Hierarchical framework for early intrusion detection in embedded computing systems

Muhammed Fauzi Abbas

2019

Muhammed Fauzi Abbas (2019). Hierarchical framework for early intrusion detection in embedded computing systems. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/136535>

<https://doi.org/10.32657/10356/136535>

HIERARCHICAL FRAMEWORK FOR EARLY INTRUSION DETECTION IN EMBEDDED COMPUTING SYSTEMS

MUHAMED FAUZI BIN ABBAS

School of Computer Science and Engineering
Nanyang Technological University

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of

Doctor of Philosophy

2019

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

7 September 2019

.....

Date

.....

Muhamed Fauzi Bin Abbas

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

7 September 2019

.....

Date

.....

Prof. Thambipillai Srikanthan

Authorship Attribution Statement

This thesis contains material from four papers accepted at conferences in which I am listed as an author.

Chapter 3 is published as Muhamed Fauzi Bin Abbas, Alok Prakash and Thambipillai Srikanthan. "Power profile based runtime anomaly detection." In 2017 TRON Symposium (TRONSHOW), pp. 1-9, Dec 2017.

The contributions of the co-authors are as follows:

- Prof. Srikanthan provided guidance for the initial project direction.
- I developed the technique and prepared the manuscript draft.
- Dr. Alok Prakash revised the manuscript.

Chapter 4 is published as Muhamed Fauzi Bin Abbas and Thambipillai Srikanthan. "Low-complexity Signature-based Malware Detection for IoT Devices." In Applications and Techniques in Information Security (ATIS), pp. 181-189, Springer, 2017.

The contributions of the co-authors are as follows:

- I developed and evaluated the technique, and prepared the manuscript draft.
- Prof. Srikanthan reviewed the research to identify shortcomings and edited the manuscript draft.

Chapter 5 is published as Muhamed Fauzi Bin Abbas, Sai Praveen Kadiyala, Alok Prakash, Thambipillai Srikanthan and Yan Lin Aung. “Hardware Performance Counters-based Runtime Anomaly Detection using SVM.” In 2017 TRON Symposium (TRONSHOW), pp. 1-9, IEEE, Dec 2017.

The contributions of the co-authors are as follows:

- Prof. Srikanthan provided the initial project direction.
- I developed the technique and prepared the manuscript draft.
- Dr. Yan Lin prepared the platform environment.
- Dr. Alok and Dr. Sai Praveen revised the manuscript.

Chapter 6 is based on the joint work with Manaar Alam who was a PhD student of Prof Debdeep Mukhopadhyay from the Institute of Technology Kharagpur, India and full-time intern student under the guidance of Prof Srikanthan, NTU. Manaar Alam was responsible for the formulation of LAMBDA. Other contributors to this work include Dr. Sai Praveen Kadiyala, Yash Shrivastava and Sikhar Patranabis.

Chapter 7 is published as Muhamed Fauzi Bin Abbas, Alok Prakash and Thambipillai Srikanthan. “Hierarchical Framework for Runtime Intrusion Detection in Embedded Systems.” In 2019 TRON Symposium (TRONSHOW), IEEE, Dec 2019.

The contributions of the co-authors are as follows:

- Prof. Srikanthan provided the initial project direction.
- I developed the technique and prepared the manuscript draft.
- Dr. Alok revised the manuscript.

7 September 2019

.....
Date

.....
Muhamed Fauzi Bin Abbas

Acknowledgements

I would like to take this opportunity to thank many people who have directly or indirectly helped me during this research. First and foremost, I want to express my heartfelt gratitude to my supervisor and mentor, Prof Thambipillai Srikanthan, whose guidance during the course of this project has been invaluable. Apart from his role as my PhD supervisor, I also wish to thank Prof Srikanthan for all that he has done for me during my long association with him. Despite being entrusted with many responsibilities, he has always found time to discuss the directions of this thesis with me. His infinite patience and understanding in this regard are truly admirable.

The guidance and discussions with Dr Alok Prakash have also been extremely helpful throughout this research to bring about clarity and sanity. Without his help, this thesis will not have been possible.

The administrative support rendered by Mr Chua Ngee Tat, Jeremiah was the best I could have asked for. His willingness to provide assistance whenever I needed any equipment or administrative support made him an indispensable part of the research laboratory and this thesis.

I would also like to thank my wife, Nurul Huda, and my little princess, Wardah Sheifa, for their love and patience, without which I would not have come this far. Last but certainly not the least, this thesis is dedicated to my parents whose unwavering support and prayers are only second to their unfaltering trust in me.

Contents

Statement of Originality	i
Supervisor Declaration Statement	ii
Authorship Attribution Statement	iii
Acknowledgments	v
List of Figures	xiii
List of Tables	xvi
Acronym	xviii
Abstract	xx
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	3
1.3 Main Contributions	4
1.4 Organization of Thesis	6
1.5 Publications	8
2 Literature Review	9
2.1 Intrusion of Malicious Software into Computing Platforms	10
2.2 Intrusion Detection System	12
2.2.1 Analysis Techniques	13

2.2.1.1	Static Analysis	13
2.2.1.2	Dynamic Analysis	14
2.2.2	Intrusion Detection System Classification	14
2.2.2.1	Network-based Intrusion Detection System (NIDS) . .	14
2.2.2.2	Host-based Intrusion Detection System (HIDS)	15
2.3	Features used in IDS	15
2.3.1	Network Traffic	16
2.3.2	System Calls (SYSCALLs)	17
2.3.3	Architectural Events	21
2.3.4	Power	22
2.3.5	IDS using Multiple Features	23
2.3.6	Techniques for IDS	24
2.3.6.1	Signature-based Detection	24
2.3.6.2	Anomaly-based Detection	25
2.3.6.3	Machine Learning in Anomaly Detection	26
2.3.6.4	Hybrid-based Detection	27
2.4	Benchmarks used in IDS	27
2.4.1	Benchmark Dataset	28
2.4.2	Benchmark Application	28
2.4.2.1	Existing Malware	28
2.4.2.2	Existing Benchmark Suite	29
2.4.2.3	Generating Applications to Test Anomaly Detection . .	29
2.4.2.4	Code Snippets for Specific Malicious Behaviour	30
2.5	Summary	31

3	Power Trace-based Real-time Anomaly Detection	36
3.1	Introduction	36
3.2	Proposed Power Trace Anomaly Detection	38
3.2.1	Correlation Analysis	39
3.2.2	Offline Power Model Generation	39
3.2.2.1	Implementation of the Proposed Techniques	43
3.2.3	Runtime Anomaly Detection using Power Model	43
3.2.4	Model Aggregation	46
3.3	Experimental Setup	49
3.3.1	Platform	49
3.3.2	Benchmark	51
3.3.2.1	Input Data	51
3.4	Test Results	53
3.4.1	Power Trace Models	54
3.4.2	Detection Results	56
3.4.3	Response Time	60
3.5	Summary	63
4	System Call Trace-based Classification of User-space Malware	65
4.1	Introduction	65
4.2	Runtime System Call-based Malware Classification	67
4.2.1	Implementation of the Proposed Techniques	68
4.3	Extracting Malware Signatures using System Call Sequences	69
4.3.1	System Call Trace Acquisition	70
4.3.2	Malware Signature Extraction	71
4.3.3	Malware Signature Refinement	73

4.3.4	Recursive Malware Signature Extraction	74
4.3.5	Discussion	74
4.4	Malware Classification	75
4.4.1	Real-time Classification	76
4.5	Experiments and Results	79
4.5.1	Runtime Detection	79
4.5.2	Experimental Setup	81
4.5.3	Test Results	82
4.5.3.1	Detection Accuracy Evaluation	82
4.5.3.2	Classification Accuracy	84
4.5.3.3	Real-time Malware Classification	84
4.5.4	Comparison with State-of-the-art	86
4.6	Summary	87
5	Rootkit Detection at Kernel-space	89
5.1	Introduction	89
5.2	Anomaly Detection using Architecture-specific Features	91
5.2.1	Methodology Overview	92
5.2.2	Generating Anomalous Applications	94
5.2.2.1	Input Data	96
5.2.3	Selection of Hardware Performance Counters	96
5.2.4	Support Vector Machine Classifier	100
5.3	Experiments and Results	101
5.3.1	Experimental Setup	101
5.3.2	Benchmarks	102
5.3.3	Test Results	104

5.4	Kernel Verification	105
5.4.1	Selection of Vulnerable System Calls	108
5.4.2	Selection of Hardware Performance Counters	109
5.4.3	Generating the Classifier Model	109
5.4.4	Experiments and Results	110
5.4.4.1	Experimental Setup	110
5.4.4.2	Benchmarks	112
5.4.4.3	Experimental Results	113
5.5	Summary	115
6	Lightweight Assessment of Malware for Embedded Architectures (LAMBDA)	117
6.1	Introduction	117
6.2	The Anomaly Detection Methodology	118
6.2.1	Association of Indicator Programs with HPCs	120
6.2.2	LAMBDA Methodology	123
6.2.2.1	Data Collection	124
6.2.2.2	Training Phase	124
6.2.2.3	Detection Phase	125
6.3	Experimental Results	128
6.3.1	Benchmarks	128
6.3.2	Experimental Setup	129
6.3.3	Test Results	129
6.3.3.1	Deriving the sensitivity matrix	129
6.3.3.2	Calculating λ value	130
6.3.3.3	Selection of Significance Level α	131
6.3.4	λ score for program proximity	133

6.3.5	Runtime λ	133
6.3.6	Evaluating the Proposed Detection Mechanism	135
6.3.6.1	Detecting Unknown Malware	135
6.3.6.2	Comparison with other state-of-art methods	136
6.3.6.3	Comparison with Machine Learning based Approaches	137
6.4	Summary	139
7	Hierarchical Framework for Intrusion Detection	141
7.1	Introduction	141
7.2	Metrics for Intrusion Detection	141
7.2.1	Power Trace	143
7.2.2	System Call Trace	143
7.2.3	Architecture-specific Features	144
7.2.4	LAMBDA	144
7.2.5	Pros and Cons of Features	144
7.3	Power Trace as the First Line of Defence	145
7.3.1	Monitoring power at runtime	146
7.4	Integrating Power Trace with System Call Trace for Resource-efficient Classification of User-space Malware	148
7.4.1	Obtaining System Call Trace	149
7.4.1.1	Software Techniques for System Call Identification . .	149
7.4.1.2	Hardware Techniques for System Call Identification . .	150
7.5	Integrating Power Trace with Architecture-specific Features for Lightweight Kernel Verification	151
7.5.1	Obtaining Architecture-specific Features	153
7.5.1.1	Software Techniques for Monitoring Architecture-specific Features	154

7.5.1.2	Hardware Techniques for Monitoring Architecture-specific Features	155
7.6	Lightweight Integrated Technique for Intrusion Detection	157
7.6.1	System Call Trace vs Architecture-specific Features	157
7.6.2	Combining Architecture-specific Features with System Call Trace and Power Trace Metric	159
7.7	Incorporating LAMBDA for Robust Anomaly Detection and Classification	161
7.7.1	Obtaining HPC Concurrently for Multiple Indicators	163
7.8	Framework for an Integrated Intrusion Detection System	164
7.8.1	Leveraging HPCs for Non-intrusive Monitoring	164
7.8.2	Multiplexing HPCs Features for Non-intrusive Monitoring	166
7.8.3	Integration for Ease of Deployment	167
7.8.4	Multi-modal Analysis	169
7.8.5	Performance Evaluation	170
7.9	Summary	176
8	Conclusions And Future Research	178
8.1	Summary of Research	178
8.2	Recommendations for Future Research	180
	References	184
	Appendix A	206
	Appendix B	210

List of Figures

2.1	Top 6 types of malware detected worldwide	11
3.1	Overview of the proposed technique	38
3.2	Offline Model Generation	40
3.3	Flowchart for generating application's power model and threshold	42
3.4	Runtime Anomaly Detection	44
3.5	Runtime Anomaly Detection using Correlation	45
3.6	Generating a single power model and selecting a threshold value	48
3.7	EFM32's Development Platform [1]	50
3.8	AEM tool used to obtain the application's power trace	50
3.9	Averaging Results	55
3.10	Power Profile for Functionality Equivalent Applications	56
3.11	Power Profile for Other Benchmark Applications	57
3.12	Rapid runtime power analysis	61
3.13	Rootkit detected using power trace	62
4.1	Overview of the proposed technique	67
4.2	Overview of Enhanced Signature Extraction Process	70
4.3	Signature Extraction	71

4.4	Refining Malware Signature Extraction	73
4.5	Enhanced Malware Signature Extraction	74
4.6	Extracting Signatures for Malware Classification	76
4.7	Common behaviour across malware families	77
4.8	Sample of <code>strace</code> output with relative time flag	77
4.9	Time Slicing System Call Trace	78
4.10	Signature Extraction within Time Slice	78
4.11	Runtime Malware Detection	80
4.12	Sorted Malware Signature by percentage (% found in malware)	83
4.13	Refined vs Enhanced Technique (Detection %)	83
4.14	Extracting Signatures for Real-time Malware Classification	85
4.15	Real-time Classification Accuracy	86
5.1	Overview of the proposed method	92
5.2	Instruction trace filter and buffer for runtime detection	93
5.3	Averaged standard deviation of hardware performance counters across twelve benchmark applications	99
5.4	Terasic's SoCKit used in the experiment	102
5.5	Hardware Performance Counter change when a Rootkit is present	108
5.6	Image of ODroid-XU3 Platform	111
6.1	Training Phase for LAMBDA	119
6.2	Detection Phase for LAMBDA	120
6.3	Behaviour of <code>ls</code> , <code>instruction tuple</code>	123
6.4	Average gap between LAMBDA (λ_0) score for benign and malware programs under different confidence interval.	132
6.5	Illustrating Runtime λ	135

6.6	Comparison of F_1 -score of the proposed method LAMBDA with other machine learning based classifiers in Box-plot representation	138
7.1	Components in a Typical Embedded System	142
7.2	Power extraction on existing embedded platforms	147
7.3	Malware Detection using Power Trace and System Call Trace at runtime	149
7.4	Detecting Rootkit using Power Trace and Architecture-specific Features at runtime	153
7.5	High-level structure of incorporating HPC into OS	154
7.6	Cortex-A9 processor monitors hardware events of LEON-3 processor	155
7.7	The frequency associated with three possible values of PC upon trapping SVC instruction via PMU and isolating the <i>read</i> SYSCALL [2].	156
7.8	Power Trace of an Application during a Rootkit Attack	158
7.9	The Proposed Framework using Multiple Metric	160
7.10	Holistic Systems Monitoring with Robust Malware Classification and System Fault Detection	162
7.11	Framework Overview for an Integrated Intrusion Detection System	164
7.12	Multiplexing Multiple HPC-based Techniques	166
7.13	A single core dedicated for monitoring the HPCs and Power Trace for the other cores	167
7.14	Hypervisor used to realise a monitoring solution for Intrusion Detection Framework	168

List of Tables

3.1	Benchmark Applications	52
3.2	Benchmark for training and testing	53
3.3	Obtaining the threshold for each application	55
3.4	Correlation coefficient amongst all benchmark applications	58
3.5	Results for trained and test applications	60
3.6	Response time of test applications	62
4.1	Processing System Call Trace	71
4.2	Malware Data Set	81
4.3	Results of Malware Classification	84
4.4	Results of Real-time Malware Classification	86
4.5	Comparison with State-of-the-art	87
5.1	Anomalous applications generated (application mixing)	95
5.2	Anomalous applications generated (code patching)	96
5.3	Considered Benchmarks from CHStone [3]	97
5.4	List of hardware performance counters found in the LEON-3 processor	98
5.5	Benchmark Applications for Training and Testing Phase	103
5.6	Training and Testing Application HPC Count	106

5.7	Results of the Test Phase	107
5.8	Duration of SYSCALL frequently targeted by a Rootkit	107
5.9	List of Monitored System Call	109
5.10	Train and Test Dataset	112
5.11	Test Results	114
6.1	Total number of different System Calls executed for the Indicator Programs <i>ls</i> and <i>ps</i> in the presence of a benign program and two different types of malwares	122
6.2	List of Selected Malware and Benign Programs for Train and Test Sets	128
6.3	Sensitivity Matrix $\mathcal{W}_0$ that is comprised of four <i>observers</i> and 5 <i>indicators</i> . . .	129
6.4	Count Matrix for 'GSM'	130
6.5	Count Matrix for 'Agent'	130
6.6	Model performance of univariate t-test for the different confidence interval . . .	132
6.7	λ_0 score for benign programs and malware	134
6.8	Analysis of <i>Unknown Malware</i>	136
6.9	Comparison with State-of-the-art	137
6.10	Comparison based on Average F_1 -score, Detection time, Memory usage and CPU Utilization	138
7.1	Pros and Cons of Various Metrics	145
7.2	Overhead of using HPC vs Power	146
7.3	Total System Call with and without 'suterusu' Rootkit	159
7.4	Detection and Classification Truth Table	159
7.5	List of HPCs used in the integrated framework	165
7.6	Extended Truth Table for Multi-Modal IDS	169
7.7	Testing and Training Sets for Performance Evaluation	171
7.8	Training Phase Dataset	171

Acronyms

ADPCM	Adaptive differential pulse code modulation
AES	Advanced encryption standard
ASLR	Address Space Layout Randomisation
BOFM	Bounded Feature Space Behavior Modeling
CRC	Cyclic Redundancy Check
CWE	Common Weakness Enumeration
DDoS	Distributed Denial of Service
DEP	Data Execution Prevention
DFADD	Fas Associated with Death Domain Homologue in Drosophila
DT	Decision Tree
FPU	Floating-Point Unit
GSM	Global System for Mobile communication
HIDS	Host-based Intrusion Detection System
HMM	Hidden Markov Models
HPC	Hardware performance counters
IDS	Intrusion Detection System
IoT	Internet of Things
JOP	Jump-oriented-Programming

KNN	k-Nearest Neighbors
LAMBDA	Lightweight Assessment of Malware for Embedded Architectures
MALWARE	Malicious Software
MLP	Multi-Level Perceptron
MMU	Memory Management Unit
NB	Naive-Bayes
NIDS	Network Intrusion Detection System
OPF	Optimum-path Forest
OS	Operating System
PC	Program Counter
PMI	Performance Monitoring Interrupts
PMU	Performance Monitoring Units
PUT	Program Under Test
RBF	Radial Basis Function
ROP	Return-oriented Programming
RSVM	Robust Support Vector Machines
RTOS	Real-time Operating System
SBS	Strip, Bind and Search
SHA	Secure Hash Algorithms
SoC	Systems on Chip
SOM	Self-organising Map
SRD	SAMATE Reference Dataset
SVD	Singular Value Decomposition
SVM	Support Vector Machines
SYSCALL	System Call
TLB	Translation Lookaside Buffer
WAD	Watchmen Anomaly Detection

Abstract

It has been well recognized in the literature that intrusion detection in embedded computing systems is both challenging and computationally demanding. The ominous threat of malware in such systems necessitates the need for lightweight real-time intrusion detection methods that are capable of keeping up with the fast evolving and sophisticated malware attacks. In this thesis, a hierarchical framework for real-time intrusion detection in embedded computing systems has been proposed.

