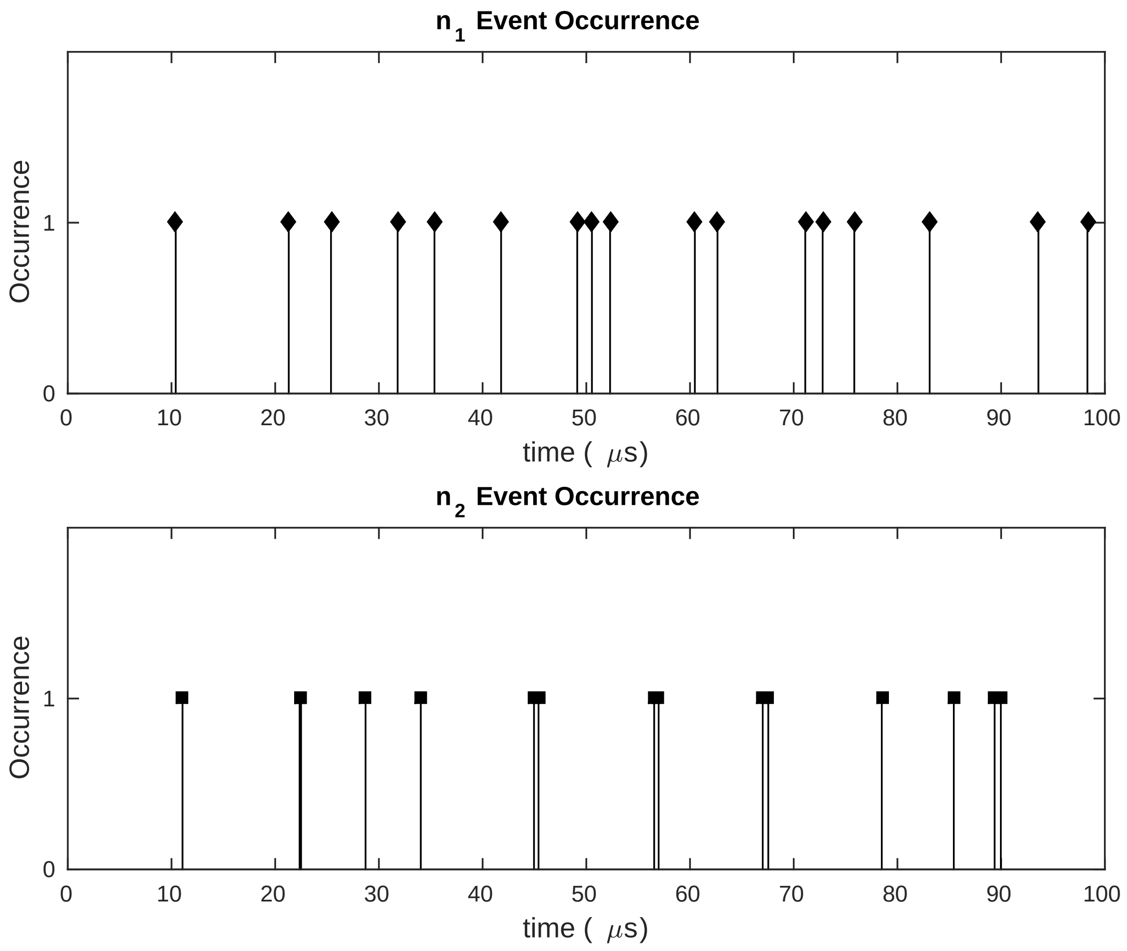


Figure 3.5: Example 12 events

In other words, p represents the x and y values for local maximums which exceed the threshold value.

With this definition of peaks, we can define the Multi-Peak metric.

Definition 20 (Multi-Peak) *Let $\mathbb{T}$ be defined over set of channel names N. For any $n \in N$, $T \in \mathbb{T}$, and $0 < c < 1$, if* $\text{length}(n.\texttt{timeSer}(T)) \geq \texttt{min-len-for-multi-peak}$, *$n.\texttt{MultiPeak}(T, c) = \texttt{NaN}$ where $\texttt{min-len-for-multi-peak}$ is the minimum time series length required for Multi-Peak metric. Otherwise, $n.\texttt{MultiPeak}(T, c)$ is the set of all peaks (Definition 19) of $n.\texttt{spectrum}(T)$ with regard to threshold $n.\texttt{FindThreshold}(T, c)$.*

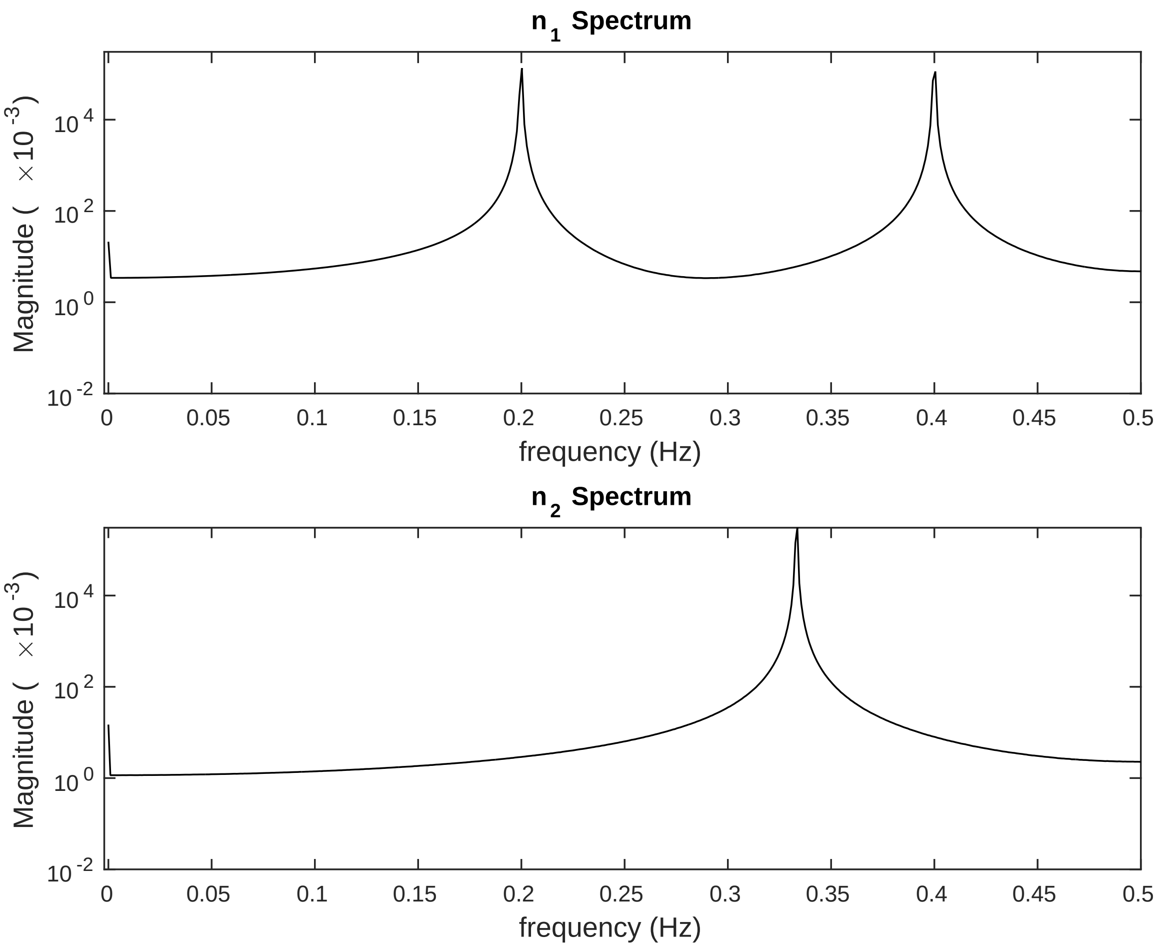


Figure 3.6: Example 12 spectra

Multi-Peak aims to detect multiple frequency components in a spectrum of different channels. These frequency components estimate the repetition pattern of corresponding event intervals. For example, if there is a major component in the repetition pattern in event intervals of length 5, then there would be major frequencies of $k/5$ ($k = 1, 2, ...$) in the spectrum. Of course this metric will also detect features that do not comply with this type of example, but they tend to remain the same in same scenarios.

Example 12 (Multi-Peak Conceptual Example) *Assume that trace T has two channel names: n_1 and n_2. Let n_1 be some channel that its corresponding events has a repetition pattern of length 5 and n_2 be some channel that its corresponding events has a repetition*

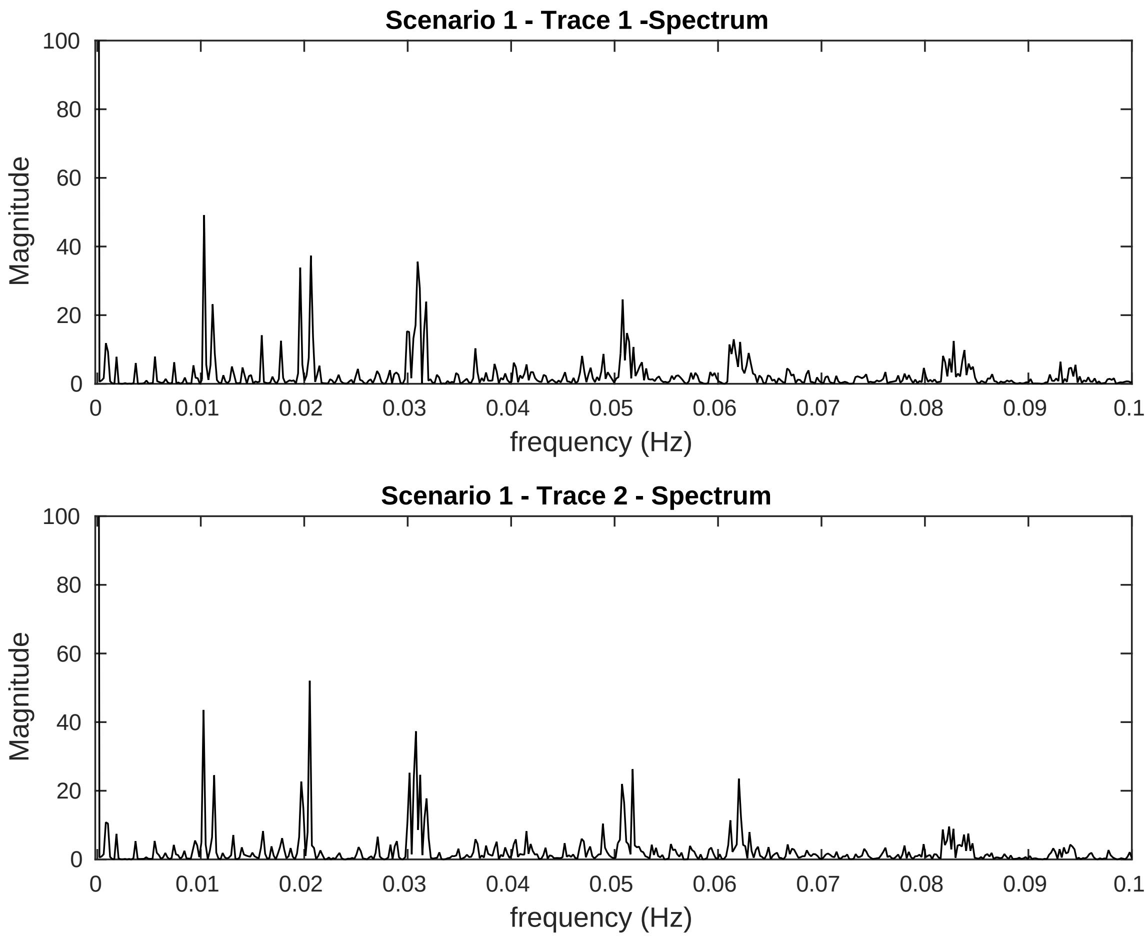


Figure 3.7: $n.\texttt{spectrum}(T_1)$ and $n.\texttt{spectrum}(T'_1)$ in Example 13

pattern of length 3*. Some noise is added to the repetition patterns of both n_1 and n_2. Figure 3.5 depicts a small segment of event series of n_1 and n_2. Also, Figure 3.6 depicts their corresponding spectra.*

As depicted in Figure 3.3, the repetition pattern of n_1 has the length of 5 and the repetition pattern of n_2 has a length of 3. It is evident that major peaks of $n_1.\texttt{spectrum}(T)$ and $n_2.\texttt{spectrum}(T)$ are at frequencies $\frac{k}{5}$ $(k = 1, 2)$ and $\frac{k}{3}$ $(k = 1)$, respectively. These frequencies align with the repetition pattern periods that are the source of event construction for these channel names.

Now, we can take a look at Example 13.

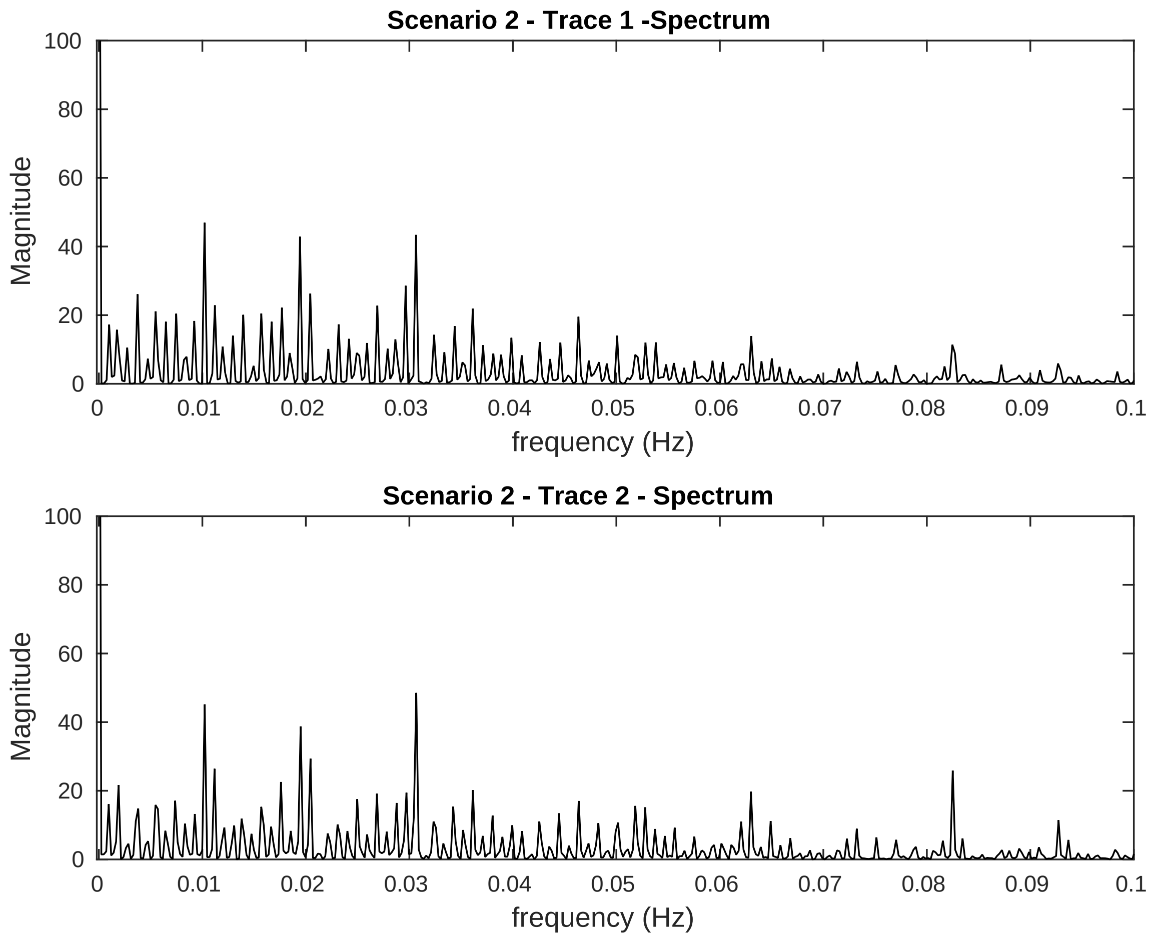
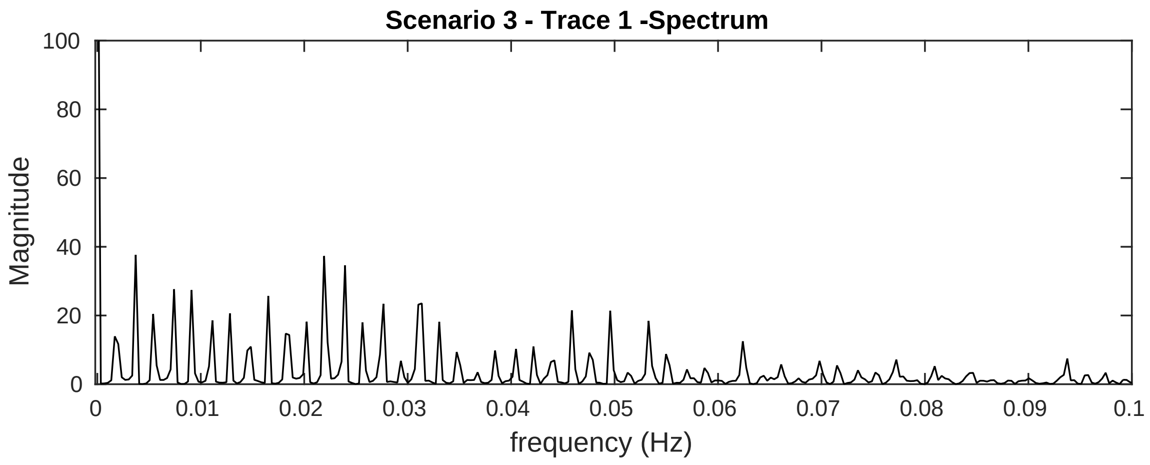


Figure 3.8: $n.\texttt{spectrum}(T_2)$ and $n.\texttt{spectrum}(T_2')$ in Example 13

Example 13 (Multi-Peak Real Example) *Let $\mathbb{T}$, the trace set explained in Section 3.3.1 be over channel name set N. Also, let $n \in N$[9] and for any $T_i \in \mathbb{T}$ represent a scenario according to subscript i. For example, T_1 and T_1' have same scenarios while T_1 and T_2 have different scenarios. Figures 3.7, 3.8, 3.9 and 3.10 depict $n.\texttt{spectrum}(T)$ of four different scenarios, each of which possess two different traces.*

As you can see in the figures provided in Example 13, spectra of different scenarios tend to have different Multi-Peak values. For example scenario 1 has fewer peaks compared to other scenarios. Likewise, scenario 4 has higher peaks at frequencies around $k \times 5 \times 10^{-3}$

[9] $n =$ `COMM,SND_PULSE_EXE,proc/boot/devb-mmcsd-jacinto5`



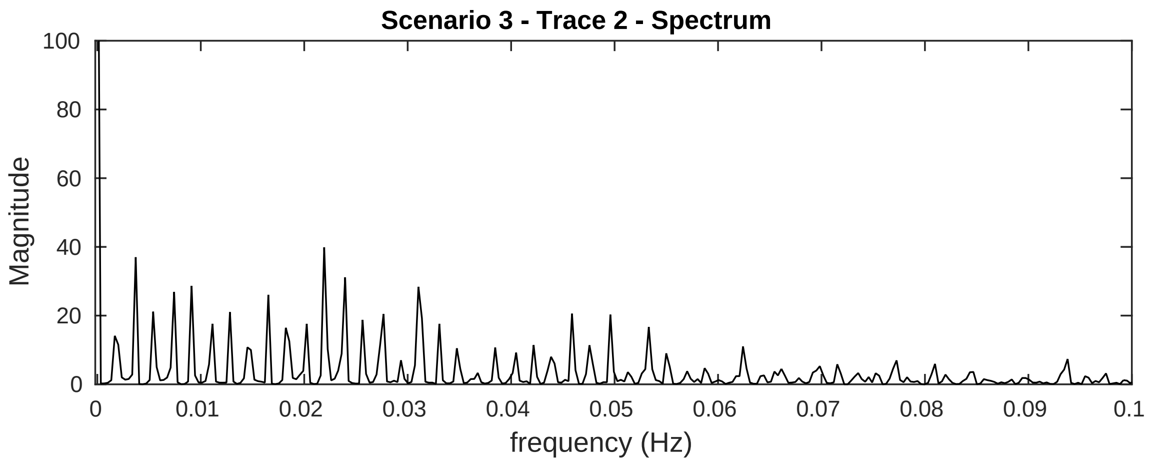
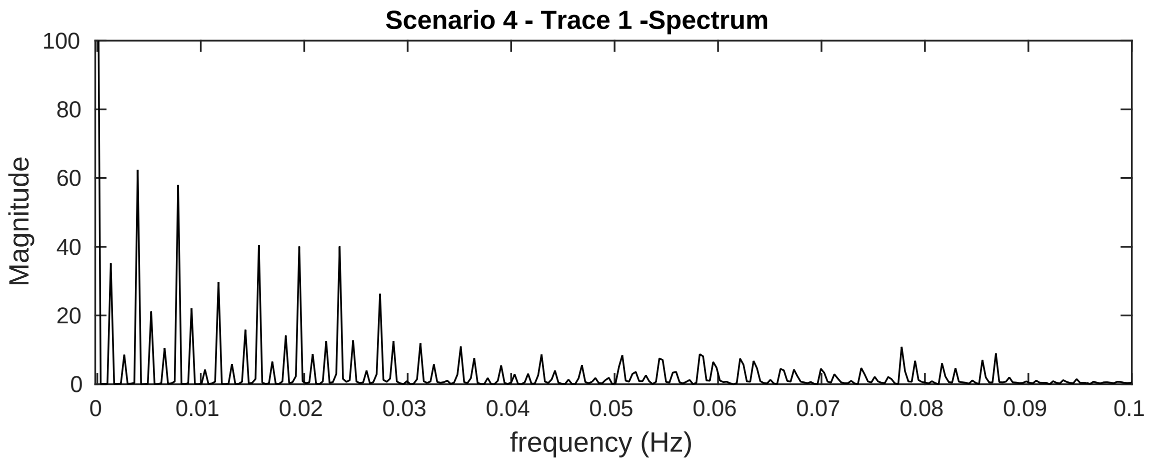


Figure 3.9: $n.\texttt{spectrum}(T_3)$ and $n.\texttt{spectrum}(T_3')$ in Example 13

while scenarios 1 and 2 have significant peaks at frequencies $k \times 1 \times 10^{-2}$ and scenario 3 has peaks at much smaller frequency intervals.



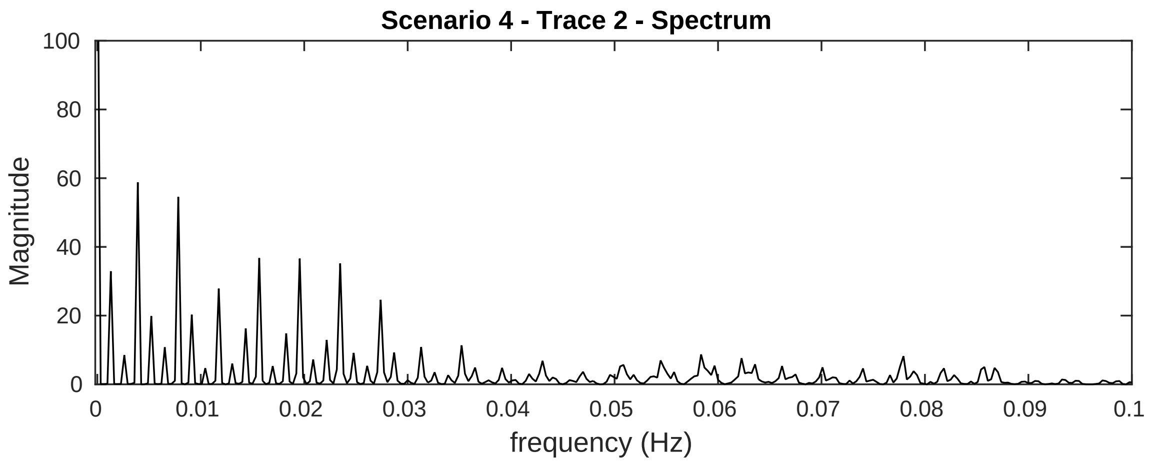


Figure 3.10: $n.\texttt{spectrum}(T_4)$ and $n.\texttt{spectrum}(T_4')$ in Example 13

3.4 Training

The third part of SiPTA that we are going to examine is the *Train Engine* depicted in Figure 3.1. This block uses the features of train traces to extract *Normal Feature Constraints*.

We define normal feature constraints as a member of channel name objects: $n.t$ is a trainer t for channel name n.

Let $\mathcal{M}$ be the set of m metrics. In other words, $\mathcal{M} = \{M_0, M_1, ..., M_{m-1}\}$. Similarly, let $\mathcal{T}$ be a set of m training blocks or trainers, i.e., $\mathcal{T} = \{t_0, t_1, ..., t_{m-1}\}$.

Figure 3.11 gives an overview of how training works. Let $\mathbb{T} = \mathbb{T}^t \cup \mathbb{T}^a$ where $\mathbb{T}^t$ is the set of training traces and $\mathbb{T}^a$ is the set of analyzing traces. Also, let $\mathbb{T}$ be over the channel name set N. Every trainer $t \in \mathcal{T}$ takes the set of channel name objects, configurations, and $\mathbb{T}^t$ as input and extracts the normal constraints of the corresponding feature for all applicable channel names under the assumption that $\mathbb{T}^t$ represent normal behavior. Then, all the normal constraints are integrated with channel name objects.