Runtime monitoring of power consumption provides for a lightweight mechanism for the early detection of anomalies. A new metric called δ has been implemented to facilitate a lightweight detection of anomalies by comparing the offline generated power trace model. Unlike existing methods, the proposed lightweight technique provides for the early detection of anomalies at runtime. The proposed method has been extensively validated to confirm that runtime power monitoring can be effectively deployed to separate benign applications from abnormal ones that have been compromised by user-space malware and/or operating systems compromised by a Rootkit.

Next, a lightweight technique has been proposed for the detection and classification of user-space malware using system call trace. This approach has been shown to rely only on a minimal set of malware signatures to effectively detect malware. The proposed approach achieves comparable performance as the state-of-the-art technique reported in the literature with a detection rate of 93.4% and classification rate of 83%. However, unlike the existing approaches, the proposed method is capable of real-time classification at runtime without compromising the detection and classification rates by observing only the Hardware Performance Counters (HPC) for extracting system call trace specific information.

Subsequently, low-level architecture-specific feature using HPC has been proposed to detect anomalies in the kernel-space (operating system) caused by a Rootkit. A model that relies on the combined analysis of benign system calls and tainted system calls compromised by Rootkit is introduced to detect such anomalies. Unlike existing methods that incur high monitoring overheads, the proposed technique employs a lightweight method that yields a comparable level of accuracy in detecting Rootkits. Experimental results confirm that the proposed approach can achieve a classification accuracy rate of up to 96% while requiring significantly less resource to store the models.

Another metric called λ is also proposed to distinguish benign programs from those infected with malware to facilitate the early detection of anomalous operations at runtime. The proposed method relies on statistical hypothesis testing based on runtime measures obtained using HPC. Performance evaluations confirm that the HPC based observation of operating system indicators makes λ an effective metric for differentiating benign and malicious applications. The detection accuracy rate is as high as 90% with a false positive rate of 1.52% and a false negative of close to 0% respectively. This compares very favourably with the state-of-the-art method reported in the literature.

The proposed techniques have been integrated into a framework to provide for a hierarchical method to facilitate the simultaneous detection of malware and Rootkit at runtime. Moreover, the proposed techniques can be incorporated into any embedded computing system without any modifications to the application or the operating system. This multi-metric approach has resulted in an effective and yet lightweight approach to realize a robust intrusion detection system. It has been shown that the framework is capable of detecting a wide range of anomalous behaviour at real-time i.e. Worm, Trojan, Viruses and Rootkit without the need to instrument the application or the operating system. It is noteworthy that the proposed techniques have also been integrated to incorporate fault-tolerance without compromising the overall

accuracy and performance. Moreover, the framework is also capable of separating abnormal operations due to system faults and potential zero days.

Finally, the proposed framework is well suited for incorporation into embedded computing systems with limited computing resources to avail the first line of defence for early detection of malware, Rootkit, system faults and potential zero days at run-time.

Introduction

1.1 Motivation

Cybersecurity has drawn a lot of attention in the past few years, primarily due to the ever-rising number of hacks and security threats. These hacks and security vulnerabilities have (especially in critical systems) resulted in the loss of both money in the order of billions of dollars and (in extreme cases) human lives. Furthermore, the vast number and diversity of electronic devices and appliances used daily by human beings, which are prone to cyberattacks, make this a concern for not only government agencies [4] and corporations but also the masses. Both industry and academia have been trying hard to keep up with these rising threats through increased investment and research into techniques to detect malicious software (malware) and anomalies so that appropriate and timely actions can be taken.

It has become common practice for hackers to repackage seemingly innocuous applications with malware embedded in them in order to gain unauthorized access to systems. A recent study [5] showed that 86% of malware samples found are repackaged versions of legitimate applications with malicious payloads. The process of differentiating such malignant applications from their legitimate counterparts is referred to as ‘anomaly detection’. Prompt anomaly detection is challenging since much of a

malware's behaviour closely matches that of legitimate applications.

In recent times, the presence of anomalies in various systems, gadgets, and networks has been reported to be on a steep rise. These anomalies can range from theft of sensitive data to triggering harmful system behaviour that can lead to severe damage. Evasive techniques used by these anomalies are ever-evolving, e.g., usage of obfuscation techniques like packing, polymorphism, and metamorphism. These attacks make it challenging for the intrusion detection system to keep up-to-date with the latest techniques. Currently, anomaly-based detection is at the forefront of detecting zero-day attacks [6]. There are various methods of generating a model for anomaly detection. However, most of these methods incur significant resource and performance overheads during implementation.

Amidst the ever-growing popularity of IoT devices that are typically ultra resource-constrained embedded systems, the number of malware [7] targeting IoTs has also multiplied. The past few years have witnessed the most significant jump in the number of malware targeted towards IoT devices [8], [9]. Other works, such as [10], [11] have shown that there is also a rapid increase in the number of embedded malware that impacts every aspect of modern life. This could be attributed to the large number of embedded devices that are relied upon in our daily lives, from smartphones to vehicles, kitchen appliances, and television sets that all contain embedded systems and software. Hence, the impact of a malware attack can be very critical and potentially life-threatening [4], [12]. Additionally, malwares have become more sophisticated, targeted, and stealthier with time [13]. This makes it extremely challenging to detect them promptly and take appropriate actions. Further, the computational effort required to be on a constant lookout and rapidly differentiate malicious application from non-malicious ones (benign) adds to the overhead in a system. This is especially troublesome for resource-constrained embedded systems in general and considerably more so for ultra resource-constrained IoT devices.

1.2 Objectives

The main objectives of this research are to develop novel techniques and design low complexity methodologies for detecting intrusion in an embedded system. The target platform consists of a general-purpose processor and an operating system (OS). The proposed methodologies should incorporate lightweight and accurate detection strategies to facilitate rapid and effective intrusion detection for the system.

1.3 Main Contributions

The major contributions of this work can be listed as follows:

- (i) *Power Trace-based Real-time Anomaly Detection*: Leverage on these power monitoring hardware capabilities and tools to generate the power profile of the unadulterated version of an application which is then stored as models in a database. This power profile is used at runtime to verify the correctness of an application currently being executed through periodic comparisons with the power profile model of the application stored in the database. The proposed approach can be used during runtime to effectively detect the variations in implementation, even when the overall application produces the same output. The contributions in this work have been published in [14].
- (ii) *System Call Trace-based Classification of User-space Malware*: An enhanced signature-based detection system that identifies and stores characteristic signatures for a group of malware instead of storing unique signatures for every malware. An intelligent mechanism to fall back on the existing approach of generating and storing unique signature per malware is also used as a last resort to ensure close to over 93% detection rate. The proposed technique offers multiple advantages over existing solutions, including high accuracy, no false alarm generation, and reduced memory resource requirement to store the signatures. This work has been published in [15].
- (iii) *Rootkit Detection at Kernel-space*: This work leverage on HPCs found in most modern embedded processors and coupled with a machine learning classifier to generate a model that differentiates between a benign version of the system and an anomalous one. A detailed HPC count analysis of various system calls (SYSCALLs) enables us to select the suitable hardware performance counter

required in detecting anomalous applications. At runtime, this offline generated model is used to verify the correctness of the invoked SYSCALL by comparing the model stored in the database with the selected HPC value after the SYSCALL has completed its execution. The proposed approach can be used to effectively detect the anomalies and Rootkit even when the distance between the HPC's values are close to each other. This work has been published in [16].

(iv) *Lightweight Assessment of Malware for Embedded Architectures*: This work leverage on the `tuple`, which is a combination of `Indicator` programs and HPC events that provides a sensitive monitor for detecting the presence of malware. The approach proposes a new metric called λ that is calculated by obtaining the distributions of various `tuples`, which are a combination of `Indicator` programs and HPC events. The sensitivity value of each `tuple` is represented using a sensitivity matrix that is obtained after the training phase. In the testing phase, when an unknown program is executed, LAMBDA classifies it based on the t-tests performed between the distributions of different features of program and benign reference. This evaluation results in the score matrix, which is eventually combined with the sensitivity matrix to obtain a λ value. The proposed approach can be used to effectively differentiate an anomaly from a benign using the λ value.

(v) *Hierarchical Framework for Intrusion Detection Framework*: The primary objective of this work is to present a sophisticated technique for integrating multiple metrics introduced throughout the thesis and propose a lightweight runtime hierarchical intrusion detection framework that can be realised on resource-constrained embedded computing platforms. The proposed framework can take advantage of individual feature characteristics to provide a comprehensive intrusion detection system to eventually provide a last line of defence for these

vulnerable systems. This chapter also goes on to suggest a method to implement this framework on the existing platform without the need to instrument the OS and applications. Portions of the contributions in this chapter have been published in [17].

1.4 Organization of Thesis

- (i) Chapter 2 starts with a detailed survey of existing intrusion detection techniques. After comparing the advantages and limitations of the available options, the chapter focuses on the different classes of the intrusion detection system and describes the existing strategies of deploying them in a static or a dynamic fashion. Various research works done in multiple metrics used in IDS have also been described in this chapter. The chapter is concluded with specific objectives and motivations for the rest of this thesis.
- (ii) Chapter 3 presents the proposed power-trace based anomaly detection methodology for intrusion detection on embedded systems. This is followed by a discussion of the proposed profile matching using correlation analysis to generate the power-trace models of the applications. The same correlation analysis is also used at runtime to detect if the executing application deviates from the model. The chapter introduces a new metric, δ , that helps differentiate between a benign program and an anomalous one. An extended detection strategy is also demonstrated to identify kernel-based anomalies in this chapter.
- (iii) Chapter 4 introduces the runtime SYSCALL-based classification methodology to detect user-space malware. The first work discusses the proposed signature extraction process for malware detection. This is followed by the SYSCALL-based classification to categorize malware into their respective families. Subsequently, it details the method to use SYSCALL signatures for real-time clas-

sification of malware. Finally, the proposed methodology is compared with a state-of-the-art, followed by a summary for this chapter.

- (iv) Chapter 5 proposes a technique that leverages architecture-specific features to detect kernel-space Rootkit in the system. The proposed techniques rely on HPCs to verify the integrity of SYSCALLs in the OS.
- (v) Chapter 6 details an anomaly detection methodology for IDS by monitoring the HPCs of OS indicators. The chapter introduces a new metric, λ that helps to differentiate between a benign program and an anomalous one. The proposed methodology is compared with current state-of-the-art methods.
- (vi) Chapter 7 presents a sophisticated framework for integrating multiple features introduced in the earlier chapters, to propose a lightweight runtime multi-modal intrusion detection framework that can be realised on resource-constrained embedded platforms. The various features have been integrated into a hierarchical framework to create a robust intrusion detection and classification system.
- (vii) Chapter 8 concludes this thesis by summarizing the significant contributions made in this research, followed by recommendations for the future extension from this work.

1.5 Publications

The work presented in this thesis has been published in several conference papers as follows:

Conference Papers

- [C-1]** Muhamed Fauzi Bin Abbas, Alok Prakash, Thambipillai Srikanthan, “Hierarchical Framework for Runtime Intrusion Detection in Embedded Systems,” In 35rd TRON SYMPOSIUM, Tokyo, Japan, December 2019.
- [C-2]** Muhamed Fauzi Bin Abbas, Alok Prakash, Thambipillai Srikanthan, “Power Profile based Runtime Anomaly Detection,” In 33rd TRON SYMPOSIUM, Tokyo, Japan, December 2017.
- [C-3]** Muhamed Fauzi Bin Abbas, Sai Praveen Kadiyala, Alok Prakash, Thambipillai Srikanthan, Yan Lin Aung, “Hardware Performance Counters based Runtime Anomaly Detection using SVM,” In 33rd TRON SYMPOSIUM, Tokyo, Japan, December 2017.
- [C-4]** Muhamed Fauzi Bin Abbas, Thambipillai Srikanthan, “Low-complexity Signature-based Malware Detection for IoT Devices,” In International Conference on Applications and Technologies in Information Security (ATIS), Auckland, New Zealand, July 2017.

Literature Review

The year 2018 was a record year for cybercrime. There has been a rapid increase in malware that impacts every aspect of modern life [18], [19], [10], [11]. These continually evolving attacks are capable of shutting down vital infrastructure and pose a severe threat to national security. These malicious attacks consistently adopt obfuscation techniques to conceal themselves from anti-malware software.

Recently, with the sheer popularity of the Internet of Things (IoT) devices that are typically resource-constrained, more hackers have turned their attention [7] to these vulnerable systems. Hackers have taken meticulous efforts to discreetly embed malware into commonly used applications to gain unauthorized access to these systems [5]. This has led to a trend of intense cyberattacks, with a 62% yearly increase [19]; large corporations are being confronted with the worst data breaches in recent history [8], [9], [20]. This dramatic rise in the number of hacking attempts, often resulting in catastrophic losses, has made information security a critical concern for everyone.

Additionally, the increasing complexity and clandestine nature of recent malware attacks [13] make it extremely challenging to detect malware and take appropriate actions. Moreover, the monitoring overhead required to differentiate malware

from non-malicious ones constantly imposes an additional burden on these resource-constrained embedded devices.

Many techniques have been proposed and employed to safeguard these computing devices from malicious intent. As more devices are being connected to the Internet, the first line of defense is typically at the network side [21], [22], [23], [24]. This is followed by the last line of defense, detection within the host device itself [25], [26], [27], [28], which is a significant area due to the sheer variety of devices, ranging from small sensor nodes to massive server farms.

This chapter begins with a discussion on various topics in the existing intrusion detection system, starting from the methods for malware analysis and the techniques to detect and identify these malicious behaviours in the system, as well as elaborating on their advantages and shortcomings. Next, an extensive review of important research efforts made in the intrusion detection system has been presented, followed by a detailed survey of features and techniques used to detect and identify malware. A review of current benchmarking tools to evaluate these intrusion detection techniques has also been presented. The existing framework for intrusion detection has also been reviewed, and is followed by a discussion on relevant commercial tools. This chapter is concluded with a summary of the specific shortcomings of the existing work and the motivations for this thesis.

2.1 Intrusion of Malicious Software into Computing Platforms

Malicious software or malware refers to any computer code that is designed to cause damage to any computing system and their surroundings intentionally. Intrusion to the system can come in different forms, e.g. including executable code, source code, assembly code, scripts, and active web content. Malware can be categorised by its

malicious intent and behaviour. Figure 2.1 summarises the different categories of malware and their widespread proliferation. Trojans and malicious applications make up the most common types of malware detected worldwide in 2018 [20].

Figure 2.1: Top 6 types of malware detected worldwide

The following briefly describes common types of malware that is categorized [20], [29] according to its intent and how it is spread:

- (i) A trojan malware refers to a seemingly benign program that has a hidden malicious functionality used to access data on a target system. Typically, the attackers use *phishing* to trick users into installing a trojan on their computer.
- (ii) A virus is a self-replicating piece of code that ‘infects’ other computer programs to corrupt it in malicious ways. A virus attempts to spread from one computing platform to another by attaching itself to a host program. When the tainted file is executed, the virus code also runs, infecting the new host and potentially damaging not only the software and data but also the hardware.

- (iii) A worm is similar to a virus but travels the Internet autonomously, exploiting vulnerabilities in network-based devices and spreading from one computing platform to another. The main aim of a worm is to deliver a malicious payload to a target machine, which can facilitate other forms of malicious attacks, such as the invasion of a trojan or the creation of a backdoor. However, even worms without a payload can consume enormous bandwidth, diminish network or local system resources, and possibly cause a denial-of-service.
- (iv) A backdoor malware gains hidden access to a system by eluding standard authentication. Backdoors can be part of an installed program as well as unauthorized modifications to an existing program.
- (v) A Rootkit is a unique form of malware that strives to maintain control over a targeted system while hiding its presence from the owner of the device. It conceals itself from the user by manipulating the OS of the latter. Moreover, Rootkits can prevent a malicious process from being visible in the system's list of processes, and keep its own files from being read.
- (vi) A 'packed' malware refers to any method used to obfuscate malicious code by compressing it within larger, seemingly innocuous data. The compressed data often contains a separate decompression code, or even a self-extracting archive, either of which is used to recreate the original code from the compressed code and then execute the malware. Encryption may also be used to conceal the malware from security software as another means to obfuscate the attack.

2.2 Intrusion Detection System

An Intrusion Detection System (IDS) is a piece of software and/or hardware designed to raise an alarm when it detects malicious activities (e.g. break-ins, penetrations, and

other forms of computer intrusion) trying to compromise the computing system [30]. In addition, IDS can also identify these anomalies and tackle malicious intrusions.

2.2.1 Analysis Techniques

There are generally two approaches to analysing and detecting malware in an intrusion detection system, i.e. static analysis and dynamic analysis.

2.2.1.1 Static Analysis

Static analysis is a technique to analyse malware by analysing the malicious code and finding known patterns (e.g. strings and import functions) without the need to execute the program. It looks for vulnerabilities in source code or binaries, specifically snippets of code, be it in source code or assembly code that are common in most malware. Several studies [31], [32], [33], [34] use static analysis in their intrusion detection.

The most significant advantage of static techniques is their potential to traverse all possible execution paths of a program, while dynamic analysis is limited to a single execution path. In theory, static techniques could also profit from the fact that currently, large amounts of existing malware are actually variations of a small number of malware programs [35], [36].

The drawback of this approach is that it is unable to detect sophisticated and polymorphic malware. In addition, current malware leverages on code obfuscation [37] and code encryption [38] to make disassembly difficult. This is due to the fact that most static techniques rely on disassembling these binary executables that disable them from analysing these complex malicious binaries effectively.

2.2.1.2 Dynamic Analysis

The aforementioned problem can be solved by performing a dynamic analysis, where the behaviour of a program is observed while it is executed in a controlled environment while monitoring various runtime activities called features (e.g. registry changes, SYSCALLs, and performance counters). Several works [39], [40], [41], [42] use dynamic analysis in their intrusion detection.

However, having a large number of features while building a model can hinder the rapid detection of malware. Thus, the relevant set of features needs to be purposefully selected so that the classification model can be built in less time and with high accuracy. Feature selection is a method of identifying essential features that are sufficient for detecting malware successfully. Selecting only relevant features can improve the learning speed as well as the implementation of the model.

2.2.2 Intrusion Detection System Classification

Intrusion detection systems can be classified according to where detection takes place, (be it in the network or host), and the detection method that is employed. Most intrusion detection systems rely on specific features such as network traffic [23], [43], [44], [45], SYSCALL [46], [47], [48], and the embedded system's resources [49], [50], [51] to monitor and analyse the systems for any unusual behaviour.

2.2.2.1 Network-based Intrusion Detection System (NIDS)

The most widely used intrusion detection systems are network-based ones, as they are seen as the first line of defence. A typical setting is a vast network of computing devices connected to the rest of the world via the Internet. The primary reason for NIDS is the frequency of attacks launched by external hackers who want to gain unauthorized access to the network for information theft or to disrupt the network.

Most NIDS monitors the network traffic to analyse the network packets, as well as the service flows for malicious payloads. The data available for intrusion detection systems can be at different levels of granularity, for example, packet-level traces and NetFlow data.

2.2.2.2 Host-based Intrusion Detection System (HIDS)

A host-based intrusion detection system is an intrusion detection system that is capable of monitoring and analysing the internals of a computing system. This was an early method of intrusion detection in which the network was not considered to be on the target system [52].

Anomaly detection techniques, typically applied to host-based intrusion detection, are required to handle the sequential nature of data. Moreover, point-based anomaly detection techniques are not as effective as the more common contextual-based and collective-based anomaly detection techniques. The techniques have to either model the sequence data or compute the similarity between sequences. A survey of different techniques used for this problem is presented by Snyder [53]. Comparative evaluations of anomaly detection for host-based intrusion detection are presented in [54], [55].

2.3 Features used in IDS

A plethora of features or metrics that enable various techniques to be implemented for IDS were discussed. It is noted that these methods are not mutually exclusive, and some of these features can be coupled to enhance the overall reliability and/or detection accuracy of the system, as discussed in the previous section. The following subsection will review the most frequently used features in recent intrusion detec-

tion techniques that are available. In the rest of this thesis, the terms ‘features’ and ‘metrics’ are used interchangeably.

2.3.1 Network Traffic

A common challenge faced by IDS is that the nature of anomalies keep changing over time as the intruders adapt their network attacks to evade the existing intrusion detection solutions. These attacks appear as anomalous patterns but the techniques described in [56], [57] are capable of modelling these data sequentially to detect anomalous sub-sequences.

Reviews of various machine learning algorithms are provided in [21], and can be used to accurately detect anomalies in a network. The authors in [58] have proposed a modelling technique using multiclass support vector machines to identify the various network attacks. They have also integrated the rank-based chi-square feature selection to reduce the scale of the dataset and select the pertinent features for building the model. Wang et al. [59] highlighted that ordinary regression-based detectors are susceptible to outliers in the dataset. They proposed a robust regression-based detector that more accurately models the real network traffic, and is more reliable than the ordinary regression model in the face of outliers. On the other hand, the authors in [22] selected features from the same dataset coupled with Multi-Level Perceptron (MLP) neural network to build a model to be able to differentiate between normal operations and anomalous behaviour. Another study [23], uses an extreme learning machine algorithm [60] that solves the standard classification and regression problems. The machine learning algorithm called Watchmen Anomaly Detection (WAD) is introduced in [45] to analyse traffic generated within a cellular network to detect anomalies in time series, such as peaks, dips and sudden rises and falls, all of which are due to erroneously configured devices and broken equipment. WAD also

incurs significantly less false positives and false negatives as compared to the other algorithms.

Other works like [39] also used a machine learning algorithm but implemented them on hardware to significantly reduce power consumption. In [61], the authors have proposed a meta-heuristic assessment model using correlation analysis and association impact scale to predict intrusions based on the selected feature, which significantly reduces the computational time of measuring association impact.

In [44], Sheikhan and Bostani presented an intrusion detection system that consists of two modules. The first Internet side module uses signatures-based IDS for classifying the Internet traffic. The second 6LoWPAN module uses anomaly-based IDS to identify the attacks from within the host by using 6LoWPAN traffic. The study by Hussein [62] proposed a hybrid approach to reduce the false alarm rate while increasing the detection accuracy. The Snort algorithm [63] was chosen to implement signature-based detection, while the Naive Bayes algorithm [64] was relied upon to implement anomaly-based detection. Naive Bayes algorithm showed better detection performance, in terms of accuracy, as compared to other algorithms such as K-means for anomaly-based detection. However, K-means algorithm showed better results in terms of reducing false alarms.

These network-based techniques perform similarly to a firewall, which works from the network size and not within the local device itself, as the first line of defence against malicious intrusion.

2.3.2 System Calls (SYSCALLs)

One of the earliest works in this area was from Forrest et al. [54]. They discovered that the sequences of SYSCALLs made by a program during routine executions are consistent, but that these sequences change in a malicious application. Short-range

correlations in a SYSCALL trace were used to capture its normal behaviour. Any deviations from this profile could then be considered as signs of potential attacks. Forrest created a database of contiguous SYSCALLs of the specified length k . Incoming SYSCALL trace are divided by k length and are compared against the database of known intrusion. Short-range correlations in a SYSCALL trace were used to capture its normal behaviour. Deviations from this profile could then be considered as signs of potential attacks. Further works capitalized on this first approach by introducing new methods based on relative frequency comparison [65], rule induction techniques [65], or Hidden Markov Models (HMMs) [65], [66]. While results show that HMMs can generate the most accurate results on average, they remain demanding computationally because of the convergence process used to reach the optimal profile. Suggestions to improve the performance of HMM-based intrusion detection such as merging multiple sub-HMMs [66] have been discussed. Such methods lack proof that combinations of models obtained from disjoint sub-sequences yield a final model that correctly accounts for long sequences. These aforementioned techniques focus solely on improving the performance of detection using the ordered sequences of SYSCALLs as a critical portrait of process behaviour.

Another work that used SYSCALL arguments as an important characteristic of the process behaviour is [67]. Therein, models to characterize SYSCALL arguments for identifying anomalous occurrences are built. Unfortunately, intrusion detection relies solely on the models built for SYSCALL arguments and does not use the call sequences. Building on the approach proposed in [67], Maggi et al. [68] improves the intrusion detection process by removing and improving inefficient argument models and leveraging the sequence of SYSCALLs through Markov chains. The authors also introduce clustering based solely on SYSCALL arguments in order to construct and classify the clusters.

The use of SYSCALLs has been widely studied in behaviour-based anomaly detection, since malicious activities often use SYSCALLs to execute privileged operations on system resources. Some take advantage of the sequence of SYSCALLs while others look at the frequency of SYSCALLs being used. However, most of these methods [69] can be circumvented by either a bypass attack or a transformational attack. Bypass attacks take advantage of design deficiencies in the detection system to circumvent the execution monitor and utilize unmonitored SYSCALLs. Transformational attacks such as mimicry attacks can adapt to an attack. By doing so, it goes undetected by the model-based detection system even though it carries the same malicious intent.

Wagner et al. [25] proposed an automated approach using static analysis to observe the SYSCALLs and detect buffer overflow in a sandbox environment. A sandbox is a type of testing environment that isolates software or program execution for evaluation, monitoring or testing. Canzanese et al. [26] use the bag-of-n-gram model to process SYSCALL traces and then apply several supervised classifiers to detect malware on Microsoft Windows hosts. While [55] demonstrated using nearest neighbour algorithm on SYSCALL traces extracted using a sandbox. Lee et al. [27] focused on the integrity of control flow by verifying the program execution path. For this purpose, the program is run in a clean environment to record whole program paths with different inputs. The result is increased security, but even with substantial storage and computational resources, this technique is not able to capture all permissible program execution paths; hence, this solution generates false alarms.

A machine learning-based method for classification of anomalies has been discussed in [70]. This approach detects the anomalies based on their efficiency in embedding function call graphs into selected feature mapping. This method takes into consideration both the relevance and the frequency of occurrence of the opcode sequence. In work [71], SYSCALL arguments, along with their interrelationships,

are taken into account while detecting the anomaly. This approach developed a behavioural Markov model to capture the correlations and hence the abnormalities. In their work [72], Canali et al. compared the complexity of anomaly detection models with their accuracy. It has been reported that the relationship is non-linear across the design space, and analytical reasoning is not sufficient to find the optimal model. With the increases in the number of anomalies, the number of features also increase. In an attempt to restrict the feature count, a bounded model (BOFM) is presented in work [73].

The work in [74] used the difference in opcodes to generate the alarm for an anomaly. The difference in the statistical distribution of the opcode and the rarity of its occurrence is used to detect the anomaly. The computation of similarity between two executables that have a similar opcode frequency is performed in [75]. Along with the relevance of opcode, the weight of sequence is also taken into account based on the ambiguity detection, which is discussed in [76]. This opcode-based method is also used in solving metamorphic anomaly detection [77].

In [78], Sanjeev et al. have presented a frequency-centred model to track down the abnormality of the running process. The work was aimed at modelling anomalies based on their SYSCALL patterns and had included an early prediction feature. A demonstration of delay gadgets overpowering signature-based protection schemes is presented in [79]. In the same work, Sanjeev et al. also presents a reconfigurable software approach in addition to the signature-based defence, which can withstand delay gadgets.

A new class of anomalies called the Jump-oriented-Programming (JOP) is discussed in [80], and is similar to Return-oriented Programming (ROP) [81]. However, JOP anomalies do not rely on the stack memory and 'ret' instruction making them more evasive. In order to handle sophisticated anomalies, a fine-grain control flow

check is presented in [82] and [83]. The check policies cover control transfer of functions, nature of indirect calls, and indirect jumps.

Unfortunately, techniques that use SYSCALLs are implemented mainly in software, and these suffer runtime penalties [84]. In addition, programs that use little or no SYSCALLs such as those solely based on arithmetic operations, and programs which do not have unique SYSCALL behaviours may fail to exhibit unique SYSCALL signatures [85].

2.3.3 Architectural Events

Another type of attack that targets OS, (e.g. the Rootkit attack), has been known to bypass techniques that use SYSCALLs. These attacks attempt to modify or redirect the SYSCALL itself, as opposed to the flow of application that uses these SYSCALLs. Work [86] addresses the detection of such attacks that modify the SYSCALL. They focused on counting the hardware events that occur during each SYSCALL execution. Ozsoy et al. [87] designed a hardware-based anomaly detection that acts as the first line of defence to detect an anomaly. After the prioritization of the hardware classifier, further software detection mechanisms were carried out. In work [42], a Singular Value Decomposition (SVD) is used to reduce the hardware event vectors into a set of characterization features. These reduced features are further used by machine learning techniques to identify anomalies.

Moskovitch et al. [40] describe how to extract various features collected by the Windows performance counter and VTrace, and then employ decision trees, and a Naive Bayes classifier to detect worms. A Self-organising Map (SOM) based approach is presented by Zhai et al. in [88] to detect the anomalies. Here the anomalies are characterized using the processor's program counters and cycles per instruction. Later, these features are used in further detection of anomalies.

Attempts to build an anomaly detector in hardware using performance counters are presented in [89]. This hardware anomaly detector is designed to run securely beneath the system software, so that robustness of the detection is increased. Work mentioned in [90] uses low-level hardware features in anomaly detection. This approach also uses a supervised machine learning technique to classify the hardware features into either benign or anomaly.

The use of machine learning to manage the immense HPC data collected is shown in [40], which selects a few features to generate a model for detecting worm malware. A significant amount of work in [41] uses the HPC-based signature to check the application before execution, and detects anomalies during runtime.

2.3.4 Power

Energy/power monitoring has, frequently and successfully, been used for side-channel analysis [91], [92] to discreetly obtain information about the application currently being executed on a platform. On the other hand, intrusion detection techniques that use energy or power measurement have been limited to obtaining a coarse-grain energy budget of processor or platform [39]. However, coarse-grain energy budget based techniques do not perform well when distinguishing between a maligned version of an application and its unadulterated version, due to the almost similar power consumption of the two [93].

Secondly, energy consumption based detection techniques are reactive rather than proactive. This implies that maligned application can only be detected after the application has finished execution and not during its execution. Such schemes might not be effective in a real-world scenario where a more proactive runtime technique is preferred to detect such maligned at the earliest onset, so that appropriate steps can be employed to minimize the consequences.

The research works presented in [94], [95], [96], [97], [98] use energy as a means for side-channel analysis. These works attempt to extract information or behaviours based on power which they obtain externally. There are also several research works that suggested monitoring energy as a signature [99], [100], [101], [102]. However, energy being a coarse-grain metric does not allow runtime anomaly detection. In addition, many applications with widely different power profile can still have similar overall energy consumption. Hence, it is difficult to differentiate between two applications by using energy consumption as a sole feature for anomaly detection. Arjunan et al. [100] split the energy measurement based on time period to detect abnormal instances, while Kim et al. [102] proposed a technique that monitors the energy consumption and compares them with the normal energy consumption to detect anomalies.

Recently, many processing platforms include sensitive power measuring hardware support and tools [1], [103] to aid in the development of energy-aware applications.

2.3.5 IDS using Multiple Features

While most research focuses on using a single feature as described in the previous subsections, there has been an increasing number of works [104], [105], [99] that look into incorporating multiple features for IDS. Sedjelmaci et al. [104] proposed an IDS framework where a trade-off between detection accuracy, false-positive rates, and energy consumption is achieved using the Nash Equilibrium concept. These anomaly-based techniques are mainly used to detect zero-day attacks. Once an attack is detected, these techniques generate and store a signature of this attack, which can be relied upon in future to detect similar attacks.

Another work in [105] used a feature set consisting of Network data, SMS, CPU utilization, the power consumed, memory occupied by libraries, and virtual memory

utilization to train Naive Bayes, Logistic Regression, Random Forest and SVM models. It was found that the Random Forest algorithm outperformed the other algorithms in detecting the applications with malware intent.

Kurniawan et al. [99] conducted a similar study that investigated the anomalies of features at execution time. These features included power consumption, network traffic and battery temperature. They applied four machine learning algorithms, namely the J48, Random Forest, SVM and LMT algorithms, on a combination of three types of features. Results indicate that battery temperature was not contributing as much to the detection as compared to the remaining two features: power and network traffic.

2.3.6 Techniques for IDS

Many existing techniques consider different aspects of IDS, e.g. monitoring complexity, detection accuracy, and application contexts that are incorporated to strike a balance between these aspects. These analysis strategies can generally be categorised into two major types, signature-based detection and anomaly-based detection. Signature-based algorithms are typically used for detecting known attacks while anomaly-based algorithms are best used for detecting unknown attacks (i.e. zero-day attacks).

2.3.6.1 Signature-based Detection

Signature-based detection systems perform pattern matching, which detect attack signatures that were observed earlier and flag them up on the system. This method has better detection rates as compared to anomaly detection and has less false alarm rates [106]. The observed behaviours present on the host are performed by host-based IDS and provided as an input to the decision engine. Unfortunately, signature-based IDS represents the identifiable features from the previously seen attacks and

are thus unable to detect new attacks.

2.3.6.2 Anomaly-based Detection

Anomalies are patterns in data that do not conform to a well-defined notion of normal behaviour. Typical daily activity tends to show a common behaviour pattern that anomaly-based detection systems can model and keep track of. Tracking any deviation from this typical activity is the key purpose of anomaly detection. Anomaly detection refers to the process of finding patterns in data that do not conform to expected behaviour. These non-conforming patterns are often referred to as anomalies, outliers, or exceptions, depending on the context and applications. The importance of anomaly detection lies in the fact that anomalies in data translate to significant, and often critical, actionable information in a wide variety of applications. Prompt anomaly detection is a challenging task since the behaviour of the application closely matches that of an unadulterated version of the application.