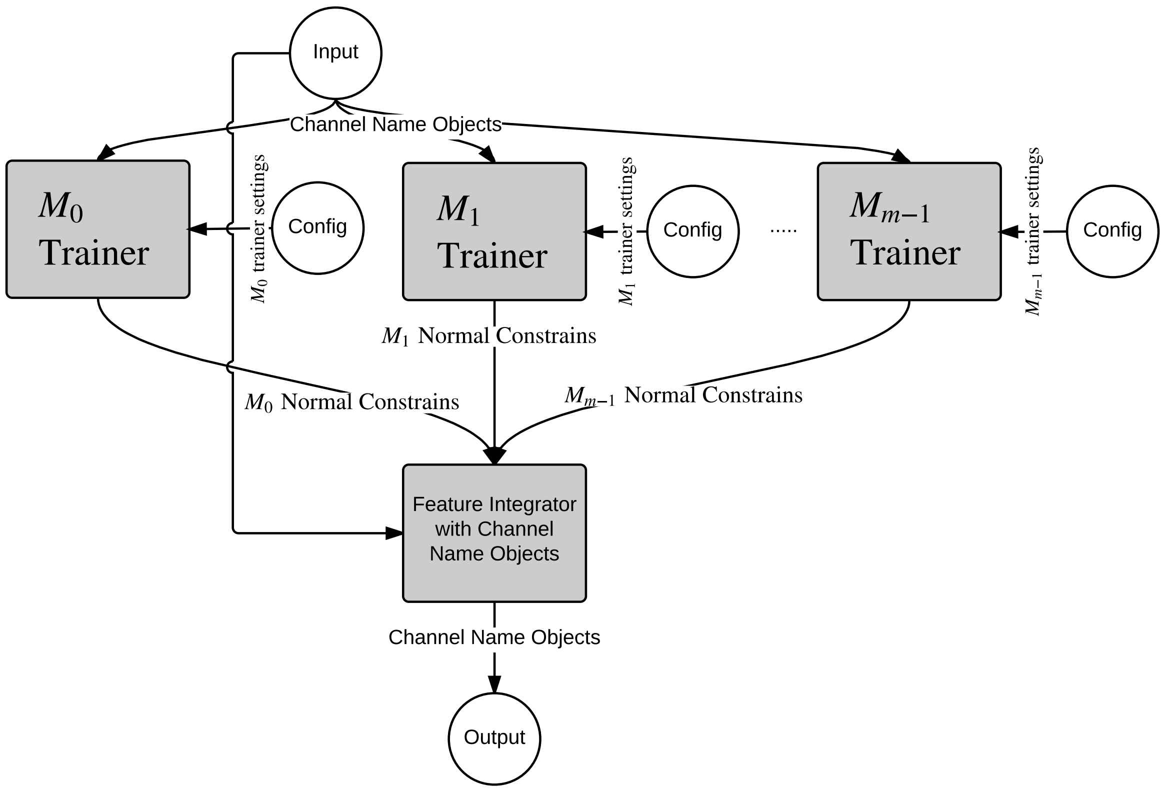


Figure 3.11: Train engine overview

Given this introduction, we can now look into different trainers and normal constrains. In the following sections we assume that $\mathbb{T}$ is the set of traces that can be partitioned into $\mathbb{T}^t$ and $\mathbb{T}^a$, which are training and analyzing sets, respectively. Also, we assume that $\mathbb{T}$ is defined over the channel name set N.

3.4.1 Channel Existence

We define the Channel Existence constraint as `normally-exists`. n.`normally-exists` should return the normal constraint of Channel Existence for channel name n if $n \in N$. Channel Existence trainer partitions N into three groups: `all`, `some`, `none`. These groups are defined as below:

- **`all`**: For any $n \in N$: if $\forall T \in \mathbb{T}^t, n.\texttt{exists}(T) = \texttt{true}$ then $n.\texttt{normally-exists} = \texttt{all}$
- **`some`**: For any $n \in N$: if $\exists T \in \mathbb{T}^t, n.\texttt{exists}(T) = \texttt{true} \wedge \exists T \in \mathbb{T}^t, n.\texttt{exists}(T) = \texttt{false}$ then $n.\texttt{normally-exists} = \texttt{some}$
- **`none`**: For any $n \in N$: if $\forall T \in \mathbb{T}^t, n.\texttt{exists}(T) = \texttt{false}$ then $n.\texttt{normally-exists} = \texttt{none}$

3.4.2 Channel Length

For every channel name, normal constraints of Channel Length is average and variance of Channel Lengths which are represented by `length-avg` and `length-var`, respectively. Definitions 21 and 22 formally define `length-avg` and `length-var`.

Definition 21 (Average Channel Length) *For any $n \in N$: if $\exists T_1, T_2 \in \mathbb{T}^t, T_1 \neq T_2 \wedge n.\texttt{length}(T_1) \neq \texttt{NaN} \wedge n.\texttt{length}(T_2) \neq \texttt{NaN}$, we can calculate the* average Channel Length*. Otherwise $n.\texttt{length-avg} = \texttt{NaN}$. In the case that that $n.\texttt{length-avg}$ is calculable, Equation 3.2 gives a formal definition of average Channel Length.*

$$\forall n \in N, n.\texttt{length-avg} = \operatorname*{Avg}_{\substack{T\in\mathbb{T}^t\wedge \\ n.\texttt{length}(T)\neq\texttt{NaN}}} (n.\texttt{length}(T)) \tag{3.2}$$

Definition 22 (Channel Length Variance) *For any $n \in N$, if $\exists T_1, T_2 \in \mathbb{T}^t, T_1 \neq T_2 \wedge n.\texttt{length}(T_1) \neq \texttt{NaN} \wedge n.\texttt{length}(T_2) \neq \texttt{NaN}$ we can calculate* Channel Length variance*. Otherwise $n.\texttt{length-var} = \texttt{NaN}$. In the case that $n.\texttt{length-var}$ is calculable, Equation 3.3 gives a formal definition for Channel Length variance.*

$$\forall n \in N, n.\texttt{length-var} = \operatorname*{Var}_{\substack{T\in\mathbb{T}^t\wedge \\ n.\texttt{length}(T)\neq\texttt{NaN}}} (n.\texttt{length}(T)) \tag{3.3}$$

3.4.3 DC Significance

For every channel name, normal constraints of DC Significance is the average and variance of DC Significances, which are represented by `dcSig-avg` and `dcSig-var`, respectively. Definitions 23 and 24 formally define `dcSig-avg` and `dcSig-var`.

Definition 23 (Average Channel Length) *For any $n \in N$: if $\exists T_1, T_2 \in \mathbb{T}^t, T_1 \neq T_2 \wedge n.\texttt{dcSig}(T_1) \neq \texttt{NaN} \wedge n.\texttt{dcSig}(T_2) \neq \texttt{NaN}$, we can calculate* average Channel Length*. Otherwise, $n.\texttt{dcSig-avg} = \texttt{NaN}$. In the case that $n.\texttt{dcSig-avg}$ is calculable, Equation 3.4 gives a formal definition of average Channel Length.*

$$\forall n \in N, n.\texttt{dcSig-avg} = \underset{\substack{T \in \mathbb{T}^t \wedge \\ n.\texttt{dcSig}(T) \neq \texttt{NaN}}}{\text{Avg}} (n.\texttt{dcSig}(T)) \tag{3.4}$$

Definition 24 (Channel Length Variance) *For any $n \in N$: if $\exists T_1, T_2 \in \mathbb{T}^t, T_1 \neq T_2 \wedge n.\texttt{dcSig}(T_1) \neq \texttt{NaN} \wedge n.\texttt{dcSig}(T_2) \neq \texttt{NaN}$, we can calculate* Channel Length variance*. Otherwise, $n.\texttt{dcSig-var} = \texttt{NaN}$. In the case that $n.\texttt{dcSig-var}$ is calculable, Equation 3.5 gives a formal definition of Channel Length variance.*

$$\forall n \in N, n.\texttt{dcSig-var} = \underset{\substack{T \in \mathbb{T}^t \wedge \\ n.\texttt{dcSig}(T) \neq \texttt{NaN}}}{\text{Var}} (n.\texttt{dcSig}(T)) \tag{3.5}$$

3.4.4 Multi-Peak

For every channel name, normal constraints of Multi-Peak are averages and variances/co-variances of peaks that are represented by `MultiPeak-avg` and `MultiPeak-cov`, respectively. `MultiPeak-avg` and `MultiPeak-cov` are sequences of averages and variances/co-variances of peak frequency and magnitude.

The Multi-Peak train engine algorithm does the following for some $0 < c < 1$ and every $n \in N$

1. Let $\mathbb{P}_n$ be a set of peaks of channel n, meaning $\mathbb{P}_n = \{P_0, P_1, ..., P_{m-1}\}$ where $m = |\mathbb{P}|$. Also, let members of $\mathbb{P}_n$ be a tuple of peak location (frequency and magnitude) and

a reference to trace that the peak came from. In this case, this step of Multi-Peak train engine sets the value $\mathbb{P}_n$ as equation 3.6.

$$\mathbb{P}_n = \{\langle p, T\rangle | \forall T \in \mathbb{T}, \forall p \in n.\texttt{MultiPeak}(T, c) \wedge n.\texttt{MultiPeak}(T, c) \neq \texttt{NaN}\} \tag{3.6}$$

2. A hierarchical cluster tree H (according to section 2.4) is formed regarding the frequency value of the peak member of every $P \in \mathbb{P}_n$. In other words, every $P \in \mathbb{P}_n$ form cluster tree with regard to $P.p.f$ (recall that $P = \langle (f, m), T\rangle$).

3. The lowest cluster count h is found such that none of the clusters in $H(h-1)$ have peaks with peaks of the same trace. Formally, Equation 3.7 describes this step.

$$h = min(h), \nexists \texttt{Clusts} \in H(h-1), \exists p, p' \in \texttt{Clust}, p \neq p' \wedge p.T = p'.T \tag{3.7}$$

4. Assign $\texttt{Clusts} = H(h-1)$, which is a set of h peak cluster.

5. Remove the clusters from `Clusts` with less that 2 peaks.

6. Assign $\texttt{ClustCnt} = |\texttt{Clusts}|$, which represents the number of peak clusters.

7. For every $0 \leq i \leq \texttt{ClustCnt} - 1$, let $C = \texttt{Clusts}(i)$. Let Equation 3.8 and 3.9 assign the values of $n.\texttt{MultiPeak-avg}$ and $n.\texttt{MultiPeak-cov}$.

$$n.\texttt{MultiPeak-avg}(i) = \langle \underset{P \in C}{\text{Avg}}(P.p.f), \underset{P \in C}{\text{Avg}}(P.p.m)\rangle \tag{3.8}$$

$$n.\texttt{MultiPeak-cov}(i) = \begin{cases} \underset{P \in C}{\text{Cov}}(P.p.f, P.p.m), & \text{if the result is invertible} \\ \langle \underset{P \in C}{\text{Var}}(P.p.f), \underset{P \in C}{\text{Var}}(P.p.m)\rangle, & \text{otherwise} \end{cases} \tag{3.9}$$

3.5 Analyzing

The fourth part of SiPTA that we are going to examine is *Analyze Engine* as illustrated in Figure 3.1. This block uses the features of train traces, analyze traces and normal feature constraints to extract *Labels for Analyze Traces.*

Let $\mathcal{M}$ be the set of m metrics. In other words, $\mathcal{M} = \{M_0, M_1, ..., M_{m-1}\}$.

Figure 3.12 provides an overview of how analyzing works. Let $\mathbb{T} = \mathbb{T}^t \cup \mathbb{T}^a$ where $\mathbb{T}^t$ is the set of training traces and $\mathbb{T}^a$ is the set of analyzing traces. Also, let $\mathbb{T}$ be over channel name set N.

As Figure 3.12 depicts, every feature extraction method has its corresponding analyzer. Each analyzer takes features, and normal feature constraints as input and outputs a score value. The block *label assigner* then takes these scores for training and analyzing traces and use these scores to score every analyzing trace. The analyzer blocks and the label assigner block has its set of configurations that are omitted in Figure 3.12 for the sake of simplicity.

As Section 3.4 showed, normal feature constraints are embedded in channel name objects. So, virtually all the inputs required for analyze engine is the set of channel name objects that are updated through train engine. The depiction of three separate inputs in Figure 3.12 is not necessary but allows for better understanding of the analyze engine.

In this section, we examine how different analyzer blocks produce the score value. Additionally, we examine how the label assigner block works.

In the following sections, we assume that $\mathbb{T}$ is the set of traces which has two partitions $\mathbb{T}^t$ and $\mathbb{T}^a$ which are training and analyzing sets respectively. Also, we assume that $\mathbb{T}$ is defined over channel name set N.

3.5.1 Channel Existence

For any $T \in \mathbb{T}$, the Channel Existence analyzer return a non-negative integer that represent how likely it is for T to be anomalous on the sole basis of Channel Existence. Equation 3.10 offers a formal formula for $\texttt{Score}_{\texttt{exists}}$:

$$\begin{aligned} \texttt{Score}_{\texttt{exists}}(T) = |\{n|(&n.\texttt{exists}(T) = \texttt{false} \wedge n.\texttt{normally-exists} = \texttt{all})\vee \\ &(n.\texttt{exists}(T) = \texttt{true} \wedge n.\texttt{normally-exists} = \texttt{none})\}| + 1 \end{aligned} \tag{3.10}$$

As Equation 3.10 explains, $\texttt{Score}_{\texttt{exists}}(T)$ counts the number of channel names in T that exist while they are absent from any of the training traces and the channel names that exists in all train traces while they do not exist in T.