Many anomaly-based intrusion detection systems typically experience high false alarm rates because of the complexity of establishing a strong and pervasive baseline [107]. The high level of deviation that results from the dynamic behaviour of computer activities significantly increases the difficulty of distilling a useful baseline to assess anomalous divergences [108].

The limits of exploring anomaly detection are presented in [109]. This work used opaque constants to develop obfuscation techniques that can evade detection tools. Developing transparent techniques that can detect an anomaly without producing any side effects is discussed in [110]. This method focused on developing virtual hardware extensions that reside outside the OS environment. Creech et al. [111] focused on anomaly detection using contiguous and discontinuous SYSCALL patterns, which reflect the intrinsic activities of the process. Their model aimed at reducing the false alarm rate during the detection process.

Another critical challenge for anomaly detection in this domain is the massive volume of data. The anomaly detection techniques need to be computationally efficient to handle these massive inputs. Moreover, the data typically comes in a streaming fashion, thereby requiring runtime analysis. Another issue that arises because of the massive input is the false alarm rate. Since the data amounts to millions of data objects, just a small percentage of false alarms can make analysis overwhelming for an analyst.

2.3.6.3 Machine Learning in Anomaly Detection

The huge volume of data that requires analysis has led to the usage of machine learning in anomaly detection for multiclass as well as single class settings. A basic multiclass anomaly detection technique using machine learning operates in two steps. First, a model is trained on the standard training data to learn the different classes. Second, each test instance is provided as an input to the model. If the model accepts the test input, it is normal, and if the model rejects a test input, it is an anomaly. This machine learning technique has resulted in several variants, which use different types of algorithms. Another common machine learning algorithm used in IDS is Support Vector Machines (SVM), which have been applied to anomaly detection in the one-class setting. Such techniques use one class learning techniques for SVM and learn a region that contains the training data instances, called a boundary. Kernels, such as the Radial Basis Function (RBF) kernel, can be used to learn complex regions. For each test instance, the basic technique determines if the test instance falls within the learned region. If a test instance falls within the learned region, it is declared as normal, otherwise, it is declared as anomalous. A variant of the basic technique finds the smallest hypersphere in the kernel space that contains all training instances and then determines which side of that hypersphere a test instance lies on. If a test instance lies outside the hypersphere, it is declared to be anomalous.

Song et al. [112] use Robust Support Vector Machines (RSVM), which are robust to the presence of anomalies in the training data. RSVM has been applied to SYSCALL intrusion detection [113].

2.3.6.4 Hybrid-based Detection

However, recent works have attempted to combine both approaches to improve detection and classification of malware [44], [62]. Several studies in [44], [62] proposed a novel hybrid architecture that combines both signature and anomaly-based techniques. While [44] proposed a hybrid architecture based on unsupervised Optimum-path Forest (OPF) for detecting anomalous internal attacks within 6LoWPAN. The signature-based intrusion detection engine that is based on supervised OPF is responsible for detecting external attacks that may occur from the Internet side. Hussein [62] integrated snort (signature-based) and algorithms such as K-means and Naive Bayes Networks (anomaly-based detection) to reduce false alarms and improve the detection rate. It is shown that different algorithms have different strengths. K-means algorithms result in less false alarms, while Naive Bayes algorithms perform well in terms of classification accuracy.

2.4 Benchmarks used in IDS

The task of anomaly detection is ubiquitous as the task is to detect the unknown. Unfortunately, there are not any established benchmark datasets that can be used to study and compare the behaviour of different algorithms for anomaly detection. The following section describes the various means by which researchers benchmark their work.

2.4.1 Benchmark Dataset

Maxion et al. [114] proposed a synthetic benchmark dataset to highlight the characteristics of the anomaly detection algorithm being tested. They proved that differences in data regularities could influence and affect the performance of the anomaly detector, such as increasing the false positive rate. Maxion noted that an important aspect of the anomaly detection system is the predictability of data as the more 'regular' the data is, the easier it is to detect anomalies. Therefore, anomaly-detection systems perform poorly on data that was configured using training data with a different amount of regularity.

Classification datasets such as those available in the UCI repository [115] are used in [116] and [117] to validate their proposed algorithms. In [118], Emmott et al. proposed a systematic approach to 'downsampling' that selects some classes of a dataset as anomalies while the other classes remain complete and are treated as normal behaviour.

2.4.2 Benchmark Application

In the end, malware intrusion comes in the form of inserted code or malicious applications. Various applications and malware are used to test these intrusion detection techniques. Pashchenko et al. [32] proposed a tool that combines both synthetic benchmarks and real-world software, while Malone et al. [41] created their benchmark software to highlight the benefits of their intrusion techniques. This section will discuss the general categories of test applications used for intrusion detection.

2.4.2.1 Existing Malware

Malware dataset can be obtained from VirusTotal [119], Drebin [120] and VirusShare [121]. These sites contain binaries, code snippets and malware source code that can be

used to train and test the intrusion detection algorithms.

It is noted that most malware is not architecture agnostic; thus, the malware made for the x86 platform does not work on an ARM-based platform and vice versa. In addition, an unsupported OS can affect the execution of a malware. Malware such as Rootkit, (see Section 2.1), is typically OS-specific. Therefore, a Rootkit malware developed for a Linux OS does not work on a platform without an OS, or on a different OS. For example, a Windows-based malware does not work on a Linux system. Though most malware can only be obtained in binary forms and can only be executed on the platform it was designed for, Rootkit malware tends to be available in source code form as it requires the malware to be compiled and installed onto the specific OS and platform.

2.4.2.2 Existing Benchmark Suite

The works [41], [46], and [122] use established benchmark application suites to represent their benign application sets, i.e. the CHStone [3], EEMBC [123], and MiBench [124] sets. These suites are typically written in simple C code and cover a wide range of application fields like media processing, digital architecture, system security, etc. In addition, input test vectors that allow for consistent testing across experiments are included with these benchmark applications.

2.4.2.3 Generating Applications to Test Anomaly Detection

During the training phase, in order to improve the quality of the detection models, more benchmark applications are required for training. Hence, authors in [88] have adopted a code stitching approach to generate additional anomalous applications for training and testing purposes. This is accomplished by combining selected benchmark applications to form a new program, where each benchmark is treated as a

separate function call. The sequence of such function calls in this new program is subsequently randomised during each run to further generate new sets of unique test programs. In addition, since the testing program is combined with different function calls and randomly mixed up during the runtime of the embedded system, the testing methodology could be used to verify the performance of the proposed system in the scenarios that have dynamic variance such as different program flow and interrupt inputs.

Another set of modified benchmark suites that tests the accuracy of anomaly detection works by inserting selected functions into a benchmark application. This is to demonstrate the sensitivity of the anomaly detection algorithm in picking up small changes in the benchmark application that is being trained.

Hamandi et al. [125] developed an Android-based malware that exploits SMS vulnerability. This was to illustrate how malware that exploits this vulnerability can go undetected by commercial anti-virus software and even Google's Play Store itself, which checks for malware.

2.4.2.4 Code Snippets for Specific Malicious Behaviour

A code snippet is a small region of reusable source code or machine code that performs a specific operation. In the context of malicious behaviour, these snippets can range from a simple buffer overflow to the more advanced ROP.

A typical code snippet comes in the form of shell-codes [126], [127]. These shell-codes do not come in the form of executables or high-level source codes. Instead, they are typically written in assembly language and then translated to ASCII codes. Subsequently, these ASCII codes are loaded onto the memory and can be executed immediately by pointing the processor's program counter to the starting address of these ASCII codes. These shell-codes are typically small, and are difficult to detect

as they are injected during runtime. A variation of this type of attack is ROP that uses existing assembly instructions in the application to form a piece of code that is malicious without the need to inject them.

The RIPE [128] test suite is a benchmark consisting of 850 distinct exploits, including code injection, return-to-libc and ROP attacks. RIPE is designed to measure the coverage provided by buffer overflow defences. These attacks are highlighted by building a small program code that exploits control-flow vulnerabilities. Techniques such as Address Space Layout Randomisation (ASLR) have no impact on RIPE. However, techniques like Data Execution Prevention (DEP) can be used to mitigate this form of attack.

The Bugbench [129] benchmark consists of several buggy programs, including Bc-1.06, Cvs-1.11.4, Gzip 1.2.4, Htpd1, Man-1.5h1, Mysql1, Mysql2, Mysql3, Ncompress, Squid-2.3, and Polymorph-0.4.0. Some programs, i.e. Mysql1, Mysql2, and Mysql3 have data race bugs, and Cvs programs have double-free bugs. In addition, Bc, Man, Gzip, Ncompress and Polymorph have buffer overflow exploits.

SAMATE Reference Dataset (SRD) [130] is another benchmark that can be used to provide security tool developers with a set of known security exploits. It is comprised of test cases that illustrate the various vulnerabilities catalogued in the Common Weakness Enumeration (CWE).

2.5 Summary

Cybersecurity has garnered much attention in the past few years in the face of ever-rising hacks and security threats. The rapid increase in the number of connected electronic devices that is used in our everyday lives, from smartphones to vehicles, kitchen appliances, and television sets, has encouraged researchers to review secu-

rity systems and propose techniques to detect intrusions or anomalies at the earliest opportunity. However, most of these devices typically run on low-cost and low-complexity embedded systems that are limited in terms of their computational power. This necessitates lightweight, yet robust, intrusion detection systems that are specifically tailored for such resource-constrained systems.

In this chapter, an extensive and comprehensive literature review on existing intrusion detection systems and their core components was performed. Existing approaches were described and categorised based on the techniques employed and their responsiveness. As discussed in Section 2.3.1, network-based intrusion detection systems primarily monitor any anomalous behaviour in the network traffic that may be a by-product of the malicious software attempting to perform illegal network transactions. On the other hand, a HIDS provides the flexibility for future-proofing as well as a comprehensive method to achieve better detection accuracy of malicious behaviour on the host device itself. Host-based intrusion detection systems can also be complemented with network-based intrusion detection systems for an even more comprehensive system.

However, the existing methodologies for HIDS are typically complex [131], [132] and not suitable for implementation on resource-constrained devices. Moreover, most intrusion detection systems do not combine multiple features such as power and SYSCALL, thereby ignoring the additional information that can be obtained by combining these features for intrusion detection. Hence, there is a need for developing a lightweight intrusion detection framework for effective detection and classification of malicious software. Such a framework can not only reduce the overheads of monitoring the system for anomalies but also maintain high detection accuracy by relying on multiple features instead of a single feature. It was evident from the survey of the existing work that a significant amount of research is needed in order to generate a comprehensive framework for an intrusion detection system that leverages on

the multiple metrics to manage resource overheads, improve detection accuracy and provide a holistic view of the attack.

There have been several existing works that rely on monitoring energy as a signature [99], [100], [101] for anomaly detection. However, these techniques that use energy or power measurement have been limited to obtaining a coarse-grain energy budget of processors or platforms [39]. Such an approach may not be sensitive enough for reliable anomaly detection since multiple applications, whether benign or malicious, can have similar energy consumption and hence be challenging to differentiate. In addition, these methods do not support runtime monitoring and can only flag up an anomaly after it has finished execution, by which time it may be too late. Additionally, existing approaches [133] and [134] require the unknown application to be identified so that the corresponding power model can be used. Chapter 3 presents a novel lightweight method of using runtime power consumption to detect anomalous behaviour during application execution time. The proposed technique relies on a single model to calculate a δ metric to decide if the application is benign or anomalous.

The next level of intrusion detection can be achieved through malware identification using SYSCALLs. Existing works in this area have used the underlying application trace information during runtime, albeit in a limited way [78], [85], [135]. Identifying the malware using the SYSCALL signature can be demanding, especially in resource-constraint embedded systems, and hence can be untenable due to the vast number of malware that needs to be identified. Moreover, none of these methods are capable of early detection. Chapter 4 proposes a low complexity technique of extracting a small number of SYSCALL signatures that can be used to represent a set of malware for not only detecting but also classifying malware at runtime. The proposed method reduces the storage requirements as well as the complexity of the detection and classification algorithms, and the resulting overheads on the platform, without sacrificing the detection and classification accuracy.

Hardware performance counters (HPCs) are on-chip tools that are very sensitive to any modifications in the application and operating system. This feature is an essential tool for intrusion detection that has been used extensively in several existing works [86], [136], [137]. However, improper use of HPCs can lead to either reduced accuracy if used too sparingly or a significantly high overhead if used too frequently. Existing techniques also select a single HPC to represent all the applications, which might lead to reduced true positive rates or increased false alarms since the various HPCs exhibit different levels of sensitivity in different applications. Existing techniques also typically use an accumulated HPC value at the end of an application execution to categorize it as a benign or a malicious application. This may be too late and hence undesirable in certain scenarios, such as when the malicious application has been allowed to run its full course. Hence, there is a need for a lightweight technique to leverage HPCs for intrusion detection that attains a balance between low overheads and high detection accuracy. Chapter 5 proposes such a technique that selects appropriate HPCs for a group of SYSCALLs and only uses them when these are invoked during application execution instead of the entire application runtime. The proposed technique has been shown to achieve high Rootkit detection rate, even during application runtime, without imposing a significant overhead on the system.

The frequent appearance of new malware attacks [18], [19], [138] raises a challenging problem given the nature of these discrete programs. Current semantic-based malware detection methods [72] and [139] rely extensively on finding frequencies of specific SYSCALL patterns in unknown programs. Such methodologies may lead to higher false negatives, as any malware having lower deviations in SYSCALL access may easily bypass such schemes. However, malware detection methods using HPCs [122], [140], [141] give rise to high false positives as they are susceptible to tiny deviations. Chapter 6 proposes a new metric called the LAMBDA, Lightweight Assessment of Malware for Embedded Architectures, denoted by λ . LAMBDA defines

a virtual boundary that separates the benign programs, which are allowed to run on a given platform, from malware that needs to be removed quickly. The metric λ is computed using carefully chosen features, which are tuples of high-level native programs (called indicators) representing OS resources, along with low-level Hardware Performance Counter events (called observers). It applies statistical hypothesis testing to distinguish between benign programs and malware accurately. The proposed technique has been shown to achieve high malware detection rates, even during application runtime, while having low rates of false positives and false negatives.

Existing frameworks have only considered a single feature [7], [142], [143] in their techniques. However, the use of multiple metrics for intrusion detection has been a coveted technique. A few works like [144] and [145] have attempted to use multiple metrics, in a limited way, to tackle a single problem, be it that of malware, Rootkits or faults. However, these techniques suffer from increased system overheads and lack a comprehensive system analysis that can be relied upon for taking appropriate remedial steps. A methodology to integrate multiple metrics intelligently and systematically that provides high detection accuracy while reducing the overheads and complexity imposed on the system is an important, yet missing aspect in existing intrusion detection systems. Finally, the implications of integrating such a framework into an existing operating system while incurring minimal overheads have not been thoroughly examined in the existing literature. Chapter 7 discusses these problems extensively and proposes a multi-modal intrusion detection framework that not only intelligently and systematically combines the proposed techniques in this thesis for a lightweight, yet highly accurate, intrusion detection system but also presents ways of integrating them into an existing operating system without compromising its integrity.

The next chapter proposes a novel lightweight technique that relies on a single model using runtime power consumption to detect anomalous behaviour during application execution.

Power Trace-based Real-time Anomaly Detection

3.1 Introduction

Malicious software, or malware, in short, are rapidly evolving with every iteration to become more advanced, and clandestine [146] as discussed in Chapter 1. The method of distinguishing malicious applications from genuine ones is known as anomaly detection. This is a challenging task since sophisticated malware frequently exhibits behaviours close to that of an unadulterated version of the application. In the past, energy and power monitoring has, frequently and successfully, been used for side channel analysis [91] [92] [94] [95] [96] [97] [98] to discreetly obtain information about the application currently being executed on a platform. However, anomaly detection techniques that use energy/power measurement have been limited to obtaining coarse-grain energy budget of processor or platform [39] [99] [101] [147]. However, the existing coarse-grain energy budget based techniques do not lend well in distinguishing between a maligned version of an application from its unadulterated version due to the almost similar power consumption [93].

In addition, energy consumption based detection techniques are reactive rather than proactive. This implies that maligned application can only be detected after the

application has finished execution and not during its execution. Such schemes might not be effective in a realistic scenario where a more proactive runtime technique is preferred to detect such misaligned applications at the earliest during runtime so that appropriate steps can be employed to minimize the consequences.

Recently, many processing platforms include sensitive power measuring hardware support and tools [1] [103] to aid the development of energy-aware applications. In this chapter, the proposed technique leverages on these power monitoring hardware capabilities and tools to generate the power trace of the unadulterated version of an application. This power trace is used to verify the correctness of the application currently being executed through periodic comparisons with the power trace model of the application stored in the database at runtime. The main advantage of our approach over these works is faster and computationally lightweight method in detecting the anomaly while maintaining a high detection rate of anomalous events. In addition, it will also be shown that using the power trace feature, not only two entirely different applications can be distinguished, but also applications that are functionally equivalent to each other but with slight variations in their implementation. Unlike existing energy-based approaches, our approach does not monitor overall energy consumption, which is a coarse-grain approach. Instead, it strives to find the power trace of each application, which can be used as a model to establish normal behaviour.

In the rest of this chapter, Section 3.2 introduces the proposed power trace based anomaly detection methodology for IDS on an embedded system. This is followed by a discussion of the proposed profile matching using correlation analysis to generate the power trace models of the applications. Section 3.2.3 describes correlation analysis being used at runtime to detect if the executing application deviates from the model. The results in Section 3.4 highlight the pros and cons of the proposed technique. Next, an extended detection strategy is proposed to identify kernel-based anomalies, followed by a summary of this chapter.

3.2 Proposed Power Trace Anomaly Detection

In this section, the proposed approach of using power trace of applications to generate models that can be used during runtime to differentiate anomalous or malicious applications from their legitimate or unadulterated version is described. As mentioned earlier, the process of differentiating such malicious applications from their legitimate counterparts is referred to as ‘anomaly detection’ in short.

Figure 3.1 illustrates the proposed two-stage design flow. The first stage is an offline process that obtains the application’s power trace and use it to generate a model of the legitimate unadulterated version of the application. In the second stage, at runtime, the power trace of the application is observed during its execution and compared against the generated model to detect any anomalies.

Figure 3.1: Overview of the proposed technique

3.2.1 Correlation Analysis

Correlation is a widely used metric for comparing two time-series, ' x ' and ' y ', where each series consists of ' m ' number of sample points. Correlation analysis was used during both the offline model generation step in Stage 1 (shown in Figure 3.1) as well as during runtime to differentiate between benign and malignant applications, in Stage 2 (shown in Figure 3.1). One type of correlation is cross-correlation, i.e. the correlation ($Corr$) between matching sample points, ' x_i ' and ' y_i ', for two time-series, ' x ' and ' y ' respectively, where the variable ' i ' is used to denote the ' i_{th} ' sample in the two time-series. This method (as shown in Eq. 3.1) identifies the degree of similarity between two time-series [148]. The normalized correlation coefficient ranges from -1 (diverging profile) to 0 (independent profile) and 1 (identical profile).

$$Corr[x, y] = \frac{m \sum x_i y_i - \sum x_i \sum y_i}{\sqrt{m \sum (x_i)^2 - (\sum x_i)^2} \sqrt{m \sum (y_i)^2 - (\sum y_i)^2}} \quad (\text{Eq. 3.1})$$

For example, multiple runs of the same application are expected to move mostly in tandem and therefore, have a high (close to 1) correlation coefficient score. However, two different applications should have a low correlation coefficient score allowing them to be differentiated. On the other hand, two versions of a single application (detailed in Section 3.3.2) that is functionally identical yet implemented differently shows low correlation coefficient score between their power profiles. This is analogous to differentiating a malignant version from a non-maligned (unadulterated) version, even though it functions similarly.

3.2.2 Offline Power Model Generation

The current and corresponding voltage measurements (at $100\mu s$ intervals) of the running application was obtained to calculate the instantaneous power. The program counter is also obtained from the system to determine the start of the application.

Figure 3.2 shows the offline steps of our approach in detail. Each application is executed ' k ' times (where $k > 1$), while capturing ' m ' power samples during each execution. Eq. 3.2 illustrates how the individual power traces, ' a^k ' from each run is stored where ' a_m^k ' represents the ' m_{th} ' power sample in the ' k_{th} ' run of the application. This ensures consistency and reliability of the generated model, and to obtain the threshold value for normalizing the model.

Figure 3.2: Offline Model Generation

$$\begin{bmatrix} a^1 \\ a^2 \\ \vdots \\ a^k \end{bmatrix} = \begin{bmatrix} a_1^1 & a_2^1 & \cdots & a_m^1 \\ a_1^2 & a_2^2 & \cdots & a_m^2 \\ \vdots & \vdots & \ddots & \vdots \\ a_1^k & a_2^k & \cdots & a_m^k \end{bmatrix} \quad (\text{Eq. 3.2})$$

Subsequently, the power values at the ' m_{th} ' sample point are averaged across ' k ' number of executions to generate a 'representative' power value, ' $\bar{a}_m$ ', where ' m '

denotes the ' m_{th} ' sample point (as seen in Eq. 3.3).

$$\bar{a}_m = \frac{1}{k} \sum_{i=1}^k a_m^i \quad (\text{Eq. 3.3})$$

The ' $\bar{a}_m$ ' values represent the averaged power values at the ' m_{th} ' sample point. These values are stored as a model, ' $\bar{A}$ ', shown in Eq. 3.4 to represent the application during runtime.

$$\bar{A} = [\bar{a}_1 \quad \bar{a}_2 \quad \cdots \quad \bar{a}_m] \quad (\text{Eq. 3.4})$$

A correlation analysis using Eq. 3.1 is performed to verify that the power model generated is indeed representative of the application. Verification of the power model is performed by calculating the correlation coefficient, ' $Corr$ ', between application model represented as ' $\bar{A}$ ' and its multiple runs, ' a^k ', seen in Eq. 3.5. The calculated correlation coefficients are stored in an array denoted as ' T^k ' that has the column size, ' k ', which represents the total number of runs.

$$T^k = \begin{bmatrix} Corr[\bar{A}, a^1] \\ Corr[\bar{A}, a^2] \\ \vdots \\ Corr[\bar{A}, a^k] \end{bmatrix} \quad (\text{Eq. 3.5})$$

The minimum value, ' τ ', is selected from this array to represent the worst case threshold for the corresponding application. Eq. 3.6 illustrates how the ' τ ' is obtained.

$$\tau = \min(T^k) = \min \begin{bmatrix} Corr[\bar{A}, a^1] \\ Corr[\bar{A}, a^2] \\ \vdots \\ Corr[\bar{A}, a^k] \end{bmatrix} \quad (\text{Eq. 3.6})$$

For each application, ' j ', the verified model, ' $\bar{A}_j$ ', and the corresponding threshold, ' τ_j ', are then stored in a database to be used in the next step of our methodology for runtime detection of anomalous applications.

Figure 3.3 summarises Stage 1 of the algorithm in a flowchart for generating an application's power model, ' $\bar{A}_j$ ' and the selection of the appropriate threshold value, ' τ_j '.

Figure 3.3: Flowchart for generating application's power model and threshold

3.2.2.1 Implementation of the Proposed Techniques

The application is compiled within the Simplicity Studio IDE [1] using the inbuilt GCC compiler. Subsequently, the binary is loaded onto the EFM32GG-STK3700 [149] development board. The EnergyAware Profiler built into the Simplicity Studio IDE is used to obtain the power trace of the application at runtime. The EnergyAware Profiler is configured to save the power trace at a sampling frequency of $100\mu s$. The application is executed ' k ' times (where $k > 10$) and the corresponding power traces are captured as a list of instantaneous power measurements at $100\mu s$ interval. A Python script calculates the average power value at each time interval and combines them to generate the power model for each application. Another Python script utilising the SciPy library [150] is used to calculate the correlation coefficient value between the power model and the power traces for ' k ' application runs. The lowest correlation coefficient is selected as the threshold value for the application.

3.2.3 Runtime Anomaly Detection using Power Model

Figure 3.4 shows Stage 2 of the proposed approach, which uses the offline power trace models generated in Stage 1, for runtime anomaly detection. It is noteworthy that identifying an application to be anomalous *after* its execution is over, may prove to be too late to prevent the ensuing damage if the application turns out to be malicious. To avoid such potentially disastrous scenarios, a runtime anomaly detection mechanism is proposed that aims to detect malicious applications as early as possible, during their execution. This is achieved by monitoring the runtime application at set intervals and calculate a 'running correlation coefficient' between the model stored in the database (obtained from Stage 1 described in Section 3.2.2) and the current runtime application.

Figure 3.4: Runtime Anomaly Detection

Figure 3.5 illustrates this process of calculating a running correlation coefficient. The proposed system detects the start of an unknown application, ' U ', using the program counter and debug information obtained from EnergyAware tool (described in Section 3.3.1) and samples the power consumption, ' u_m ', of the runtime application at a time interval of $100\mu s$ for a total of ' m ' number of samples. In this study, the time interval is set at $100\mu s$, as it is the minimum allowed sampling frequency in the EnergyAware tool [1]. The relatively high sampling frequency enables the proposed technique to monitor even the minor power fluctuations in the system in order to cre-

ate an accurate power profile. Eq. 3.7 illustrates how power measurement is captured at fixed time interval and stored. It then fetches the corresponding application model, ' $\bar{A}_j$ ', stored in the database created in the previous section.

$$U = [u_1 \quad u_2 \quad \cdots \quad u_m] \quad (\text{Eq. 3.7})$$

Figure 3.5: Runtime Anomaly Detection using Correlation

Next, the running correlation coefficient is calculated using the power samples observed thus far from the running application and the corresponding model of the application, as discussed in Section 3.2.3. This process is repeated periodically, every $500\mu s$. The interval of $500\mu s$ allows five power samples to be obtained from the EnergyAware tool, which is sampled at $100\mu s$. It was found through experimental results that minimum five samples were required to ensure a stable correlation metric and reduce the monitoring overhead while still achieving a high detection accuracy. For example, in Figure 3.5, the correlation metric until the $500\mu s$ is above the application's threshold value of 0.991 (that was calculated using Eq. 3.6), implying that the application is not anomalous. When the correlation metric is calculated again at the $1000\mu s$ timestamp, the correlation metric remains higher than the application's

threshold value. However, it reduces to 0.85 in the next time interval, which is below the application's threshold value.

At this stage, the time interval of calculating the correlation metric is reduced to $200\mu s$ to monitor the application more frequently. $200\mu s$ is equivalent to two power samples, which is the minimum required number of samples to calculate the correlation coefficient. Further, to reduce false alarms, the runtime application for three consecutive monitoring periods are observed (a total of $600\mu s$), before triggering the alarm. The overall strategy not only allows detection of anomalous applications at runtime but also minimizes the number of false alarms.

3.2.4 Model Aggregation

The method discussed in Section 3.2.3 requires the system to identify the running application so that the corresponding model can be extracted and used in the comparison. Identifying each application, ' A_j ', especially at runtime, introduces challenges in implementation. Thus, a single aggregated model is proposed, which can be used to represent ' j ' number of benign applications. Eq. 3.8 illustrates how each application power trace, ' $\bar{A}_j$ ' is stored where ' a_m^j ' represents the ' m_{th} ' power sample for the ' j_{th} ' application. This single model, ' B ', is used to represent the ' j ' number of benign applications. For applications that finish faster than others, the model is appended with 0s in order to have the same number of samples in all applications.

$$B = \begin{bmatrix} \bar{A}_1 \\ \bar{A}_2 \\ \vdots \\ \bar{A}_3 \\ \bar{A}_j \end{bmatrix} = \begin{bmatrix} a_1^1 & a_2^1 & a_3^1 & \cdots & 0 & 0 \\ a_1^2 & a_2^2 & a_3^2 & \cdots & a_{m-1}^2 & a_m^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_1^3 & a_2^3 & 0 & \cdots & 0 & 0 \\ a_1^j & a_2^j & a_3^j & \cdots & a_{m-1}^j & a_m^j \end{bmatrix} \quad (\text{Eq. 3.8})$$

In addition to the application model, a single threshold is required to differentiate between similar application. This selected threshold value, ' T ', is the minimum value from the list of application's threshold value, ' τ_j ', as shown in Eq. 3.9. In this equation, ' j ' represents the total number of benign applications.

$$T = \min \begin{bmatrix} \tau_1 \\ \tau_2 \\ \vdots \\ \tau_j \end{bmatrix} \quad (\text{Eq. 3.9})$$

When an unknown application, ' U ', is executed, the system starts sampling the power at $100\mu s$ interval and records them as shown in Eq. 3.7. After obtaining the first five samples, the system subsequently calculates the correlation coefficients, as shown in Eq. 3.10 using the aggregated model seen in Eq. 3.8.

$$\rho = \text{Corr}[U, B] = \begin{bmatrix} \rho_1 \\ \rho_2 \\ \vdots \\ \rho_j \end{bmatrix} \quad (\text{Eq. 3.10})$$

These correlation coefficients are normalised (seen in Eq. 3.11) by using the selected threshold value, ' T ', obtained from Eq. 3.9, and rounding down the result.

$$\tilde{\rho}_i = \begin{bmatrix} \left\lfloor \frac{|\rho_1|}{T} \right\rfloor \\ \left\lfloor \frac{|\rho_2|}{T} \right\rfloor \\ \vdots \\ \left\lfloor \frac{|\rho_i|}{T} \right\rfloor \end{bmatrix} \quad (\text{Eq. 3.11})$$

Figure 3.6 shows the extension to the proposed Stage 1 approach, where only a single power model and threshold value is used in the anomaly detection.

Figure 3.6: Generating a single power model and selecting a threshold value

A single metric ‘ δ ’ is proposed to decide whether an application is within the trained set. ‘ δ ’ is calculated as the sum of all the elements in the matrix ‘ $\tilde{\rho}_i$ ’.

$$(\delta = 0 \rightarrow \text{anomaly}) \wedge (\delta \geq 1 \rightarrow \text{benign}) \quad (\text{Eq. 3.12})$$

As shown in Eq. 3.12, if ‘ δ ’ is 0 for 3 consecutive intervals, then the application is categorised as showing anomalous behaviour. However, if ‘ δ ’ is equal to 1, then the application is within the trained set and can be identified. ‘ δ ’ greater than 1, means that the application is similar to more than one benign application’s power trace.

In the next section, the proposed power trace based anomaly detection techniques is applied to differentiate not only two entirely disparate applications, but also show the effectiveness of our approach to differentiate two applications that are functionally identical but differ in their implementation.

3.3 Experimental Setup

In this section, the experimental platform used in our work is first presented. Subsequently, the proposed methodology for detecting malicious versions of an application by observing their runtime power consumption are evaluated at set intervals by comparing the runtime power trace to the power trace model generated offline. As stated earlier, the consistency of applications that need to be executed in an embedded system makes it feasible to store these models for all potential applications.

3.3.1 Platform

A development platform *EFM32GG-STK3700* [149] from Silicon Labs is used for all the experiments performed in this work. This platform uses Silicon Labs' *EFM32 Giant Gecko* ARM® Cortex®-M3 based 32-bit microcontroller that contains a microprocessor running at 48 MHz, 128 kB of RAM and 1024 kB of flash memory. The platform runs on uC/OS-III real-time operating system [151]. An important feature of this platform is the Advanced Energy Monitoring (AEM) system that allows accurate measurement of power consumption of an application in real-time with the help of current (I) and voltage (V) sensors embedded along microcontroller's power rail.

This development platform, shown in Figure 3.7, also includes a software tool called *EnergyAware* Profiler, shown in Figure 3.8, that graphically displays real-time current and voltage information that can be saved for offline processing and analysis. In addition, this software tool also allows users to correlate between the power consumption and the application code segment currently being executed on the platform. In our setup, the development board is connected to a computer running the *EnergyAware* profiler software tool via the USB-JTAG. The experiments were performed on a desktop computer running on *Intel Xeon E5-1650 @ 3.5Ghz* with 4-core CPU and 16GB RAM with *Windows 7 Professional* OS.

Figure 3.7: EFM32's Development Platform [1]

Figure 3.8: AEM tool used to obtain the application's power trace

3.3.2 Benchmark

As will be seen in the subsequent subsections in the paper, the power trace-based method can be used to distinguish between two entirely different applications. On the other hand, it is rather difficult to differentiate between a legitimate version of an application and another version that has been minimally, yet dangerously, modified by appending unknown code to it [51]. To evaluate our methodology on such versions of the application, *two versions* of an application were collected that are functionally identical but implemented differently. Multiple implementations of selected security related benchmark applications were selected for the evaluation stage since malware typically target these security applications to compromise them [152].

3.3.2.1 Input Data

Standard input data provided by the benchmark suite (as shown in Table 3.1) was used for evaluation. The input data for applications in benchmark suites are typically selected based on the assumption that it is representative of the realistic input to the application during their execution time [153]. Future experiments will include a wider variety of input data.

In order to verify the functional equivalence of the two versions of an application, the standard input data from one of the implementations (benchmark suite) was selected to execute both versions and verify that the output is identical. For example, as shown in the second row of Table 3.1, two versions of AES was used, one from CHStone [3] and the other from kokke [154], while using the input data from CHStone for both versions. The same input data is also used while generating the offline power trace model for both application versions. Table 3.1 lists the *ten benchmark applications* that have been selected to illustrate the proposed methodology.

Table 3.1: Benchmark Applications

#	Benchmark Application	Description	Version 1	Version 2	Test Input Data
1	Advanced encryption standard (AES) encryption and decryption	Rijndael was selected as the National Institute of Standards and Technologies Advanced Encryption Standard (AES). 128-bit keys and blocks.	CHStone [3]	kokke [154]	CHStone
2	Adaptive differential pulse code modulation (ADPCM) decoder and encoder	ADPCM is a technique for converting analogue information to binary information.	CHStone	MiBench [124]	CHStone
3	Blowfish encryption and decryption	Blowfish is a symmetric block cipher with a variable length key.	Bradconte [155]	Koc [156]	Bradconte
4	Cyclic Redundancy Check (CRC) 32bit	CRC checks are often used to detect errors in data transmission.	Barr (fast) [157]	Barr (slow)	Barr
5	Secure hash algorithm (SHA-256)	Used in the secure exchange of cryptographic keys and for generating digital signatures.	Bradconte	CHStone	Bradconte

To evaluate the proposed methodology, a total of 13 benchmark applications were used. Among these 13 applications, five pairs of *functionally equivalent* applications were selected to show that the proposed methodology is ‘sensitive’ enough to differentiate even between two very closely related applications. Apart from these functionally equivalent applications, two entirely different applications were used to confirm that the power trace based anomaly detection approach can certainly differentiate between entirely different applications.