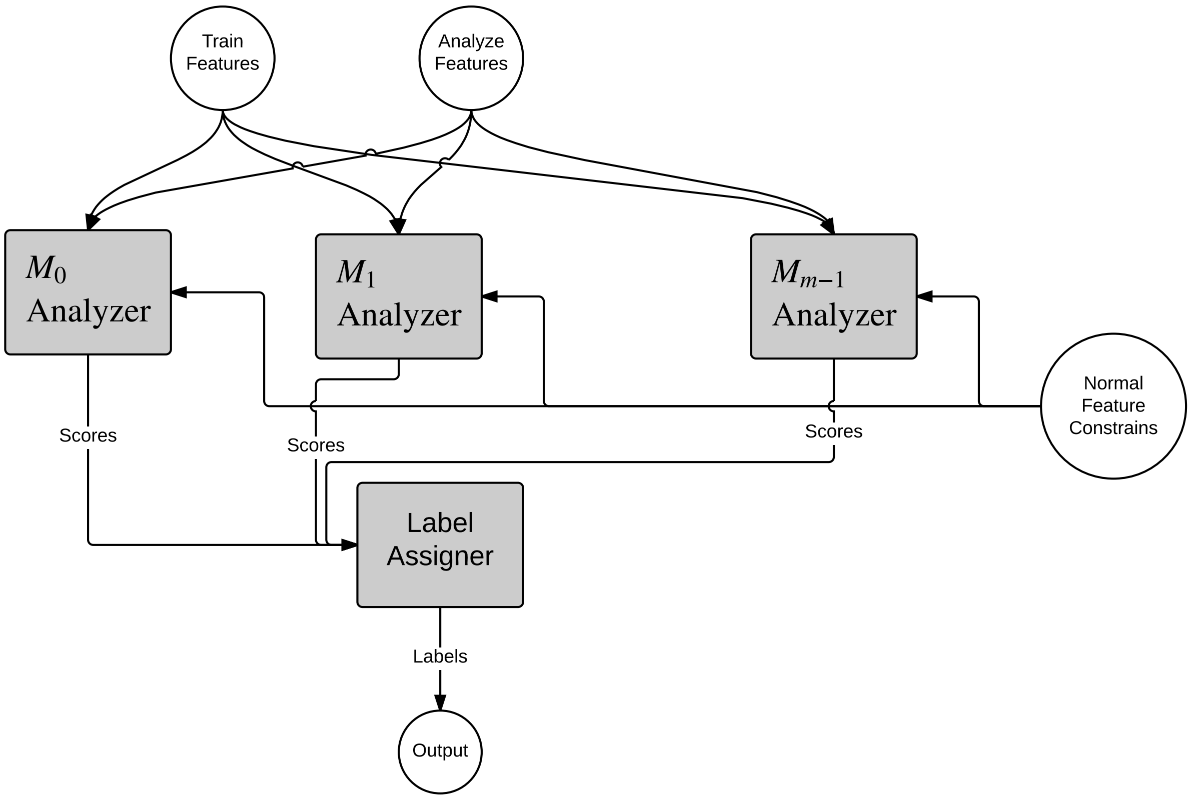


Figure 3.12: Analyze engine overview

3.5.2 Channel Length

For any $T \in \mathbb{T}$, the Channel Length analyzer return a non-negative real value that represents how likely it is for T to be anomalous on the sole basis of Channel Length. Equation 3.11 gives a formal formula for $\texttt{Score}_{\texttt{length}}$.

$$\texttt{Score}_{\texttt{length}}(T) = \underset{\substack{n \in N \wedge \\ n.\texttt{length}(T) \neq \texttt{NaN} \wedge \\ n.\texttt{length-avg} \neq \texttt{NaN}}}{\text{Avg}} \left(\frac{(n.\texttt{length}(T) - n.\texttt{length-avg})^2}{n.\texttt{length-var}} \right) \tag{3.11}$$

3.5.3 DC Significance

For any $T \in \mathbb{T}$, the DC Significance analyzer return a non-negative real value that represents how likely it is for T to be anomalous on the sole basis of DC Significance. The equation for calculating DC Significance score is very similar to Equation 3.11, which Equation 3.12 gives a formal formula for.

$$\texttt{Score}_{\texttt{dcSig}}(T) = \underset{\substack{n \in N \wedge \\ n.\texttt{dcSig}(T) \neq \texttt{NaN} \wedge \\ n.\texttt{dcSig-avg} \neq \texttt{NaN}}}{\text{Avg}} \left(\frac{(n.\texttt{dcSig}(T) - n.\texttt{dcSig-avg})^2}{n.\texttt{dcSig-var}} \right) \tag{3.12}$$

3.5.4 Multi-Peak

Compared to the other metrics we presented in this thesis, Multi-Peak is the most sophisticated metric in every aspect so far. This sophistication extends to feature extraction and training.

Assume that for some $T \in \mathbb{T}$, and $n \in N$, both $n.\texttt{MultiPeak}(T, c)$ and $n.\texttt{MultiPeak-avg}$ has at least one peak. Also, under the assumption that $p_t \in n.\texttt{MultiPeak}(T, c)$ and $p_{\text{Avg}} \in n.\texttt{MultiPeak-avg}$ and $p_{\text{Cov}} \in n.\texttt{MultiPeak-cov}$ are average and variance/covariance information of same peaks in Multi-Peak normal constraints. By matching these peaks together, the cost of p_t deviation from normal constraint ($\texttt{Score}_{\texttt{MultiPeak}}(T, n, p_t)$) is determined by Equation 3.13. In this equation, $S = \texttt{Score}_{\texttt{MultiPeak}}(T, n, p_t)$, $f = p_t.f$, $m = p_t.m$, $\bar{f} = p_{\text{Avg}}.f$, $\bar{m} = p_{\text{Avg}}.m$, and $\Sigma = p_{\text{Cov}}$. In the event that p_{Cov} is not a covariance matrix, $f_{\sigma^2} = p_{\text{Cov}}.m$ and $m_{\sigma^2} = p_{\text{Cov}}.m$.

$$S = \begin{cases} \begin{bmatrix} f - \bar{f} & m - \bar{m} \end{bmatrix} \times \Sigma^{-1} \times \begin{bmatrix} f - \bar{f} \\ m - \bar{m} \end{bmatrix}, & p_{\text{Cov}} \text{ is covariance} \\ \dfrac{(f - \bar{f})^2}{f_{\sigma^2}} + \dfrac{(m - \bar{m})^2}{m_{\sigma^2}}, & p_{\text{Cov}} \text{ is two variances} \end{cases} \tag{3.13}$$

Now, under the assumption that there are k_p peaks in $n.\texttt{MultiPeak}(T, c)$ and k_n peaks in $n.\texttt{MultiPeak-avg}$ and $n.\texttt{MultiPeak-cov}$. Let $k = \min(k_p, k_n)$ and $K = \max(k_p, k_n)$. We can match k peaks of $n.\texttt{MultiPeak}(T, c)$ to k peaks of $n.\texttt{MultiPeak-avg}$[10]. Let us name the results as $S_0, ..., S_{k-1}$. The challenge is that to match these peaks in such a way

[10]Remeber that the number of peaks in $n.\texttt{MultiPeak-cov}$ is equal to the number of peaks in $n.\texttt{MultiPeak-avg}$

that $\sum_{i=0}^{k-1} S_i$ minimizes. This matching is not feasible using brute-force algorithm because there are ${}^{K!}/_{(K-k)!}$ different combinations.

In order to do so, we make a weighted complete bipartite graph $G = \{V, E, w\}$ (see Section 2.3) with $2K$ vertices. On the left side of this graph there are peaks from $n.\texttt{MultiPeak}(T, c)$ while on the right side there are peaks from $n.\texttt{MultiPeak-avg}$ and $n.\texttt{MultiPeak-cov}$. The remaining nodes on the side with k peaks will have $K - k$ dummy peaks. Equation 3.13 determines the weight of the edges if none of the vertices on either side of the corresponding edge is a dummy node; choosing the corresponding peaks from $n.\texttt{MultiPeak}(T, c)$ and $n.\texttt{MultiPeak-avg}$ and $n.\texttt{MultiPeak-cov}$ as inputs for this equation. If the edge is connected to a dummy vertex, then its weight is 0. Then Hungarian algorithm which is explained in Section 2.3 finds the best matching for peaks in $n.\texttt{MultiPeak}(T, c)$ with regard to $n.\texttt{MultiPeak-avg}$ and $n.\texttt{MultiPeak-cov}$ and outputs the minimum possible $\sum_{i=0}^{k-1} S_i$.

Let $\texttt{Score}_{\texttt{MultiPeak}}(T, n)$ be $\frac{(K-k+1)}{k} \times \sum_{i=0}^{k-1} S_i$ for minimum value of $\sum_{i=0}^{k-1} S_i$ if there is at least one peak in $n.\texttt{MultiPeak}(T, c)$, and $n.\texttt{MultiPeak-avg}$ and $n.\texttt{MultiPeak-cov}$. Otherwise, let $\texttt{Score}_{\texttt{MultiPeak}}(T, n) = \texttt{NaN}$. In this case, the Multi-Peak score for T is the average of all $\texttt{Score}_{\texttt{MultiPeak}}(T, n)$ with none-NaNvalues. Formally, Equation 3.14 describes $\texttt{Score}_{\texttt{MultiPeak}}(T)$.

$$\texttt{Score}_{\texttt{MultiPeak}}(T) = \underset{\substack{n \in N \wedge \\ \texttt{Score}_{\texttt{MultiPeak}}(T,n) \\ \neq \texttt{NaN}}}{\text{Avg}} \left(\texttt{Score}_{\texttt{MultiPeak}}(T, n)\right) \tag{3.14}$$

3.5.5 Label Assigner

The job of the *label assigner* is to use the scores from $\mathbb{T}^t$ and $\mathbb{T}^a$ to assign labels to traces in $\mathbb{T}^a$. Consider the models in Figures 3.2, 3.11, and 3.12. In these models there are m feature extraction methods. Let $\mathbb{T}^t = \{T_0, ..., T_{n-1}\}$ and assume $n > m$. In this case, let the S_t be the score observation matrix for $\mathbb{T}^t$. Equation 3.15 formally defines S_t.

$$S_t = \begin{bmatrix} \texttt{Score}_{M_0}(T_0) & \cdots & \texttt{Score}_{M_0}(T_{n-1}) \\ \vdots & \ddots & \vdots \\ \texttt{Score}_{M_{m-1}}(T_0) & \cdots & \texttt{Score}_{M_{m-1}}(T_{n-1}) \end{bmatrix} \tag{3.15}$$

In the current version of SiPTA $m = 4$, $M_0 = \texttt{exists}$, $M_1 = \texttt{length}$, $M_2 = \texttt{dcSig}$ and $M_3 = \texttt{MultiPeak}$. Also, let $\Sigma_t = \text{Cov}(S_t)$. As $n > m$, except for extremely specific cases, Σ_t is going to be invertible.

For any $T \in \mathbb{T}$, let S_T be a column vector of scores from different methods of SiPTA and let $S(T)$ be the overall score for trace T. Equation 3.16 calculates $S(T)$ using Mahalanobis distance [34] formulation.

$$S(T) = S'_T \times \Sigma_t^{-1} \times S_T \tag{3.16}$$

Also, let $S_{\max} = \max\{S(T) | T \in \mathbb{T}^t\}$. In this case, for any $T \in \mathbb{T}$, let $s(T) = {}^{S(T)}/_{S_{\max}}$ be the normalized score for T.

Figure 3.13 gives a real example of the values of scores in some test case. In this figure, $\mathbb{T}^t = \{T_1, ..., T_5\}$ and $\mathbb{T}^a = \{T_6, ..., T_{15}\}$.

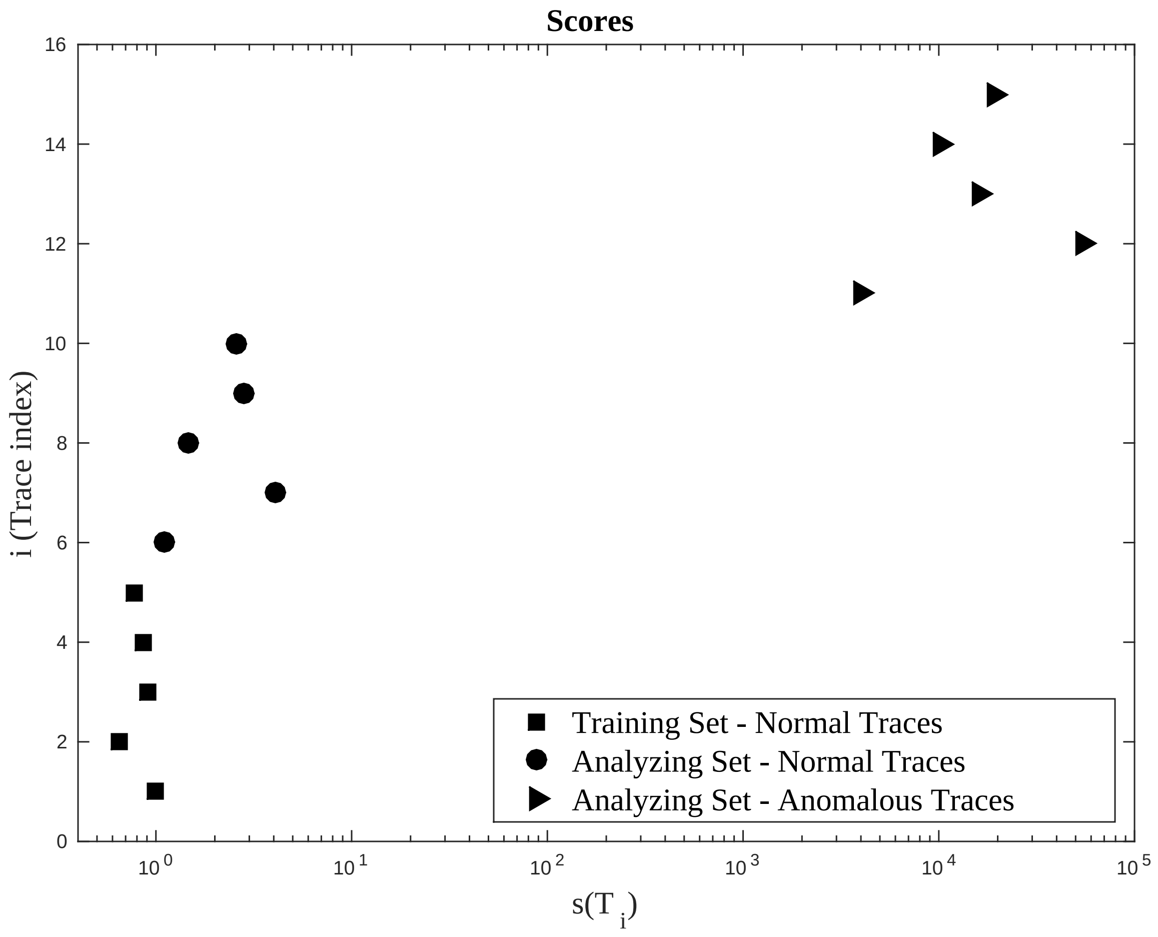


Figure 3.13: Scores

As Figure 3.13 depicts, the score values for dirty traces are much higher than clean traces. If a threshold values r between 8 and 2000 distinct clean and dirty traces, SiPTA can achieve perfect classification.

Chapter 4

Experimental Setup

While SiPTA in theory is well suited to the problem domain of periodic systems, we followed up our theory work with an experimental evaluation to provide quantitative results. The following section describes the conducted experiments used to evaluate the proposed technique, the evaluation process, and the results.

4.1 Experimental Setup and Workload

Our experiments use the QNX Neutrino 6.4 real-time operating system. While the framework is independent from system traces, we used QNX, because we have three distinct applications from which we can gather traces: a hexacopter application [8], a QNX CAR infotainment unit, and Blackberry phones running QNX. This allows us to gather system traces from very different execution contexts and properly evaluate SiPTA.

To collect data, we used the QNX kernel logging facility for our experiments. The kernel logging facility provides trace capabilities through QNX tracelogger. A trace collected from tracelogger contains a chronological order of system events such as system calls, message passes, interrupts, I/O etc. Every trace entry corresponds to a system event and a set of additional related information, such as the source/destination of a message pass. A sample trace is shown in the following snippet:

```
TIMESTAMP, CPU, EVENT,      PID, PROC, Details
t1,        1,   PROCCREATE, 1,   A,    PPID: 0 ...
t2,        1,   THCREATE,   1,   A,    ...
```

```
t3,         2,   INT_ENTR,   1,   A,    ...
t4,         2,   INT_EXIT,   1,   A,    ...
t5,         1,   MSG_SND,    1,   A,    To: B ...
t6,         1,   MSG_RECV,   3,   B,    From: A ...
.....
tn,         1,   KER_CALL,   1,   A,    SIGKILL ...
```

The above snippet shows the life-cycle of a process as recorded by the system. The process A, once created, spawns a thread, addresses an interrupt, sends message to some other process B before being terminated.

Our experiment uses only a subset of all the information that is contained in a trace. Specifically, our comparisons only consider the following event attributes: *class, event, time, pid, and process name*. This is consistent with related work [13] which reduces the used attributes to similar lists.

The experimental workload consists of different execution scenarios of the available systems. Each set contains normal and anomalous traces. The traces are not shared between the scenarios. Each scenario runs between 10 to 20 seconds. Each trace contains all logging information for a single scenario, and within one scenario each trace is approximately the same length in terms of run time. For a given scenario, we split the set of traces arbitrarily into a training subset and a testing subset where a trace belongs to only one of those subsets. Depending on the detection technique, the training set can contain just normal traces or it can contain a combination of normal and anomalous traces (e.g., we need anomalous traces as part of the training set for the comparison with Neural Networks). Similarly, the testing set can contain a set of normal traces, a set of anomalous traces, or a combination of both.

The experimental setup and workload is specific for embedded systems. In particular, we picked four embedded devices (details later in the section), and used some test scenarios to collect traces. As most events in embedded systems are periodic in nature, our test cases attempt to cover general cases that exhibit periodicity. Each scenario contained two sets of traces: $\mathbb{T}^{\mathfrak{t}}$ and $\mathbb{T}^{\mathfrak{a}}$. Of these sets, $\mathbb{T}^{\mathfrak{t}}$ trains the system with relevant information to be able to identify anomalous traces from the set $\mathbb{T}^{\mathfrak{a}}$. Recall that an anomalous trace is one that does not conform with the expected behavior learnt from the training traces.

- *Hexacopter*: The first scenario uses traces from an unmanned aerial vehicle (UAV) platform, which implements non-trivial software and systems control [8]. The platform runs on beaglebone white with ARM Cortex-A8 processor and comprises a networked system of hardware and software components. The hexacopter is field tested through several mission-critical applications including iceberg monitoring on the open sea and creating infrared maps over critical Canadian Solar infrastructure. For our experiment, we created two test scenarios. In both scenarios, the training traces correspond to the hexacopter running normally. The classes of anomalous traces involved, for instance, a periodic recursive listing on the file system[1] and getting stuck in a tight loop.

- *QNX CAR infotainment device*: The second set of scenarios uses QNX CAR [5], which is the leading in-car infotainment system. The device was running QNX Neutrino on an *i.MX6Q* (Sabre lite) board with an ARM Cortex A9 quad-core processor. We created the following scenarios on this platform:

Normal trace	**Anomalies**
Idle	Induced network traffic, user inputs
Play MP3	Seeking, fast-forward, different song
Play Video	Seeking, fast-forward, different video
Run *top* command	Induced network traffic, more shell tasks

- *QNX-based BlackBerry Z10 phone*: The third set of scenarios uses the BlackBerry phone. These phones run a modified version of QNX Neutrino, however, the tracelogging facilities are still available. We created the following set of scenarios on this platform:

Normal trace	**Anomalies**
Record video	Zooming and toggling the camera light on/off
Play youtube video	Playing different video, toggling HD on/off
Play game	Change sound, using different controls
Run flash applet	Reload page, leave page

- *Embedded target*: This scenario uses a non-commercial demonstrator platform for QNX Neutrino [4]. The hardware runs on the *i.MX6Q* (sabre lite) similar to our QNX CAR case study. For this scenario, the platform executes a sound alarm application displaying audio frequencies recorded through the microphone. The normal behavior runs the system in a quiet setup. The anomalies include producing different loud sounds nearby.

[1]`while [ 1 ]; do sleep 2; ls -lR /; done`

- *Variable delay traces*: Finally, we ran SiPTA on a set of traces which were generated on QNX Car platform by varying the delay between the events. We emulated this behavior in the traces by inserting delays between two instances of high system activity. For example, a scenario in which the device pings the network every other second, and another in which the interval is increased to 5 seconds. Considering the slow traces as normal and the faster ones as anomalies, SiPTA was able to correctly identify and appropriately label the traces as we show in Chapter 5.

4.2 Evaluation Criteria

To measure the effectiveness of SiPTA, we need to compare the results to different existing techniques. The prevalent metric for comparison in anomaly detection is receiver operating characteristics curve (ROC) analysis [18].

ROC Analysis

ROC analysis is a common technique in research to compare different classifiers based on their performance [43, 31]. ROC analysis explains the trade-off between the true positive rate (TPR), plotted on the y-axis, and the false positive rate (FPR), plotted on the x-axis. To compare different classifiers, the common approach [18] is to consider the classifier with the higher area under the ROC curve as the better classifier.

Figure 5.1 shows an ROC curve for one of the experiment runs. The algorithm calculates the values for a point *(FPR, TPR)* using Equations 4.1 and 4.2 where a classifier threshold interprets the input probabilities into a binary output of 0 (negative) or 1 (positive).

$$\text{TPR} = \frac{\text{Positives correctly classified}}{\text{Total Positives}} \tag{4.1}$$

$$\text{FPR} = \frac{\text{Negatives incorrectly classified}}{\text{Total Negatives}} \tag{4.2}$$

To obtain all points for the ROC curve when using Markov Model, the algorithm varies the threshold over a range of all input probabilities to obtain the corresponding points $(\texttt{FPR}, \texttt{TPR})$ plotted in the figure. In SiPTA, this corresponds to varying the value r.

In addition to the area under the ROC curve, some important characteristics of the curve help with the analysis of the classifier performance. For example, point $(0, 1)$ indicates perfect classification while the region under the dotted line $\texttt{TPR} = \texttt{FPR}$ indicates that the classification is worse than making a random guess.

4.3 Comparison to Alternative Approaches

To evaluate SiPTA, we compare it to other approaches implementing alternative concepts. For example, one alternative uses a stochastic approach, which assumes that anomalies change the probabilistic characteristics of event transitions in a sequence of system events. The most popular technique for this approach is the first-order Markov Model technique [14]. Another alternative that we use for comparison is Neural Networks [10]. The two techniques were chosen due to their popularity in the domain and thus a wider applicability for existing work. The disadvantage of using Neural Networks however is the inability to use the ROC analysis to compare its performance to the other techniques, as discussed later in this section.

4.3.1 Markov Model Technique

Markov Model is a discrete-time stochastic process used to study the probability of the change of a random variable value. First-order Markov Models are commonly used to study the probabilistic characteristics of a single transition between two events within the trace sequence [44, 22, 42, 43].

To compare to the performance of SiPTA, we implemented the anomaly detection engine using first-order Markov Model. For an input trace sequence, an *event* represents a Markov Model state so that the Markov Model will describe the probability of occurrence of a transition between an *event* and its first successor. Following the work-flow first-order Markov Model approach, the preprocessor splits the trace into sub-traces based on the *process name* and extracts only the *event name* and *event class* attributes. For each sub-trace, the Markov Model calculates a transition probability matrix [44], which indicates the probability of transition between any trace entries. The averaged transition probability matrices calculated for the training set describe the normal behavior of the system.

In the testing phase, the classifier compares the transition probability matrix of the test trace and the normal behavior matrix to decide if the test trace is normal or anomalous. We assign an anomaly flag for each transition in the test trace that occurs with a probability value that lies outside a defined region around the mean value of probabilities that describe the normal behavior of the system. For each experiment, we performed several experiment runs using different sizes for that region to select the region size that yields the best ROC curve. For the final binary classification, the percentage of anomalous transitions within the trace indicates, whether the trace is anomalous or not. Varying a threshold over the range of the percentages, ranging from 0% to 100%, yields points of ROC curve as described earlier.

For the sake of simplicity of the implementation, transition probability matrices consider only the transitions in a randomly selected normal trace during the training phase instead of considering all possible transitions combinations. This consideration reduces the calculations by excluding the transitions that rarely occur and have no effect on the final classification result.

4.3.2 Neural Networks Technique

Artificial Neural Networks (NN) are massively connected networks of computational nodes or *"neurons"*. The nodes are usually organized into layers (input, output, and hidden layers) with weighted connections between them [24]. As the network learns, it updates the weights on the connections to improve classification.

For the purpose of comparison, the Kohonen self-organizing network (KSON) was used. The network uses unsupervised learning algorithms to cluster inputs into groups with similar characteristics. The learning is called unsupervised because the output characteristics of the network are determined internally and locally by the network itself, without any data on desired outputs. The nodes distribute themselves across the input space to recognize groups of similar input vectors, while the output nodes compete among themselves to be fired one at a time in response to a particular input vector [24]. Thus, due to this competitive learning, similar input vectors activate physically close output nodes. We want to take advantage of this characteristic of KSON to classify the traces.

The input vectors for the network are an encoded representation of the events in a trace. To generate the encoding we extract event names from the trace and then count the number of occurrences of each event. The count for each event is then scaled by dividing it by the total number of logged events in the trace. The input vector is thus a collection of event to count mappings for the trace.

The training sets for the network need to contain both clean and anomalous traces. When the network is trained with only clean traces, it is not able to classify anomalous traces as part of a different cluster during testing. Thus, unlike other approaches, Neural

Networks imposes constraints on the training set. During the testing phase, the network determines the cluster that the trace belongs to and thus classifies the trace. The classification is typically discrete, with the output being either 0 (clean) or 1 (anomalous), however the value can be within that interval in case of more uncertainty.

The difficulty with using Neural Networks technique for comparison with the other approaches is the lack of a classifier threshold. The classification takes place within the internal structure of the network using specialized learning algorithms. We can thus alter the structure of the network, but cannot alter any thresholds that influence the cluster that the network will choose. As a consequence, we can only report the detection rate for this technique but cannot perform an ROC analysis.

Chapter 5

Experimental Results

Table 5.1 shows the detection rates (TPR) and false-alarm rates (FPR) for each of the three approaches, namely SiPTA, Markov Model and Neural Networks. Contrary to Neural Networks, SiPTA and Markov Model implement a binary classifier. This allows us to use ROC curves to compare their performance, but a similar comparison cannot be done with Neural Networks.

As mentioned in Section 4.2, the points $(\texttt{FPR}, \texttt{TPR})$ asymptotically represent the ROC curve for each approach. Figures 5.1, 5.2, 5.3, and 5.4 show the ROC curves for the hexacopter, QNX-Car audio play, QNX-Car run *top* command, and variable ping speed experiments, respectively. The remaining experiments yielded ROC curves similar to Figures 5.1 and 5.2 which show perfect classification for both SiPTA and Markov Model.

ROC curves clearly demonstrate the trade-off between the detection rate and the false-alarm rate for a binary classifier. ROC curve points closer to $(0, 1)$ indicate better results with 100% detection rate and 0% false alarm rate. Although such points are most desirable, for our comparisons in Table 5.1, we favor the detection rate over false-alarm rate. The reason behind this preference is that for most cases the penalty of false negatives outweighs false positives.