In addition, a unique handwritten application, called PRINT-LOOP, was used to evaluate the performance of the proposed methodology in the presence of a kernel-based malware, commonly known as a rootkit. In this handwritten PRINT-LOOP application, the task prints out a string every $100ms$ while accessing common SYSCALLs

Table 3.2: Benchmark for training and testing

Training	Testing	
ADPCM V1	ADPCM V1	ADPCM V2
AES V1	AES V1	AES V2
CRC-32 V1	CRC-32 V1	CRC-32 V2
SHA-256 V1	SHA-256 V1	SHA-256 V2
BLOWFISH V1	BLOWFISH V1	BLOWFISH V2
GSM	GSM	DFADD
Print-loop	Print-loop	Print-loop + Synthetic Rootkit V1
-	-	Print-loop + Synthetic Rootkit V2
-	-	Print-loop + Synthetic Rootkit V3

that are targeted by the rootkit. The operating system on the evaluation platform was infected with 3 synthetic rootkits while testing the handwritten PRINT-LOOP application to showcase that the technique can also detect this type of attack. These synthetic rootkit attempts to hijack common SYSCALLs used by the PRINT-LOOP application.

As shown in Table 3.2, the 13 applications are divided into testing and training sets. 7 applications have been selected for training, including the handwritten PRINT-LOOP application. 9 applications were selected for the testing phase, which includes executing the handwritten application with the OS infected with 3 different synthetic Rootkit. In addition to the remaining 7 applications in the testing set, all applications used during the training phase (seen in column 2 in Table 3.2) were included to confirm that the power trace based technique does not trigger an alarm if a known application is executed during runtime.

3.4 Test Results

This section presents the experimental results of the proposed anomaly detection system based on power trace. The accuracy of the proposed method has been evaluated based on whether it can be deployed to effectively differentiate at runtime between the unknown application and the trained model of benign applications. In addi-

tion, our method demonstrates the ability to detect an anomalous application *during* its execution and not after the execution is over, as done in most existing work.

3.4.1 Power Trace Models

Figure 3.9 shows the power trace from different runs of an application, where the X-axis represents the time stamps, and the Y-axis represents the power consumption at different time instances. The results for five runs is shown in this figure. The dotted line in this graph denotes the average power consumption across the five runs of this application. As discussed earlier in Section 3.2.2, the average power consumption profile is stored in the database as a model to represent this application. Using this technique, the sub-figures in Figure 3.10 show the power trace model for the two versions of an application. In order to ensure that these models are indeed representative of the application, the application was executed ten times and their instantaneous power values were recorded for each run. Subsequently, as shown in Eq. 3.3, these values were averaged across the runs to generate the power model of the application, as illustrated in Figure 3.9. The correlation of power trace across runs was also calculated, as illustrated in Table 3.3, to ensure that the model accurately represents the application, as confirmed by a high(close to 1) correlation coefficient. The verified model is then stored in a database to be used for anomaly detection at runtime. As mentioned in Section 3.3.2, both versions of the same benchmark application use the same input data for this process.

It can be seen from Figure 3.10 that most of the power traces can also be visually differentiated from each other. Comparing the power trace model of one application (e.g. ADPCM version 2 in Figure 3.10.d) to a different application (e.g. Blowfish in Figure 3.10.a, SHA-256 in Figure 3.10.e, etc.), it can be seen that the proposed technique can differentiate the power trace models of distinct applications. On the other

Figure 3.9: Averaging Results

Table 3.3: Obtaining the threshold for each application

#	Model	Application Runs					
		τ_{R1}	τ_{R2}	τ_{R3}	τ_{R4}	τ_{R5}	$\min(\tau)$
1	ADPCM V1	0.999	0.995	0.998	0.999	0.995	0.995
2	ADPCM V2	0.997	0.986	0.998	0.996	0.968	0.968
3	AES V1	0.997	0.998	0.997	0.997	0.991	0.991
4	AES V2	0.987	0.980	0.996	0.994	0.999	0.980
5	BLOWFISH V1	0.997	0.996	0.998	0.996	0.998	0.996
6	BLOWFISH V2	0.999	0.999	0.991	0.999	0.988	0.988
7	CRC32 V1	0.998	0.999	0.999	0.999	0.999	0.998
8	CRC32 V2	0.995	0.999	0.999	0.995	0.999	0.995
9	SHA-256 V1	0.999	0.998	0.999	0.998	0.998	0.998
10	SHA-256 V2	0.997	0.999	0.999	0.999	0.998	0.997
11	GSM	0.990	0.998	0.999	0.976	0.985	0.976
12	DFADD	0.989	0.982	0.998	0.959	0.995	0.959
13	MIPS	0.998	0.999	0.997	0.988	0.985	0.985
14	PRINT-LOOP	0.965	0.986	0.992	0.985	0.969	0.965

hand, differentiating between the two functionally equivalent versions of an application can be challenging for specific applications. For example, while the two versions of AES application exhibit vastly different power traces, other application such as SHA-256 and CRC-32, etc. have similar power trace but with vastly different execution times. For example, SHA-256 version 1, takes an additional 10ms to complete as compared to SHA-256 version 2. Similar differences in timelines can also be seen for CRC-32, ADPCM and Blowfish applications.

Figure 3.10: Power Profile for Functionality Equivalent Applications

3.4.2 Detection Results

In the previous subsection, it was shown that the power trace models for many applications could be visually distinguished from others. However, in order to quantify the differentiation between the power trace models of various applications, the correlation coefficients were calculated between these power trace models for an instance of a runtime application. Figure 3.11 shows the power trace for the remaining applications, while Table 3.4 shows the correlation coefficient for all the application models against the runtime applications used in our experiments. A correlation coefficient of 0.959 and above shows that they are the same application, while a correlation coefficient of

Figure 3.11: Power Profile for Other Benchmark Applications

less than 0.959 signifies a different application.

Table 3.3 shows how the lowest correlation coefficient between the model and multiple runs for each application is calculated and stored. It was found empirically to be a suitable threshold to represent the worst case that any run of the application can deviate from the model. In addition, it helps to generate a tight bound to effectively distinguish an application from other applications or even other versions of the same application.

Table 3.4: Correlation coefficient amongst all benchmark applications

	RUNTIME APPLICATION														
#	MODEL	ADPCM V1	ADPCM V2	AES V1	AES V2	BLOWFISH V1	BLOWFISH V2	CRC-32 V1	CRC-32 V2	SHA-256 V1	SHA-256 V2	GSM	DFADD	MIPS	PRINT-LOOP
1	ADPCM V1	1.00	0.26	0.58	0.30	0.67	0.66	0.44	0.37	0.42	0.42	0.28	0.69	0.44	-0.38
2	ADPCM V2	0.24	0.99	0.56	0.86	0.22	0.25	-0.46	-0.45	0.22	0.19	0.28	0.69	0.44	-0.38
3	AES V1	0.41	0.60	0.98	0.82	0.48	0.43	-0.24	-0.32	0.39	0.36	-0.04	0.38	0.62	-0.23
4	AES V2	0.28	0.78	0.79	0.99	0.33	0.32	-0.46	-0.53	0.29	0.27	-0.04	0.38	0.62	-0.23
5	BLOWFISH V1	0.69	0.22	0.65	0.33	1.00	0.69	0.40	0.41	0.55	0.55	0.36	0.68	0.56	-0.45
6	BLOWFISH V2	0.66	0.16	0.56	0.22	0.73	0.98	0.42	0.56	0.56	0.56	0.29	0.68	0.45	-0.44
7	CRC-32 V1	0.53	-0.44	-0.07	-0.52	0.38	0.38	1.00	0.49	0.30	0.31	0.27	0.44	-0.03	-0.31
8	CRC-32 V2	-0.41	-0.41	-0.15	-0.56	0.43	0.55	0.48	1.00	0.30	0.32	0.27	0.44	-0.03	-0.31
9	SHA-256 V1	0.43	0.23	0.57	0.28	0.54	0.50	0.30	0.27	1.00	0.70	0.28	0.65	0.44	-0.24
10	SHA-256 V2	0.45	0.28	0.59	0.32	0.55	0.51	0.27	0.26	0.64	0.99	0.28	0.87	0.44	-0.44
11	GSM	0.04	-0.30	0.48	0.28	0.39	0.32	0.25	0.19	0.29	0.28	0.99	0.27	0.76	-0.22
12	DFADD	0.33	0.28	0.49	-0.37	0.70	0.72	0.40	-0.12	0.67	0.60	0.28	0.99	0.44	-0.44
13	MIPS	0.65	0.29	0.86	0.68	0.56	0.45	-0.07	0.07	0.44	0.43	0.65	0.43	1.00	-0.22
14	PRINT-LOOP	-0.22	-0.25	-0.46	0.16	-0.45	-0.42	-0.26	0.36	-0.24	-0.28	-0.24	-0.48	-0.22	1.00

It can be seen from Table 3.4, that if a runtime application is compared to its model stored in the database, the correlation coefficient is close to 1.00. For example, row 3 of column 3 in Table 3.4 shows that the correlation coefficient between the runtime application ADPCM version 1 and the corresponding model of ADPCM version 1 has a correlation coefficient of 1.00, thereby confirming that the application is legitimate. Similarly, all the correlation coefficients along the diagonal of Table 3.4 show a value higher than 0.959 and in most cases 1.00, which confirms that if a legitimate application is executed during runtime, it can be verified against the stored power trace model for confirmation by calculating the correlation coefficient.

On the other hand, the correlation coefficients between different applications or even between the two versions of the same application are significantly less than 0.959 in most cases which is lower than all the application's threshold as seen in Table 3.3. In particular, applications like ADPCM version 2 and CRC-32 version 2 have vastly different power traces that lead to an extremely low correlation coefficient (in this case -0.41).

Next, the correlation coefficients of the 6 pairs of functionally equivalent application (e.g. AES version 1 and AES version 2) were observed. It can be seen that the correlation coefficients between the two versions of an application are also below the 0.959 threshold for all the applications.

Results in Table 3.5 have shown that all applications used during the training phase can be accurately detected as a benign application. In addition, all the 'functionally equivalent' applications were correctly classified as an anomaly. The untrained DFADD application was also correctly classified as an anomaly. Moreover, in the 3 instances where the benign PRINT-LOOP application was executed on a Rootkit-compromised operating system, the method detected them as anomalies. However, this method is unable to identify if the attack came from the application or the operating system.

Table 3.5: Results for trained and test applications

Test Applications	Classification
ADPCM V2	anomaly
AES V2	anomaly
CRC-32 V2	anomaly
SHA-256 V2	anomaly
BLOWFISH V2	anomaly
DFADD	anomaly
Print-loop + Rootkit V1	anomaly
Print-loop + Rootkit V2	anomaly
Print-loop + Rootkit V3	anomaly
Trained Applications	Classification
ADPCM V1	benign
AES V1	benign
CRC-32 V1	benign
SHA-256 V1	benign
BLOWFISH V1	benign
GSM	benign
Print-loop	benign

3.4.3 Response Time

Figure 3.12 illustrates the process of calculating the correlation coefficient periodically at runtime, between the power trace of the runtime application and the model stored in the database. At any time instant, the correlation coefficient is obtained by comparing the power trace of the application executed thus far and the corresponding portion of the power trace model from the database. The periodic calculation of the correlation coefficient is performed more frequently if the correlation coefficient goes below the application threshold, such that it can be monitored more closely. In order to minimize false alarms, the proposed technique continues to monitor the application for a pre-defined number of periods beyond this time (when the correlation coefficient reduces below application's threshold), before triggering the alarm, if the correlation coefficient continues to drop. In our experiments, three consecutive periods were monitored before the system triggers an alarm. Section 3.2.3 proposes the duration of each period as $500\mu s$, $250\mu s$ and $100\mu s$. Therefore, the method detects an anomaly within

1ms of deviation. More frequent testing can lead to improved response time but at the expense of resource overheads to calculate the correlation coefficient more frequently.

3.12.a: AES

3.12.b: BLOWFISH

3.12.c: CRC32

3.12.d: ADPCM

Figure 3.12: Rapid runtime power analysis

Table 3.6 shows the response time of the proposed runtime anomaly detector. The value in the 2nd column of the table represents the total time of the test application. Column 3 displays the time it took for the system to detect the anomaly, and the 4th column shows the percentage of when the system detects the running application as anomalous. For example, ADPCM version 2 was detected as an anomalous halfway through execution (54.5%). On the other hand, the AES version 1 application could only be detected near the end of its execution (77.2% of the application runtime). This happens if the execution time of the modified application being monitored is shorter than the original application that was trained and stored in the database. In other words, the malicious version of the application is a truncated version of the legitimate application. However, even in such cases, the proposed methodology can

Figure 3.13: Rootkit detected using power trace

effectively detect the presence of a malicious application, albeit near the end of execution. It is still noteworthy that, in all the cases, as shown in Table 3.6, the proposed methodology could detect a different version of the application before its completion.

Table 3.6: Response time of test applications

Test Application	Total Runtime (ms)	Time Detected (ms)	Runtime (%)
ADPCM V2	2.2	1.2	54.54
AES V1	2.2	1.7	77.27
CRC-32 V1	4.3	3.4	79.07
Print-loop + Rootkit V1	800	0.3	0.04
Print-loop + Rootkit V2	800	1.0	0.13
Print-loop + Rootkit V3	800	1.2	0.14

Figure 3.13 illustrates the proposed technique to be used to detect an anomaly when a benign application is executing on Linux OS infected by the *suterusu* Rootkit. The Rootkit is injected into the OS during the application's execution at the time interval 21. It can be seen that the method is able to detect the anomaly during runtime that could trigger the alarm for further action to be taken.

3.5 Summary

In this chapter, a lightweight power trace based technique was proposed for anomaly detection on an embedded system. As opposed to existing methods [39], [93], [101], the proposed approach relies on the power trace of the benign applications to generate a model that can differentiate between trained benign applications and other unknown, untrained or malicious applications. A one-time model generation is required that represents all the benign applications, to capture the relevant power trace information as described in this chapter.

It has been shown in Table 3.5 that power trace can be used to accurately detect anomalies at not only at the user application level but also at the OS's kernel level. The proposed methodology was used to successfully differentiate between entirely different applications as well as similar applications, using power trace as the only feature. In addition, it was also shown that the power trace method could even be used to detect threats from Rootkit that directly compromise the operating system.

Additionally, as shown in Figure 3.13, the methodology is able to detect Rootkit almost instantaneously during runtime. It has been shown that power trace can be used to detect anomalies within $1ms$ of power-trace deviation accurately. Lastly, this technique is able to detect anomaly coming from the user-space application or the OS's kernel-space. Therefore, once the power trace method detects the anomaly, it can trigger an alarm to enable more complex methods for anomaly classification.

The proposed technique was compared with an existing method in terms of response time performance. It is noteworthy, the majority of the existing work relies on lump sum energy profile. On the other hand, the proposed method relies on the runtime power profile enable early anomaly detection.

While the power trace based anomaly detection technique proposed in this chapter can be used to effectively detect an anomalous application, it is unable to classify the source of the anomaly. The next chapter presents a novel real-time malware classification technique that can be performed using a minimal set of system call signatures.

System Call Trace-based Classification of User-space Malware

4.1 Introduction

In the previous chapter, a novel anomaly detection technique using the application's power-profile was presented, that can be relied upon to detect an anomalous application during runtime. This chapter aims to not only detect but also classify the type of malicious application encountered at runtime. Existing works have proposed to use the 'system calls' made by an application during its execution as a feature to not only detect but also classify the type of malware. These techniques are either based on machine learning [158], [159] or malware signatures [160], [161]. Machine learning based techniques require meticulous selection of features that would best help classify the malware. In addition, machine learning based detector need careful training and a long time for tuning [162]. Additionally, machine learning faces false alarms that can quickly erode the user's trust in the machine-learning-based solutions [163].

On the other hand, the signature-based matching technique relies on the `short sequences` of SYSCALLs made by a program during normal executions to differentiate between a benign and malicious application [54]. The authors in [25] proposed an automated approach to extract malware behaviour by observing the SYSCALLs

in a sandbox environment. Similarities and distances between malware behaviours are computed, which allow for the classification of malware behaviours. Currently, signature-based malware detection techniques are still the most widely used method for detecting malware [164], [165]. However, the classification process uses a phylogenetic tree [25] which has the limitation of wrongly classifying a few malware behaviours. Additionally, these methods also need to extract and store unique signatures for every malware [166] for runtime detection. This puts additional pressure on the memory and computational resources of the already resource-constrained IoT devices. Moreover, these techniques depend on malware signatures that occur at any point in the malware execution, which limits the responsiveness of this technique. This chapter proposes a malware classification technique that relies on malware signatures using sequences of SYSCALLs that is less computationally intensive while maintaining a high malware detection rate. Unlike existing signature-based approaches, our approach does not monitor specific byte-sequences or strings that are unique for every malware. Instead, it strives to find common signatures across a group of malware to save on the computation and storage costs of these malware signatures and therefore is especially useful for extremely resource-constrained IoT devices. The proposed technique has also been extended to look for malware signatures at regular intervals during application execution for real-time detection and classification of malware, unlike existing approaches that typically classify malware only at the end of application execution.

In the rest of this chapter, Section 4.2 introduces the runtime SYSCALL-based malware classification methodology for IDS. Section 4.3 describes the proposed signature extraction process for malware detection. This is followed by a discussion of SYSCALL-based classification used to separate malware into their families. Section 4.4.1 details how SYSCALL signatures can be used for real-time classification of malware. Section 4.5 provides the results and highlights the pros and cons of

the approach. Finally, the proposed methodology is compared with a state-of-the-art followed by a summary for this chapter.

4.2 Runtime System Call-based Malware Classification

This section presents the proposed method for extracting a small set of malware signatures that are used to detect the presence of malware. Figure 4.1 illustrates the overall design flow of the proposed malware signature extraction technique and its subsequent usage for real-time malware classification. Our method works in two stages where the signatures are first extracted in an offline step and then subsequently used at runtime to detect and classify malware.