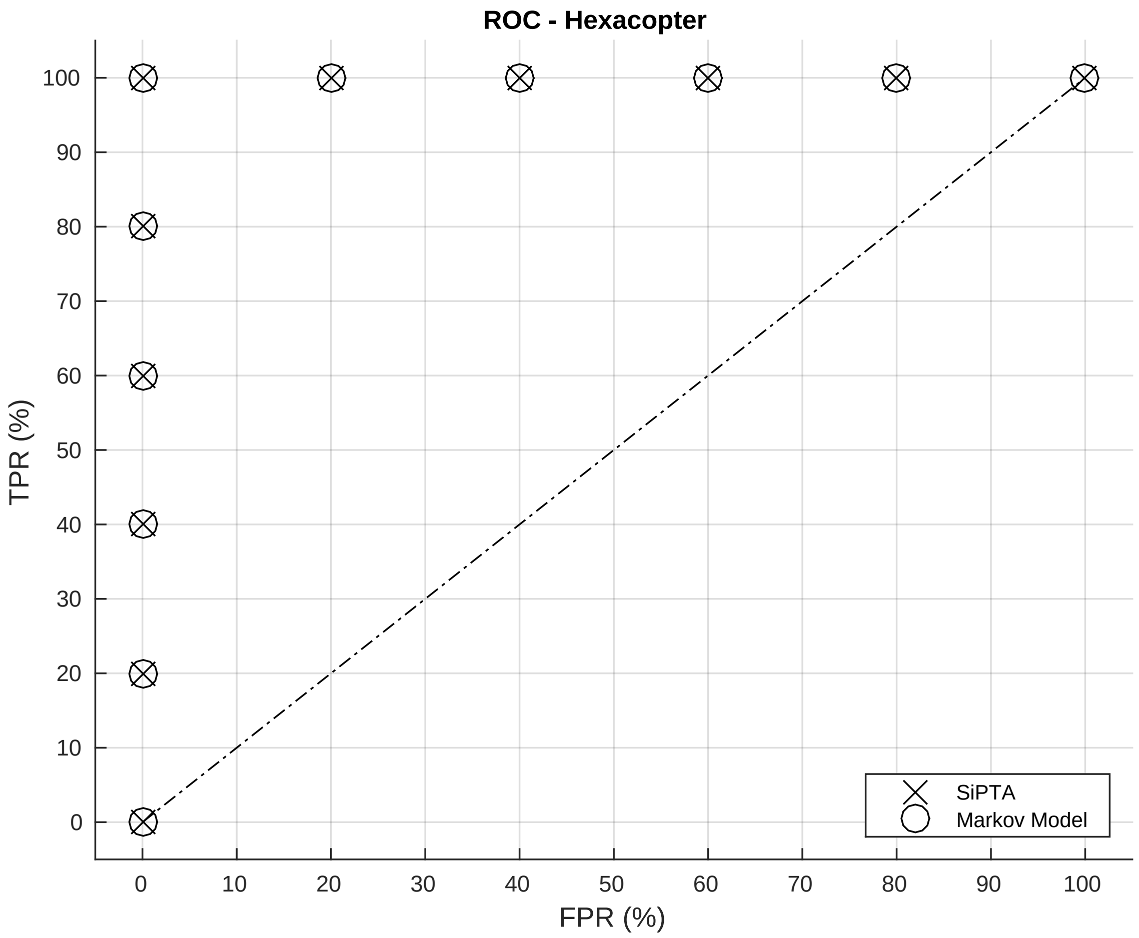


Figure 5.1: ROC for hexacopter

Case studies	SiPTA		Markov Model		Neural Networks	
	Detection Rate	False-Alarm Rate	Detection Rate	False-Alarm Rate	Detection Rate	False-Alarm Rate
Hexacopter	100%	0%	100%	0%	91%	0%
QNX-Car Idle	100%	0%	100%	0%	91%	0%
QNX-Car Play MP3	100%	0%	100%	0%	45%	0%
QNX-Car Play Video	100%	0%	100%	0%	83%	0%
QNX-Car Run *top* command	100%	0%	80%	20%	26%	0%
Variable speed ping	100%	0%	80%	0%	75%	0%

Table 5.1: Results summary

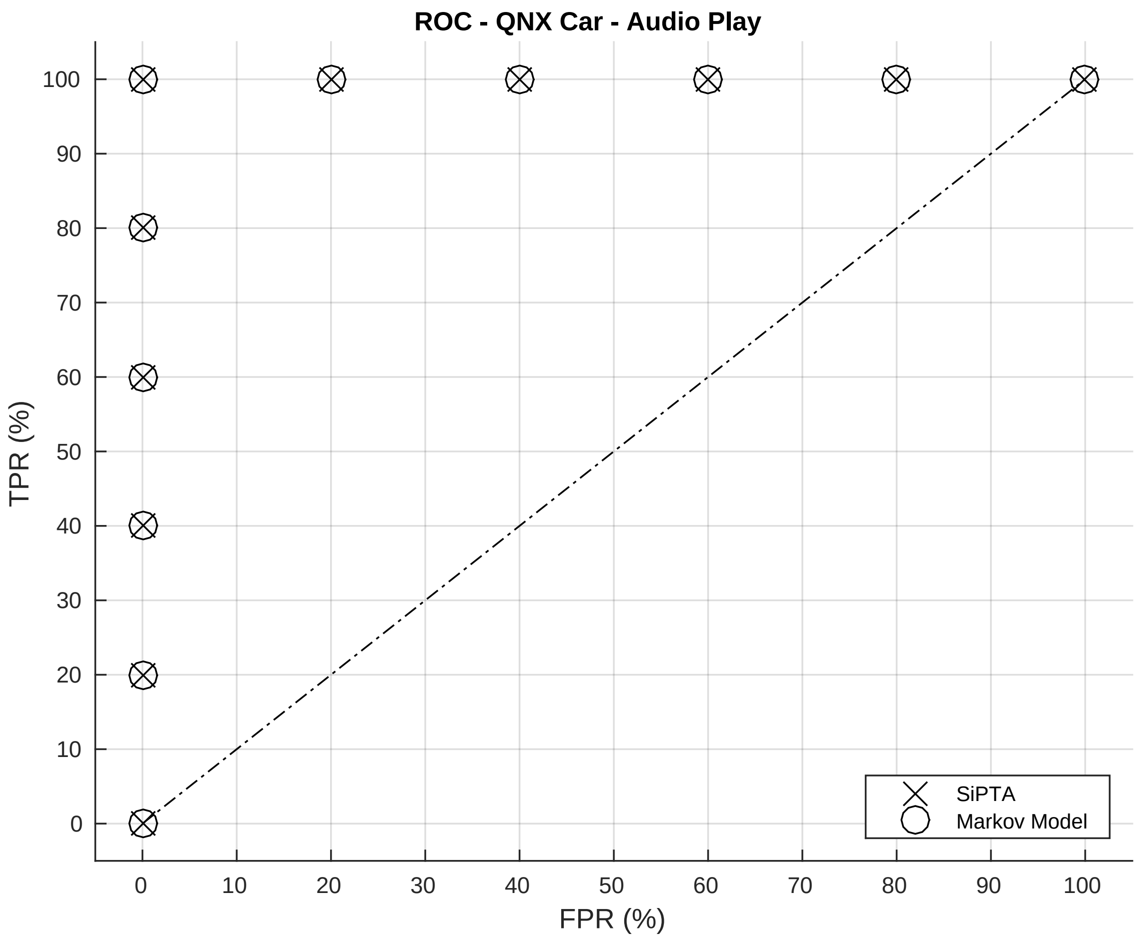


Figure 5.2: ROC for QNX car audio play

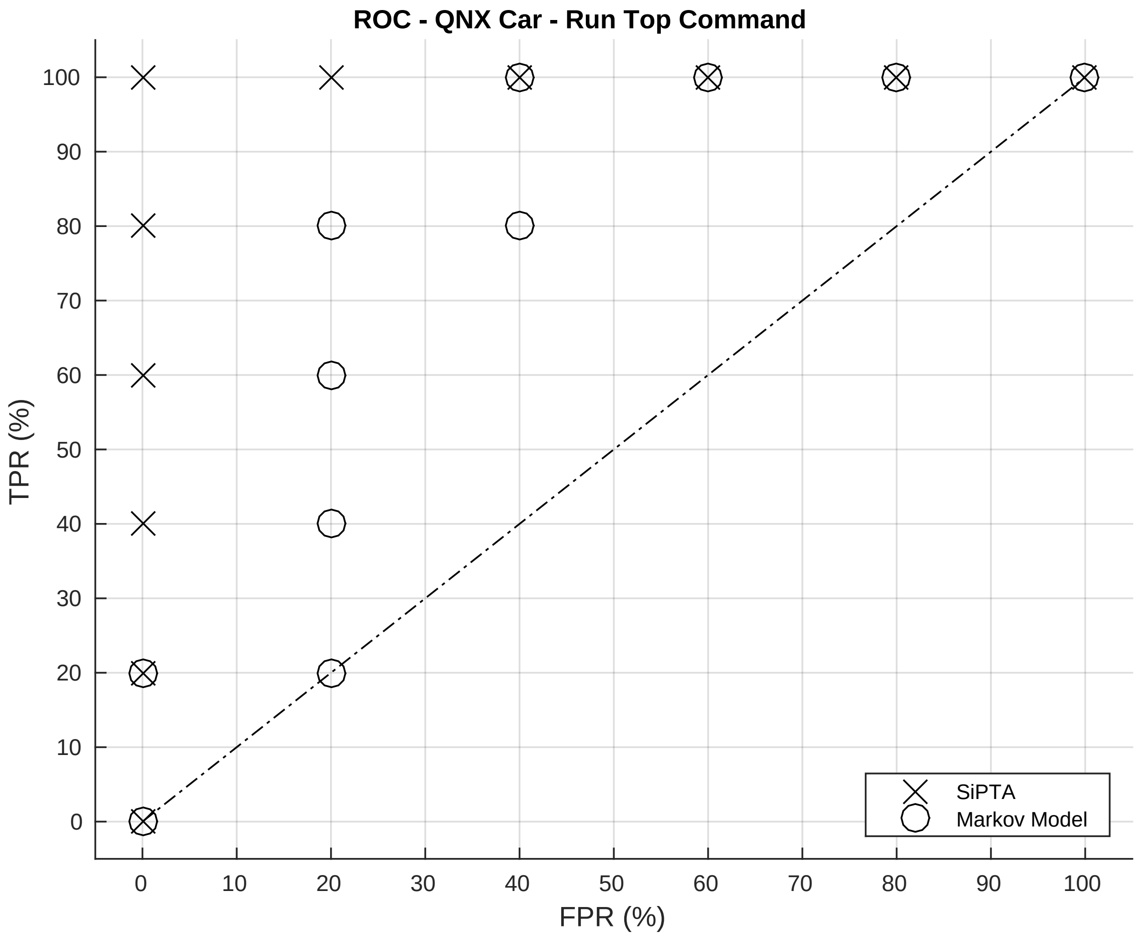


Figure 5.3: ROC for QNX car - *top* command

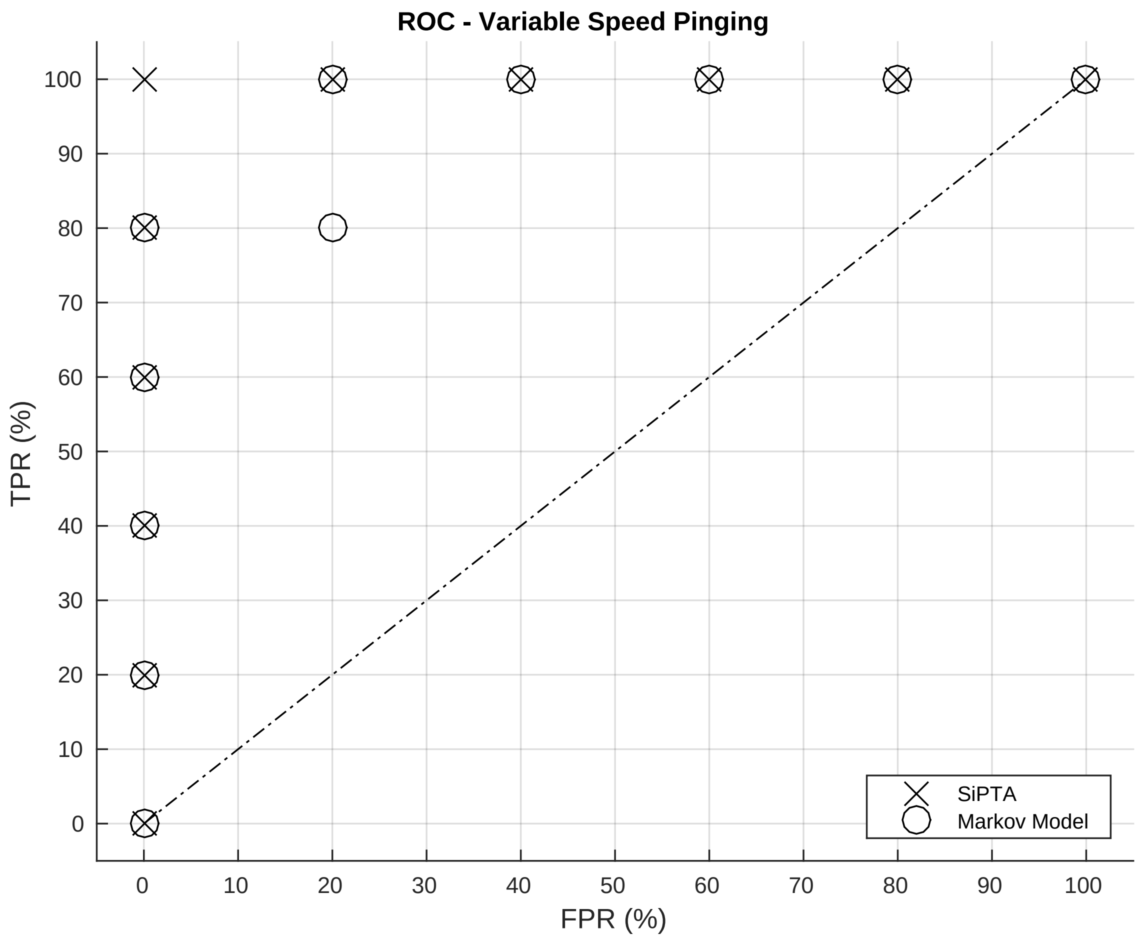


Figure 5.4: ROC for QNX car pinging scenario

Chapter 6

Discussion

SiPTA outperforms all other approaches. In every studied case, our approach yields better results than other studied approaches. For instance, as Figures 5.4 and 5.3 indicate, SiPTA yields perfect classification results for the variable speed ping scenario while other techniques do not. One conjecture for why SiPTA worked is that a change in the ping frequency introduces more irregular behavior in the trace (or less depending on the direction of the rate change). With this irregularity, more noise will appear on the spectra of the channels. As these spectra are normalized, this noise grows in significance as the peaks show less magnitude. Because for this case, we consider higher ping speed as anomaly, the peak-frequency and DC-significance metric to have lower values in anomalous case. For example, Figure 6.1 shows two spectra for the parameter `THREAD-THREADY-2367525`; one plot for a normal and the other for an anomalous case. Figure 6.2 also shows two spectra, in this case for the parameter `THREAD-THREADY-1`. The distinction between the peak-frequency values for the normal and anomalous cases is visible and obvious.