Figure 4.1: Overview of the proposed technique

1. **Trace Acquisition:** The *strace* tool in Linux is used to obtain the SYSCALL trace in a sandbox environment. The trace includes timestamps of each call of the ‘system calls’.
2. **Time Slicing for Real-time Analysis:** The SYSCALL trace generated in the previous step is then segmented at fixed time intervals by relying on the timestamps in the trace.
3. **Signature Extraction:** A list of system calls (ranging from two to five system calls) is extracted from the SYSCALL trace. This list is called as the ‘signature’

for the application, which can be used to uniquely identify the executable. All common signatures for each category (e.g. benign) is aggregated into a single list. This procedure is also repeated for the malware set. For malware, only those malware signatures, which are not found in the list of signatures for benign application, are retained.

4. **Signature Refinement:** The signature list is then further pruned to remove any redundant signatures that do not improve the detection accuracy. The pruned list is referred to as the Refined Signature List.
5. **Malware Detection:** Next, this refined signature list is used to verify its effectiveness in detecting malware and identify the ones that cannot be captured with this signature list.
6. **Malware Signatures:** Additional signatures are extracted from malware that were not detected using the Refined Signature list.
7. **Recursive Signature Extraction:** These new signatures are augmented with the Refined Signature list to generate the final list of signatures. Steps 5 and 6 can be repeated iteratively until the algorithm achieves the desired detection accuracy.

4.2.1 Implementation of the Proposed Techniques

Blocks #2 to #4 in Figure 4.1 are implemented using *bash shell* scripts in Linux OS. The script written for block #2 checks the timestamp and segments the SYSCALL trace accordingly. Script written for block #3 implements Algorithm 1 that is described in Section 4.3.2. The script written for block #4 checks if any two signatures can detect the same malware. If yes, then the shorter signature is selected. Block #5 implements the algorithm illustrated in Figure 4.11 described in Section 4.5.1 on a

SPARC platform. Implementation for block #6 uses the same script written for block #3, but on malware not detected in block #5. The bash shell scripts used in this chapter has been included in Appendix A.

4.3 Extracting Malware Signatures using System Call Sequences

Malware is first categorised into their respective families based on VirusShare classification [121]. The VirusShare website lists the categories for each malware sample as provided by the various commercial anti-virus tools. The malware label assigned by the most number of commercial anti-virus tools is finally taken as the selected label for the malware. For example, a malware sample with the md5 of c67268de0050dabf6f3834af2b67d8a9 is labelled as Trojan by 24 and as Virus by 15 anti-virus tools. Therefore, in our dataset, this malware is classified as Trojan. In this work, three categories of malware are selected; Trojan, Worm and Virus.

As can be seen in Figure 4.2, for each malware family, the SYSCALL trace is obtained from the malware by executing it in a sandbox environment and using the *strace* tool in Linux during execution (#1 in Figure 4.2). The SYSCALL traces will also include the timestamp of when each SYSCALL is executed. The traces are then processed to extract timing information to enable real-time detection. They are also analysed to extract *signatures*, which are essentially unique sequences of SYSCALLs that can be used to detect if a particular executable is a malware. A minimal set of signatures are selected that can be used to classify a malware family. In a similar fashion, signatures from benign executables were also obtained (#3 in Figure 4.2). Finally, both the benign and malware signature list are analysed to select relevant signatures that can be used to detect malware while eliminating the false positive (#4 to #7 in Figure 4.2). The description of the various steps involved in the system is as

follows:

Figure 4.2: Overview of Enhanced Signature Extraction Process

4.3.1 System Call Trace Acquisition

This section explains block #1 in Figure 4.2. The executables are run in a Virtual Machine Linux sandbox environment, as explained later in Section 4.5.2. This sandbox ensures that the SYSCALL trace can be obtained in a controlled and isolated environment. The execution step generates the real-time SYSCALL traces of the executable as seen in Eq. 4.1 where α and n represents each SYSCALL in the trace and the total number of SYSCALL, respectively.

$$Executable_{trace} = [\alpha_1 \ \alpha_2 \ \alpha_3 \ \alpha_4 \ \cdots \ \alpha_n] \quad (\text{Eq. 4.1})$$

Similar to [167], in this work common applications such as Firefox and various Linux commands (ls, pip, etc.) are used as benign applications. Similar to malware, the benign applications' executables are also used to obtain their SYSCALL traces by running them on the same sandbox environment. The SYSCALL trace was generated once and stored for all the malware and benign executables. Table 4.1 shows an excerpt of the SYSCALL trace before (Column 1) and after (Column 2) processing. This involves removing both the inputs and outputs. The following trace is obtained:

$$Executable_{trace} = [brk, access, access, open, fstat64, mmap2, close].$$

Table 4.1: Processing System Call Trace

RAW TRACE	PROCESSED TRACE
brk(NULL)=0x892e000	brk
access("etc/ld.so.nohwcap",F OK) = -1 ENOENT	access
access("etc/ld.so.preload",R OK) = -1 ENOENT	access
open("etc/ld.so.cache",O_RDONLY O_CLOEXEC) = 3	open
fstat64(3, fst mode=S_IFREG 0644, st_size=82832, ...g) = 0	fstat64
mmap2(NULL, 82832, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf7719000	mmap2
close(3)	close

Both the malware and benign traces are then passed on to the next step to extract their respective signatures.

Figure 4.3: Signature Extraction

4.3.2 Malware Signature Extraction

This section corresponds to block #3 in Figure 4.2. As shown in Figure 4.3, a signature is generated by extracting a sequence of two to five SYSCALLs for each SYSCALL, till the end of the trace. The minimum length of the SYSCALL list is two since using a signature with a single SYSCALL will lead to a significant number of false positives. Moreover, as suggested by the authors in [139], a maximum of five SYSCALLs sequence is obtained, since signatures longer than five SYSCALLs do not improve detection accuracy. Here, *a list of such sequential system calls is defined as a 'signature'*. All duplicate signatures are removed.


```

for every syscall in trace do
    add( $SL$ , [ $Si$ ,  $Si + 1$ ]);
    add( $SL$ , [ $Si$ ,  $Si + 1$ ,  $Si + 2$ ]);
    add( $SL$ , [ $Si$ ,  $Si + 1$ ,  $Si + 2$ ,  $Si + 3$ ]);
    add( $SL$ , [ $Si$ ,  $Si + 1$ ,  $Si + 2$ ,  $Si + 3$ ,  $Si + 4$ ]);
end
for every  $i$  in signature do
    for every  $j$  in trace do
        if  $SL_i == SL_j$  then
            remove( $SL$ ,  $SL[j]$ );
        end
    next  $j$ ;
    end
next  $i$ ;
end

```

Algorithm 1: Signature Extraction for Malware and Benign Application

This set of signature (represented as Eq. 4.2) can be subsequently used to identify a particular malware or benign executable, where α represents a SYSCALL and n denotes the total number of SYSCALLs in the executable's trace. This procedure is repeated for each malware and benign traces. Algorithm 1 shows pseudocode for this procedure.

$Executable_{signature} =$

$$\begin{bmatrix} [\alpha_1\alpha_2] & [\alpha_1\alpha_2\alpha_3] & [\alpha_1\alpha_2\alpha_3\alpha_4] & [\alpha_2\alpha_3] & [\alpha_2\alpha_3\alpha_4] & \cdots & [\alpha_{n-2}\alpha_{n-1}\alpha_n] & [\alpha_{n-1}\alpha_n] \end{bmatrix}$$

(Eq. 4.2)

Signatures for each malware is extracted and then compiled into a set of malware signatures to represent all the malware, as shown in Eq. 4.3, where j and i represent the total number of malware and benign applications, respectively. This procedure is repeated for benign executables, as shown in Eq. 4.4. The Union operator, $\cup$, ensures all the signatures are collated to form a database of malware and benign signatures respectively.

$$Malware_{signature} = M_1 \cup M_2 \cup M_3 \cup M_4 \cup \cdots \cup M_j \quad (\text{Eq. 4.3})$$

$$Benign_{signature} = B_1 \cup B_2 \cup B_3 \cup B_4 \cup \cdots \cup B_i \quad (\text{Eq. 4.4})$$

Finally, as shown in Figure 4.4, the set of benign signatures and the set of malware signatures are cross-referenced. Any signature from the malware set that is

found in the benign set is removed as a potential signature to identify malware. This is to eliminate any false positive by ensuring that any signatures used for malware detection cannot be found in benign executables.

Figure 4.4: Refining Malware Signature Extraction

At this juncture, a set of signatures has been obtained that can accurately detect all the malware used in the training phase. This is represented as the shaded area in Figure 4.4. The next subsection details how to reduce this set of signature while maintaining its detection accuracy.

4.3.3 Malware Signature Refinement

As shown in block #4 in Figure 4.2, the refined signature list can be further reduced while still maintaining the same detection rate. This is mainly due to these signatures detecting the same set of malware. Thus removing either one of these signatures yield the same detection rate. The signatures are sorted in descending order of their number of occurrences in the set of malware signatures. The first signature is selected starting from the most occurring one, and the subsequent signatures are incorporated into the list if it improves the detection accuracy.

4.3.4 Recursive Malware Signature Extraction

Figure 4.5: Enhanced Malware Signature Extraction

This section describes block #7 in Figure 4.2. In cases where the malware signatures fail to detect certain malware, these missed malware can be looped back into the system to obtain additional SYSCALL signatures, as shown in Figure 4.5. These additional signatures were not included in the refined malware signatures as these malware are unique (outliers) and do not share any similarities in SYSCALL signatures with other malware.

The system can be configured to allow high detection accuracy or less number of signatures. Therefore, the overall purpose of the system is to find the balance between the minimal set of malware signatures and high detection accuracy. This significantly reduces the iterations for recursive malware detection algorithm.

4.3.5 Discussion

This section showed that a single signature is enough to designate the executing application as malware. However, it is unable to classify the malware into specific categories. This information can be useful to discern the intention of the attack and

take necessary remedial steps. The section also introduces a malware detection method, using a minimal number of unique signatures that are found only in malware. However, this approach is only able to detect the presence of malware without classifying them.

4.4 Malware Classification

Figure 4.6 illustrates the process of using signatures for malware classification. All malware belonging to a particular malware family (e.g. Trojan, Virus and Worm shown in Figure 4.6) typically exhibit similar behaviour, necessitating a common set of signatures to classify them. However, some behaviours can be seen across different families as illustrated in Figure 4.7. The signatures corresponding to these behaviours are removed as they do not help in the malware classification. Next, the steps described in Section 4.3 are used to obtain the set of malware signatures to identify each malware family as shown in the block labeled 'Signature Extraction' in Figure 4.6. After all the signatures for each malware group are extracted, these sets of signature are compared to identify and remove the common signatures between them to reduce miss-classification. This step can be seen in the block labeled 'Signature for Malware Classification' in Figure 4.6.

The methodology was further extended to classify malware into their respective family. To allow classification, each malware family is initially treated as an entire malware set. However, this technique extracts signatures that can be found at any point of the execution stage, either in the early stages of execution or near the end, depending on the malware behaviour. But, this makes it untenable when early detection and classification is required, such as critical systems where early detection of malware is of paramount importance. To tackle this, a new technique is incorporated that generates signatures to detect malware at different stages of execution.

Figure 4.6: Extracting Signatures for Malware Classification

4.4.1 Real-time Classification

In order to perform a real-time malware classification, the SYSCALL trace is generated and stored for all the malware and benign executables, annotated with the corresponding timestamps, as shown in Figure 4.8. All the malware and benign SYSCALL traces are then time sliced at $1ms$ interval, and the signatures for each malware class is generated at each time interval. Based on extensive experiments, using over 400 malware samples, it has been observed that it takes at most $1ms$ to obtain at least five SYSCALLs as needed by the classification algorithm described in Section 4.3.2.

Common behavior across malware families

Figure 4.7: Common behaviour across malware families

```
0.000000 execve("/bin/ls", ["ls"], [/ * 22 vars */]) = 0
0.001573 brk(NULL) = 0x34000
0.000271 uname({sysname="Linux", nodename="odroid", ...}) = 0
0.000417 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
0.000281 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
0.000197 open("/tls/v7l/neon/vfp/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000193 open("/tls/v7l/neon/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000173 open("/tls/v7l/vfp/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000172 open("/tls/v7l/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000173 open("/tls/neon/vfp/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000170 open("/tls/neon/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000170 open("/tls/vfp/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000169 open("/tls/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000169 open("/v7l/neon/vfp/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000170 open("/v7l/neon/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000169 open("/v7l/vfp/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000181 open("/v7l/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000169 open("/neon/vfp/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000169 open("/neon/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000169 open("/vfp/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000169 open("/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000169 open("/home/odroid/pmctrack/src/lib/libpmctrack/tls/v7l/neon/vfp/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000174 stat64("/home/odroid/pmctrack/src/lib/libpmctrack/tls/v7l/neon/vfp", 0xbe97c590) = -1 ENOENT (No such file or directory)
0.000213 open("/home/odroid/pmctrack/src/lib/libpmctrack/tls/v7l/neon/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.000172 stat64("/home/odroid/pmctrack/src/lib/libpmctrack/tls/v7l/neon", 0xbe97c590) = -1 ENOENT (No such file or directory)
0.000169 open("/home/odroid/pmctrack/src/lib/libpmctrack/tls/v7l/vfp/libselinux.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
...
```

Figure 4.8: Sample of strace output with relative time flag

Figure 4.9 illustrates the process of time-slicing the runtime SYSCALL trace at a pre-defined interval such that the signatures can be generated to represent the different stages of malware execution.

Figure 4.9: Time Slicing System Call Trace

As shown in Figure 4.10, each time slice interval is treated similar to an entire executable, and the techniques explained earlier in Section 4.3.5 are employed to generate the set of malware signatures.

Figure 4.10: Signature Extraction within Time Slice

4.5 Experiments and Results

This section presents the experimental results to show the performance and effectiveness of the proposed approach. A method of monitoring SYSCALL at runtime is presented first. This will be followed by the experiment setup and the data set used in this chapter. Finally, a recent state-of-art method is discussed for comparison in terms of detection and classification accuracy.

4.5.1 Runtime Detection

As shown in Figure 4.11, the application is executed on the system while the system monitors keep track of any SYSCALL that is being executed. During the training phase, these SYSCALL trace can be obtained using the `perf` tool found in Linux OS. However, this method of monitoring SYSCALL trace at runtime is untenable as the monitoring overhead can impede the performance of the system.

To mitigate this, a non-intrusive method is detailed in [168] on how this can be achieved on a SPARC platform. However, another method can be borrowed from [2] that develops a means to implement Rootkit to identify SYSCALL by extracting hardware performance counters (HPC) from ARM-based platforms which are more commonly used. Most ARM processors have been equipped with Performance Monitoring Units (PMU) that allow many architectural hardware events to be monitored. This includes a set of registers called HPC that can be configured to monitor a set of hardware events. These registers are digital counters that increment after the completion of a monitored event. These hardware performance counters often provide an indication of the run-time behaviour of a particular application, hence allowing necessary software debugging and performance tuning to be carried out. The key advantage of using hardware performance counters is that they do not require source code instrumentation. The PMU requires an interrupt service routine to be registered to handle

Figure 4.11: Runtime Malware Detection

Performance Monitoring Interrupts (PMI). The interrupt frequency needs to be set by initialising a performance counter to the desired value and an event selection filter is set to specify which event if to be counted.

The technique in [2] propose using HPC that correspond to EXC SVC and EXC SMC events that are leveraged to extract the SYSCALL. Both these counters are available in both the ARMv7 and ARMv8. A challenge that could occur is instruction skid that refers to the number of instructions that are executed past an instruction which triggers a counts overflow before the PMI halts execution. The interrupt shadow refers

to these instructions collectively, and any HPC event that occurs within this shadow is missed. Fortunately, SYSCALL occur far less frequently than many other codes, thus losing a single SYSCALL invocation due to an interrupt shadow is found to be a rare occurrence which was verified by checking the counter value in the ISR upon overflow.

Once the SYSCALL can be identified, monitoring the SYSCALL trace becomes a trivial process. If the SYSCALL matches with the first SYSCALL in the malware signatures, the system starts to monitor and check if the subsequent SYSCALLs also match. If yes, then the application is flagged as malware.

4.5.2 Experimental Setup

All experiments are conducted on a Virtual Machine with a 4-core CPU and 4GB RAM running Ubuntu 13.04 Linux OS with kernel version 3.8.0. 420 malware samples targeting Linux OS were obtained from the VirusShare [121] website that includes 3 malware families such as Trojan, Virus and Worm, as shown in 4.2. These malware have already been labelled with their families and variant names by the website. They were also scanned using AVG anti-virus software to confirm their malicious nature.

Table 4.2: Malware Data Set

Malware Family	Number
Worm	146
Trojan	134
Virus	140

The benign applications were selected from commonly used applications such as Firefox, Lynx and native Linux commands (ls, pip, netstat, ps, etc.). In this chapter, the 10-fold cross validation method is used in the experiments to evaluate the performance of our approach. To validate the proposed malware classification approach, two different datasets are used: one is for training, while the other is for testing. The

training data and the test data set was split 90/10. The experiment was repeated 10 times, and the results were averaged. The metrics used for evaluation are detection and classification accuracy.

4.5.3 Test Results

This section presents the experimental results of the proposed malware detection system based on the signature of SYSCALL trace. The accuracy of the proposed method has been evaluated based on whether it can be deployed to effectively detect malware at runtime.