Markov Models fail on certain scenarios. The Markov Model results indicate that the technique does not work for all experiments as SiPTA does. The Markov Model technique calculates the probabilities of transitions between events where the classifier aims to find any irregularities among those probabilities. Although the technique seems to work reasonably well and is currently the dominantly used one [13], it failed to handle certain scenarios. We conjecture that Markov Model fails for these scenarios, because the anomaly has an insignificant effect on the transition probabilities. This is due to events changing their inter-arrival time without affecting their transitions probabilities, which represents a whole class of anomalies that SiPTA can detect, however, Markov Model cannot.

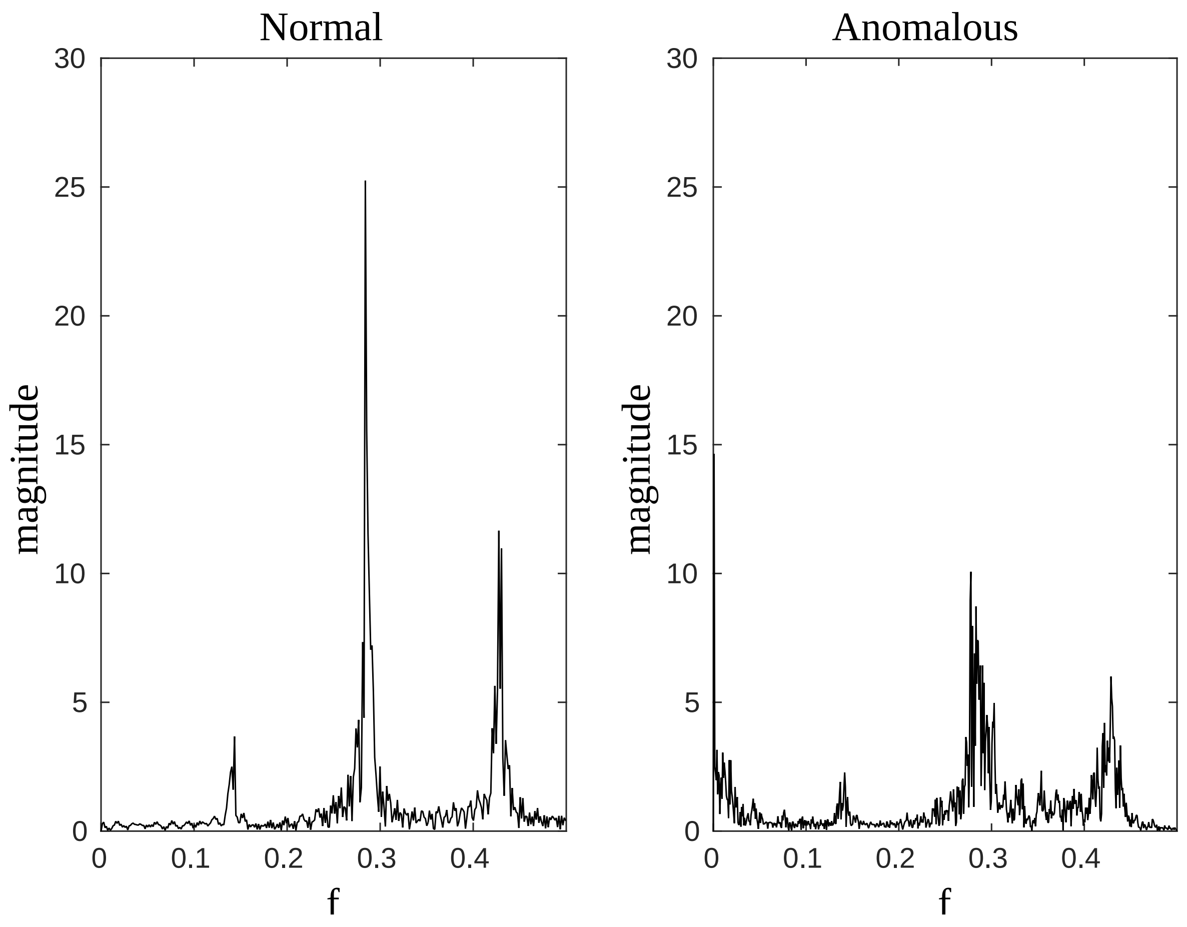


Figure 6.1: Spectra of parameter `THREAD-THREADY-2367525`

Neural Networks are ill-suited for trace-based anomaly detection. Neural Networks is an established technique that is used for detecting anomalies, however, due to its limitations, it is less suitable for this given problem. The results for Neural Networks indicate that the technique classifies anomalous traces only for some of the scenarios. Neural Networks are only able to classify anomalous traces unless trained with at least one anomaly similar to the one occurring. Table 5.1 shows that Neural Networks have a 0% false alarm rate throughout the experiments, because when the network is not trained with anomalous traces, then it will tend to default to classifying traces as normal. So, if the occurring anomalous trace is not similar enough to any anomalous trace used in training, then the network will classify the trace as normal. Unlike SiPTA and Markov Model, Neural Networks require a different training set comprising both normal and anomalous traces.

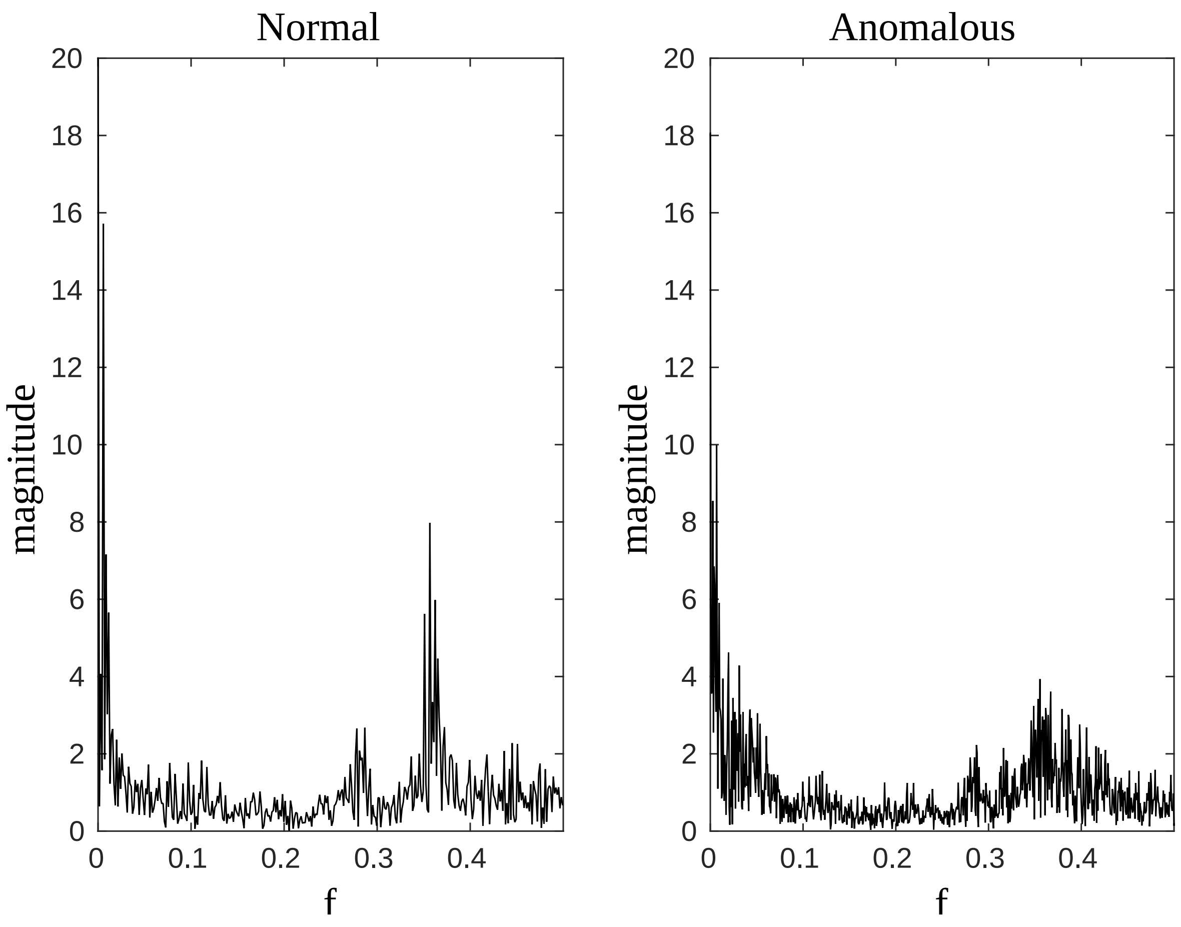


Figure 6.2: Spectra of parameter `THREAD-THREADY-1`

The concepts are widely applicable. It can be observed that SiPTA has an inherently modular structure. Each component, namely trace-to-signal modeling, signal processing algorithm, metrics, and classification technique can be modified to suit application specific scenarios as long as the modules complement each other. For example, one can use some alternative method to model trace-to-signal mappings other than assigning channels to each parameter. Similarly, one can plug-in another signal processing algorithm that exploits some other domain knowledge.

Threats to validity. Our framework and SiPTA base on the assumption that anomalies show changes in the periodicity of events in the trace; otherwise such anomalies will pass undetected. Furthermore, similar to all other approach, SiPTA assumes that the trace contains evidence of the anomaly. Our approach still needs supervised learning from a labeled trace set, similar to Markov Models. While SiPTA was able to outperform all other studied approaches and we created a comprehensive set of scenarios and traces, naturally more evidence is necessary to gain confidence that the system will consistently outperform other approaches also in other settings. Also, as stated in Section 3.5.5, there is minimum trace demand for training traces.

Chapter 7

Conclusion

Identifying an incorrect behavior is crucial for safety-critical embedded systems. In this work we demonstrated that, with an appropriate application of signal processing algorithms on execution traces, one can identify an incorrect system behavior. We demonstrated the feasibility of such an approach by implementing these algorithms into SiPTA. In our experiments, we performed a holistic evaluation of SiPTA by running it on execution traces from varied execution scenarios. To demonstrate the effectiveness of SiPTA, we compared it with state of art techniques of Markov Model and Neural Networks. The results indicate that SiPTA outperforms all the studied contemporary approaches.

APPENDICES

Appendix A

Configuring MATLAB Implementation

In this chapter, we will delve into how to configure the MATLAB implementation

The implementation of SiPTA in MATLAB uses a single `xml` [12] file to configure different aspects of the implementation. The `xml` file contains elements. Each element might contain multiple other elements or a single text field. There are no attributes used in the configuration `xml` file.

Now we will do a breadth-first examination of the configurations in the configuration `xml` file.

As depicted at Illustration A.1 the top of the `xml` tree there is an element with tag `All`. This element contains the following sub elements as following.

- **`mode`** is the running mode which can take one of the following three values.
  - *`release`*: In this mode, SiPTA takes the databases containing train and analyze dataset, and runs the entire anomaly detection from scratch, and then loses all the data that was produced in the middle.
  - *`debug`*: Unlike release mode, debug modemode calculates and stores all the features for a set of input traces as well as all data that was produced in the middle of calculation. Then this data can be restored and redistributed as the user wants between training and analyzing sets. It is useful for debugging and testing different parts of SiPTA and testing the performance of SiPTA as it allows reusing the results of a greater part of the calculation.

  - *`debug2`*: This mode is like the `debug` mode but it also allows spectrum likelihood. The danger with this mode is that the features of spectrum likelihood uses enormous amounts of memory, so this mode has to be used with caution.

- **`workerCnt`**: This number is the number of allowed processors to work on the anomaly detection. This number must be less than or equal to the number of present CPUs in the system. If not so, SiPTA will exit with an error.

- **`dataBase`**: This element contains sub-elements for configuring how SiPTA should access trace databases.

- **`ser`**: This element contains sub-elements for configuring how SiPTA extracts time series and spectra.

- **`trace`**: This element contain sub-elements which specify the address of different trace databases.

- **`levels`**: This element contains the setting that SiPTA uses which features. In this context *`level`* means *feature extraction methods*.

- **`levelNo1` ... `levelNo`*`N`***: These elements contain configurations for different feature extraction methods specified in element `levels`.

Illustration A.1: element `All` sub-elements

```
<?xml version="1.0" encoding="utf-8"?>
<All>
    <mode>RunningMode</mode>
    <workerCnt>WorkerCount</workerCnt>
    <dataBase>
        ...
    </dataBase>
    <trace>
        ...
    </trace>
    <ser>
        ...
    </ser>
    <levels>
        ...
    </levels>
    <levelNo1>
        ...
    </levelNo1>
    ...
    <levelNoN>
```

```
        ...
    </levelNoN>
</All>
```

Now we will examine the configuration in `dataBase` element as depicted in Illustration A.2. This element contains children elements that have text fields. These text fields are then used to configure how SiPTA will access trace databases.

- **`driver`**: This setting represents the driver type according to MATLAB [2] database toolbox [3] configurations. For example, for using SQLite [33] databases on Linux, database should use JDBC [37]. The correct configuration for driver then has to be `org.sqlite.JDBC`.
- **`urlPrefix`**: This represents the urlPrefix string which database toolbox [3] needs to be configured. For example, according to the previous settings, this string should be `jdbc:sqlite:`.
- **`workingDir`**: As SiPTA is designed to use read-only compressed databases, it requires a temporary directory in which it can store the extracted databases and connect to them. This directory is determined by `workingDir` element.

Illustration A.2: element `dataBase` sub-elements

```
<?xml version="1.0" encoding="utf-8"?>
<All>
    ...
    <dataBase>
        <driver>DriverType</driver>
        <urlPrefix>URLPrefix</urlPrefix>
        <workingDir>WorkingDirectory</workingDir>
    </dataBase>
    ...
</All>
```

Now we will examine the configuration in `trace` element as depicted in Illustration A.3. This element contains children elements that have text fields. These text fields are then used to configure how SiPTA will choose different databases as training, analyzing, clean and dirty trace sources.

- **`cleanList`**: In `debug` mode or `debug2` mode presented in Illustration A.1, this will point to the text file which contains the addresses to clean trace databases. This field is not important in release mode.

- **`dirtyList`**: In `debug` mode or `debug2` mode presented in Illustration A.1, this will point to the text file which contains the addresses to dirty trace databases. This field is not important in release mode.

- **`trainList`**: This will point to the text file which contains the addresses to train trace databases. In `debug` mode or `debug2` mode, this list have to be a subset of `cleanList`.

- **`trainList`**: This will point to the text file which contains the addresses to analyze trace databases.

Illustration A.3: element `trace` sub-elements

```
<?xml version="1.0" encoding="utf-8"?>
<All>
    ...
    <trace>
        <cleanList>CleanTraces</cleanList>
        <dirtyList>DirtyTraces</dirtyList>
        <trainList>TrainTraces</trainList>
        <analyzeList>AnalyzeTraces</analyzeList>
    </trace>
    ...
</All>
```

Now we will examine the configuration in `ser` element as depicted in Illustration A.4. This element contains children elements that have text fields. These text fields are then used to configure how SiPTA will construct time series and frequency series.

- **`src`**: This will determine what would be the source for construction of time series. It could be either `RealTime` or `LogicalTime`. `RealTime` means that the sequence of time is real time in which events occurred and `LogicalTime` means the index number of the trace entry is treated as the time source. The proposed method and the results presented in this thesis, all used the option `RealTime`

- **`const`**: This will determine how the time series will be constructed. for example the value `Diff` determines the use of `diff` function which is the only option used in SiPTA right now.

- **`chClm`**: This will determine which columns of the trace will be used to form $\{\mathbb{P}_0, ..., \mathbb{P}_{n-1}\}$. In this thesis, we used the columns `class,event,pname`.

- **`realTimeClm`**: If the value `RealTime` is chosen for `src`, then this will determine the name of the column which contains the real-time information.
- **`norm`**: This will determine the method for normalization and construction of spectra. In this thesis, we used `NormPer` which means that spectra are created by normalization of periodograms.
- **`normDC`**: This is a `Yes`/`No` field which determines whether the DC value is included in the normalization or not.

Illustration A.4: element `ser` sub-elements

```
<?xml version="1.0" encoding="utf-8"?>
<All>
    ...
    <ser>
        <src>Source</src>
        <const>ConstructionMethod</const>
        <chClm>ChannelColumnList</chClm>
        <realTimeClm>RealTimeColumn</realTimeClm>
        <norm>NormalizationMethod</norm>
        <normDC>IsDCIncludedInNormalization</normDC>
    </ser>
    ...
</All>
```

Now we will examine the configuration in `levels` element as depicted in Illustration A.5. This element contains children elements that have text fields. These text fields are then used to configure the metrics that SiPTA will use for feature extraction.

- **`cnt`**: This element determines the number of metrics will be used which should be a positive integer.
- **`no`**M: For any $1 \leq M \leq$ `LevelCount`, this will determine which method will be as metric number M. The currently implemented methods are `ChannelExistence`, `ChannelLength`, `DCSignificance`, `MultiPeak`, and `SpectrumLikelihood`.

Illustration A.5: element `levels` sub-elements

```
<?xml version="1.0" encoding="utf-8"?>
<All>
    ...
    <levels>
```

```
        <cnt>LevelCount</cnt>
        <no1>Metric1</no1>
        ...
        <noN>MetricN</noN>
    </levels>
    ...
</All>
```

The next set of elements are named as `levelNo1` through `levelNo`N where N = `LevelCount` where `LevelCount` is from Illustration A.5. These elements contain configurations of their corresponding metric in sibling `levels` element These configurations have similar child elements. As depicted in Illustration A.6, these child elements are **`general`**, **`train`**, and **`analyze`** which are the configuration of that metric during feature extraction, training, and analysis, respectively.

Illustration A.6: element `levelNoM` sub-elements

```
<?xml version="1.0" encoding="utf-8"?>
<All>
    ...
    <levelNoM>
        <general>
            ...
        </general>
        <train>
            ...
        </train>
        <analyze>
            ...
        </analyze>
    </levelNoM>
    ...
</All>
```

References

[1] Audit data from MIT Lincolin lab. http://www.ll.mit.edu/mission/. Accessed: 2014-03-01.

[2] MATLAB. http://www.mathworks.com/products/matlab/. Accessed: 2015-11-25.

[3] MATLAB Database Toolbox. http://www.mathworks.com/products/database/. Accessed: 2015-11-25.

[4] QNX Accelerator Kits. http://www.qnx.com/products/reference-design/acceleratorkits.html. Accessed: 2014-01-01.

[5] QNX CAR Platform for Infotainment. http://www.qnx.com/products/qnxcar/. Accessed: 2014-01-01.

[6] QNX Neutrino RTOS. http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html. Accessed: 2014-01-01.

[7] System call dataset from University of New Mexico. http://www.cs.unm.edu/~immsec/data-sets.htm.

[8] Unmanned Aerial Vehicle (UAV). https://uwaterloo.ca/embedded-software-group/projects/unmanned-aerial-vehicle-uav-exemplar. Accessed: 2014-11-10.

[9] ISO-26262, Road Vehicles – Functional Safety. http://www.iso.org/iso/catalogue_detail?csnumber=43464, 2011.

[10] A. K. Ghosh and A. Schwartzbard. A Study in Using Neural Networks for Anomaly and Misuse Detection. In *Proc. 8th USENIX Security Symposium*, pages 23–36. USENIX, 1999.

[11] Emin Aleskerov, Bernd Freisleben, and Bharat Rao. Cardwatch: A neural network based database mining system for credit card fraud detection. In *Computational Intelligence for Financial Engineering (CIFEr), 1997., Proceedings of the IEEE/IAFE 1997*, pages 220–226, 1997.

[12] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 16, 1998.

[13] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009.

[14] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly dtection for discrete sequences: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 24(5):823–839, 2012.

[15] X. Cheng, K. Xie, and D. Wang. Network Traffic Anomaly Detection Based on Self-Similarity Using HHT and Wavelet Transform. In *Fifth International Conference on Information Assurance and Security, IAS*, volume 1, pages 710–713. IEEE, 2009.

[16] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT press, 1999.

[17] G. Creech and J. Hu. A Semantic Approach to Host-Based Intrusion Detection Systems Using Contiguous and Discontiguous System Call Patterns. 2013.

[18] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[19] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[20] Ryohei Fujimaki, Takehisa Yairi, and Kazuo Machida. An approach to apacecraft anomaly detection problem using kernel feature space. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 401–410. ACM, 2005.

[21] J. Gao, G. Hu, X. Yao, and R. K. C. Chang. Anomaly Detection of Network TrafiňĄc Based on Wavelet Packet. In *Asia-Pacific Conference on Communications. APCC*, pages 1–5. IEEE, 2006.

[22] S. Jha, K. MC. Tan, and R. A. Maxion. Markov Chains, Classifiers, and Intrusion Detection. In *csfw*, volume 1. Citeseer, 2001.

[23] Leslie A Johnson. DO-178C, Software Considerations in Airborne Systems and Equipment Certification. *Crosstalk, October*, 1998.

[24] F.O. Karray and C. De Silva. *Soft Computing and Intelligent Systems Design: Theory, Tools and Applications*, volume 1. Pearson Education Limited, 2004.

[25] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[26] Vipin Kumar. Parallel and distributed computing for cybersecurity. *IEEE Distributed Systems Online*, (10):1, 2005.

[27] Leveson, N. G. and Turner, C. S. An Investigation of the Therac-25 Accidents. *Computer*, 26(7):18–41, 1993.

[28] Lions, J.-L. Ariane 5 flight 501 Failure, 1996.

[29] J. W. S. Liu. Real-Time Systems, 2000.

[30] W. Lu and A. A. Ghorbani. Network Anomaly Detection Based on Wavelet Analysis. *EURASIP Journal on Advances in Signal Processing*, 2009, 2009.

[31] S. S. Murtaza, W. Khreich, A. Hamou-Lhadj, and M. Couture. A Host-based Anomaly Detection Approach by Representing System Calls as States of Kernel Modules. 2013.

[32] Alan V Oppenheim, Ronald W Schafer, John R Buck, et al. *Discrete-Time Signal Processing*, volume 2. Prentice-hall Englewood Cliffs, 1989.

[33] Mike Owens and Grant Allen. *SQLite*. Springer, 2010.

[34] Kay I Penny. Appropriate critical values when testing for a single multivariate outlier by using the mahalanobis distance. *Applied Statistics*, pages 73–81, 1996.

[35] Andy D Pimentel, LO Hertzbetger, Paul Lieverse, Pieter Van Der Wolf, and Ed F Deprettere. Exploring embedded-systems architectures with artemis. *Computer*, 34(11):57–63, 2001.

[36] S. Rawat and C. S. Sastry. Network Intrusion Detection Using Wavelet Analysis. In *Intelligent Information Technology*, pages 224–232. Springer, 2005.

[37] George Reese. *Database programming with JDBC and Java.* " O'Reilly Media, Inc.", 2000.

[38] M. Salagean and I. Firoiu. Anomaly Detection of Network Traffic Based on Analytical Discrete Wavelet Transform. In *8th International Conference on Communications (COMM)*, pages 49–52. IEEE, 2010.

[39] Clay Spence, Lucas Parra, and Paul Sajda. Detection, synthesis and compression in mammographic image analysis with a hierarchical image probability model. In *Mathematical Methods in Biomedical Image Analysis, 2001. MMBIA 2001. IEEE Workshop on*, pages 3–10. IEEE, 2001.

[40] James Stewart. *Calculus: Early Transcendentals.* Cengage Learning, 2015.

[41] Michail Vlachos, S Yu Philip, and Vittorio Castelli. On periodicity detection and structural periodic similarity. In *SDM*, volume 5, pages 449–460. SIAM, 2005.

[42] N. Ye, X. Li, Q. Chen, S. M. Emran, and M. Xu. Probabilistic Techniques for Intrusion Detection Based on Computer Audit Data. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 31(4):266–274, 2001.

[43] N. Ye, Y. Zhang, and C. M. Borror. Robustness of the Markov-Chain Model for Cyber-Attack Detection. *IEEE Transactions on Reliability*, 53(1):116–123, 2004.

[44] Nong Ye et al. A markov chain model of temporal behavior for anomaly detection. In *Proceedings of the 2000 IEEE Systems, Man, and Cybernetics Information Assurance and Security Workshop*, volume 166, page 169. West Point, NY, 2000.

[45] Mohammad Mehdi Zeinali Zadeh, Mahmoud Salem, Narendra Kumar, Greta Cutulenco, and Sebastian Fischmeister. SiPTA: Signal processing for trace-based anomaly detection. In *Embedded Software (EMSOFT), 2014 International Conference on*, pages 1–10. IEEE, 2014.

[46] M. Zhou and S. D. Lang. Mining Frequency Content of Network Traffic for Intrusion Detection. In *Proceedings of the IASTED International Conference on Communication, Network, and Information Security*, 2003.