4.5.3.1 Detection Accuracy Evaluation

As discussed in Section 4.3.2, the common SYSCALL-based signatures were identified from all the 420 malware samples' trace that do not appear in the benign applications' trace. Figure 4.12 plots the number of malware, in terms of percentage, that contain these identified signatures. The signatures, as plotted on the X-axis, are sorted in descending order of their number of occurrences in the malware samples. For example, from Figure 4.12, *Signature 1* appears in 48% of the malware samples, while *Signature 'n'* appears in 2% of the malware. An average of about 91.6% detection rate was achieved, as shown in the first column of Figure 4.13, using only the top five most frequently occurring signatures.

In order to improve the detection rate, additional signatures were identified for the malware that cannot be detected using the top five most frequently occurring signatures, as described in Section 4.3.4. For the 420 malware samples, five additional signatures were identified that could be used to detect all the malware. The second column in Figure 4.13 shows the detection rate improving to 93.4% detection rate after using the additional set of signatures. Hence, it is evident that the proposed

Figure 4.12: Sorted Malware Signature by percentage (% found in malware)

method only needs a total of ten signatures to detect 420 malware samples. The small, but highly effective, set of signatures do not add a significant overhead to the malware detection mechanism and hence is useful for resource constraint devices.

Figure 4.13: Refined vs Enhanced Technique (Detection %)

4.5.3.2 Classification Accuracy

As illustrated in Figure 4.6, each set of malware signature is extracted using the method described in Section 4.3.5 and Section 4.3.4.

To demonstrate malware classification, the same set of benign applications and malware were used in this experiment as in the previous subsection. The SYSCALL trace was generated and stored for all the malware (see Section 4.5) and benign application executables. However, each malware family is treated as a set of malware during the training phase.

Table 4.3: Results of Malware Classification

Malware Family	Number of Signatures	Classification Accuracy
Worm	7	81.4
Trojan	9	85.2
Virus	7	82.4

For each malware family, the number of signatures identified is shown in the second column of Table 4.3. It is evident that the proposed method only needs a total of 23 signatures to classify the 420 malware samples into their respective families. It is noteworthy that even though, the number of signatures has increased from 10 to 23, the ability to classify malware is essential in malware mitigation. The third column of Table 4.3 shows the accuracy of the proposed malware classification technique. The values in the column represent the number (in percentage) of malware that were correctly classified in the appropriate category. The overall average classification accuracy stands at 83%.

4.5.3.3 Real-time Malware Classification

To demonstrate real-time malware classification using SYSCALL signatures, the same set (see Section 4.5) of benign applications and malware were used in this experiment.

As shown in Figure 4.14, the SYSCALL trace is obtained from the executables and are sliced at a fixed time interval. For each time interval, the signatures are extracted using the technique described in Section 4.4.1. For example, the following excerpts show the malware signatures for first four time-intervals for `Virus` malware family.

Figure 4.14: Extracting Signatures for Real-time Malware Classification

Figure 4.15 shows the real-time classification accuracy for different malware families, where the X-axis represents the time stamps and the Y-axis represents the real-time classification accuracy (in %) at the different time instances. Results for 3 malware families have been shown in this figure. Each line in this graph denotes the real-time classification accuracy for each malware family at different time instances. Table 4.4 shows that the average real-time classification accuracy for each malware family is comparable to the classification accuracy results shown in Table 4.3.

Figure 4.15: Real-time Classification Accuracy

Table 4.4: Results of Real-time Malware Classification

Malware Family	Average Number of Signature per Time Instance	Real-time Classification Accuracy
Worm	3	80.8
Trojan	5	84.9
Virus	3	83.1

4.5.4 Comparison with State-of-the-art

To demonstrate the effectiveness of the proposed approach in this chapter, it is compared with the latest state-of-art method [169]. Zhang et al. (2018) proposed a sensitive SYSCALL-based principal component initialised multi-layer neural network to detect malware. In this work, the authors have compared with several research works [26], [55] published in this area.

Table 4.5 shows the comparison in terms of detection accuracy (column 2) and classification (column 3) accuracy between the approach in [169] and the proposed techniques in this chapter. The results for the [169] approach has been taken as reported in their work. It can be observed that while the proposed approach achieves similar detection and classification accuracy as the current state-of-the-art work, it goes one step further by allowing for real-time classification. As explained earlier in Section 4.4.1, the ability to detect and classify malware in real-time is essential, especially for critical systems where any delay in detecting and classifying anomalous

behaviour can lead to disastrous outcomes. Additionally, unlike the computationally complex multi-layer neural network based technique proposed in [169], our approach relies on a more straightforward heuristic search based approach for rapid malware detection and classification without sacrificing the detection or classification accuracy.

Table 4.5: Comparison with State-of-the-art

Method	Detection Accuracy	Classification Accuracy	Real-time Classification Accuracy
State-of-the-art Approach [169]	92	85.68	Not Available
Proposed Approach	93.4	83	82.9

4.6 Summary

In this chapter, a real-time SYSCALL-based signature technique was proposed for malware detection and classification. As opposed to existing methods, the proposed approach relies on a minimal set of SYSCALL signatures for detecting a group of malware instead of using a unique signature per malware. Not only does the method allows to detect malware at runtime with a minimal set of signatures, but it was also able to classify the malware early before it has a chance to complete.

It has been shown in Figure 4.13 that SYSCALL trace can be used to detect malware. The proposed methodology was used to successfully extract 10 signatures to detect 420 malware with 93.4% detection accuracy.

Additionally, as shown in Table 4.3, the proposed method only needs a total of 23 signatures to classify the 420 malware samples into their respective families with an average classification accuracy of 83%. It is noteworthy that even though, the total number of signatures has increased from 10 to 23, the ability to classify malware is essential in malware mitigation. In addition, it is also observed that the proposed

approach achieves similar detection and classification accuracy as the current state-of-the-art work, but it goes one step further by allowing for classification at runtime. Unlike existing methods such as [164], [165], and [166], the proposed technique is able to perform classification within $1ms$ time frame with an 82.9% classification accuracy. This provides early detection and classification, which is required in critical systems where early detection of malware is of paramount importance.

Moreover, implementing these techniques using HPC allows us to avoid the instrumentation of either the operating system or the application. This makes the proposed technique scalable and easy to implement since new operating systems or applications do not need to be instrumented to obtain the SYSCALL trace.

While the SYSCALL-based technique proposed in this chapter, can be used to detect and classify malware at `user-space` application, it is unable to detect nor classify an attack that resides in the operating system's `kernel-space` (i.e. Rootkit). The next chapter presents a novel architectural event-based technique that can be performed at runtime using hardware performance counters.

Rootkit Detection at Kernel-space

5.1 Introduction

The previous chapter presented a technique to detect and classify user-space malware by analysing the SYSCALL trace during application execution. The technique was then extended to provide for real-time malware detection and classification that allows early detection of malware so that appropriate remedial steps can be taken as soon as possible. However, the technique only relies on the sequence of SYSCALLs and does not focus on the execution characteristics of each SYSCALL. Therefore, any changes to the SYSCALL itself cannot be detected by the technique proposed in the previous chapter. This chapter leverages architectural events through hardware performance counters (HPCs) and a machine learning based classifier to generate a model that differentiates between unadulterated applications and anomalous ones. Subsequently, the same technique is applied to SYSCALLs to detect any malicious SYSCALLs that indicate an attack on the OS.

Hardware performance counters have been used to profile the performance of an application on a platform and can be used as a reliable indicator of the system's behaviour. In addition, existing processors have a plethora of hardware performance counters that can be relied upon to monitor and analyse various scenarios. How-

ever, typical systems only allow a few of them to be monitored concurrently in order not to impede system performance. Thus, the selection of the most appropriate set of hardware performance counters for anomaly detection becomes an important aspect. Moreover, the sensitivity of hardware performance counters makes it an inconsistent metric if the wrong HPCs are selected. Therefore, it is important to ensure that the same application can be accurately modelled using the selected hardware performance counters. The fixed number of applications running on a typical embedded system makes it feasible to store such models for all applications that can be executed during the lifetime of the embedded system. At runtime, these models are used to verify the integrity of these SYSCALLs by comparing the runtime characteristics through HPCs with the one stored from offline characterisation. The fact that there are a fixed and finite number of SYSCALLs in an operating system makes the proposed strategy feasible.

Existing work has explored the use of HPCs and machine learning for anomaly detection. A machine learning based method for classification of anomalies has been discussed in [70] that detects the anomalies based on their efficiency to embed function call graphs into selected feature mapping. HPC-based signatures in [86] focus on generating Rootkit signatures by modelling infected SYSCALLs. Attempts to build an anomaly detector in hardware using performance counters are done in [89]. This hardware anomaly detector is designed to run securely beneath the system software such that robustness of the detection is increased. Ozsoy et al. [87] designed a hardware anomaly detector that acts as the first line of defence to differentiate between benign and anomaly. After the prioritization of the hardware classifier, further software detection mechanisms are carried out. In work [42], a singular value decomposition (SVD) is used to reduce the hardware event vectors into a set of characterization features. These reduced features are further used by machine learning techniques to identify anomalies. These methods use HPC to model the malicious behaviour and

benefit from low-performance overheads. However, they are limited by a high false positive rate.

In this chapter an architecture-specific feature-based detection technique is proposed that uses SVM classifier to generate a model using selected hardware performance counters to distinguish anomalous applications from benign ones. Next, this technique is extended to verify the integrity of the SYSCALLs at runtime. The main advantage of this approach is that it is faster and computationally more lightweight method in detecting the anomaly while maintaining a high detection rate. In addition, usage of hardware performance counters that cannot be evaded by any obfuscation techniques [42], gives this method an advantage over existing anomaly detection techniques.

In the rest of this chapter, Section 5.2 introduces kernel verification using architecture-specific features. This is followed by the benchmark applications and generated anomalous applications used in this chapter. Section 5.2.3 describes the selection of HPC events used in this experiment. Section 5.3 presents the results obtained from the techniques proposed in this chapter and highlights the pros and cons of the approach. Finally, the proposed methodology is deployed to verify SYSCALLs, followed by a summary for this chapter.

5.2 Anomaly Detection using Architecture-specific Features

In this section, an overview of the proposed methodology is presented. This is followed by the benchmark applications used in our experiments. Subsequently, the steps taken in selecting the HPCs to be used as features together with SVM classifier to generate the model are outlined. The model is then used at runtime to differentiate benign applications from anomalous ones.

5.2.1 Methodology Overview

Figure 5.1 illustrates the proposed two stages of the design flow. The first stage is an offline process in which appropriate hardware performance counters are selected for a given set of applications. This is followed by generating a list of anomalous applications from the given benign applications. The benign and anomalous applications are then executed multiple times to obtain the event counts for the selected hardware performance counters that are then fed into an SVM classifier algorithm to generate a model that is used to differentiate between benign application and anomalous ones.

Figure 5.1: Overview of the proposed method

The second stage which uses a dedicated hardware processor to detect malicious software in the main processor, which is aimed to achieve fine-grained, real-time detection with minimal performance overhead. The monitoring imposes no overhead onto the LEON3 soft-core processor as all the monitoring is done by the Cortex-A9 processor that is executing the monitoring algorithm. Moreover, a dedicated monitoring system ensures anomaly detection, even if the main processor being monitored is compromised.

This monitoring processor utilizes the model generated by the SVM classifier to identify if the application being executed on the main processor is benign or anomalous at runtime.

Figure 5.2 illustrates how the proposed multi-processor framework is designed in such a way that the Cortex-A9 processor via the HPS2FPGA bridge has access to the LEON3 system address space through the AHB2AXI/AXI2AHB [170] bus and is able to extract the events count at runtime with minimal overheads to the LEON3 main processor. The benchmark applications are executed on the LEON3 soft-core processor [171] that becomes the main processor while the Cortex-A9 processor is configured for runtime anomaly detection.

Figure 5.2: Instruction trace filter and buffer for runtime detection

The Terasic SoCKit platform [172] is used to illustrate our methodology of anomaly detection using hardware performance counters. The runtime applications are executed on the LEON3 soft-core processor that has been designated as the main processor while the Cortex-A9 processor executes the monitoring algorithm with the SVM model that is generated offline. Additionally, this evaluation platform will also be used to demonstrate the ability to detect anomalous application at runtime using the four selected hardware performance counters.

5.2.2 Generating Anomalous Applications

In this work, the application package of the widely recognized CHStone benchmark suite [3] is considered. This suite is written in C and covers a wide range of application fields like media processing, digital architecture, system security etc. One of the main advantages of these benchmark applications is that the test vectors are inbuilt, and hence, these applications are self-contained. Table 5.3 lists the twelve CHStone benchmark applications used in this experiment. The table also provides some characterization of these applications by listing the total lines of codes and function count found for each benchmark application.

In order to improve the quality of the model, more benchmark applications are needed for the training step. Hence, a code stitching approach was adopted, similar to [88], for generating more anomalous applications for the training and testing steps.

Table 5.1 and Table 5.2 list the thirty-one additional applications that were generated using the code stitching method from the twelve CHStone benchmark applications. These thirty-one anomalous applications are then split into two different sets. The first set is the random mixing of two different benchmark applications that are randomly mixed to obtain twenty new applications for the training and testing phase of our classification. The random mixing helps to analyse the performance of our

proposed method in detecting complex anomalies. In addition, the sequence of sub-functions to be mixed are randomly varied at every interval so that the new sample in the data is always unique.

Table 5.1: Anomalous applications generated (application mixing)

Stitched Code	Combination
Case 1	ADPCM + AES
Case 2	ADPCM + DFADD
Case 3	AES + MIPS
Case 4	MOTION + SHA
Case 5	GSM + JPEG
Case 6	JPEG + BLOWFISH
Case 7	GSM + DFDIV
Case 8	ADPCM + DFMUL
Case 9	AES + DFSIN
Case 10	MOTION + BLOWFISH
Case 11	DFADD + BLOWFISH
Case 12	AES + MOTION
Case 13	GSM + BLOWFISH
Case 14	GSM + DFMUL
Case 15	SHA + DFDIV
Case 16	MOTION + DFMUL
Case 17	SHA + DFADD
Case 18	AES + DFDIV
Case 19	JPEG + GSM
Case 20	ADPCM + DFSIN

The second set that comprises of eleven applications takes the *reset* function from ADPCM application and injects (this injected code will be known as a *patch*, see Table 5.2) it into the original benign benchmark applications (in this case the twelve CHStone benchmark applications) at random locations. As the injected function comes from ADPCM, it has been omitted from this modification. This exercise is meant to improve the model that better represents the original application and is able to differentiate applications that are slightly modified versions of the original application. All these generated applications will all be labelled as anomalous applications during the SVM classifier training and testing.

Table 5.2: Anomalous applications generated (code patching)

Stitched Code	Combination
Case 21	PATCH + AES
Case 22	PATCH + BLOWFISH
Case 23	PATCH + DFADD
Case 24	PATCH + DFDIV
Case 25	PATCH + DFMUL
Case 26	PATCH + DFSIN
Case 27	PATCH + GSM
Case 28	PATCH + JPEG
Case 29	PATCH + MIPS
Case 30	PATCH + MOTION
Case 31	PATCH + SHA

5.2.2.1 Input Data

The standard input data provided by the CHStone benchmark suite were chosen for evaluation. The input data for CHStone benchmark are selected based on the assumption that it represents the realistic input to the application during execution time [3].

5.2.3 Selection of Hardware Performance Counters

Most modern processors have been equipped with Performance Monitoring Units (PMU) that allow the monitoring of many architectural hardware events. This includes a set of registers called hardware performance counters that can be configured to monitor a set of hardware events. These registers are digital counters which increment after the completion of a monitoring event. These hardware performance counters often indicate the run-time behaviour of a particular application, hence allowing necessary software debugging and performance tuning to be carried out. The key advantage of using hardware performance counters is that they do not require source code instrumentation.

Table 5.3: Considered Benchmarks from CHStone [3]

Application	Description	Lines of code	Function count
DFADD	Double-precision floating-point addition	494	17
DFMUL	Double-precision floating-point multiplication	363	16
DFDIV	Double-precision floating-point division	419	19
DFSIN	Sine function for double-precision floating-point numbers	789	31
MIPS	Simplified MIPS processor	232	1
ADPCM	Adaptive differential pulse code modulation decoder and encoder	547	15
GSM	Linear predictive coding analysis of global system for mobile communications	380	12
JPEG	JPEG image decompression	1397	31
MOTION	Motion vector decoding of the MPEG-2	441	13
AES	Advanced encryption standard	723	11
BLOWFISH	Data encryption standard	1413	6
SHA	Secure hash algorithm	1286	8
AES	Advanced Encryption Standard	1100	11
CRC	32-bit Cyclic Redundancy Check	188	4

Table 5.4 lists the hardware performance counters found in the LEON3 soft-core processor. The hardware events on a platform can be categorized into different groups, such as load/store events (e.g. *LOAD and STORE Instructions*, *LOAD Instructions* and *STORE Instructions*), cache events (e.g. *Instruction Cache Miss*, *Instruction MMU TLB Miss*, *Data Cache (Read) Miss*, *Data MMU TLB Miss* and *Branch Prediction Miss*), instruction events (e.g. *Integer Instructions*, *Floating-Point Unit Instruction Count*, *CALL Instructions* and *Regular Type 2 Instructions*), execution events (e.g. *Execution time*) and so on. This gives us many options to pick as features for training a model using machine learning.

Table 5.4: List of hardware performance counters found in the LEON-3 processor

#	Hardware Performance Counters
1	Instruction Cache Miss
2	Instruction MMU TLB Miss
3	Data Cache (Read) Miss
4	Data MMU TLB Miss
5	Integer Instructions
6	Floating-Point Unit Instruction Count
7	Branch Prediction Miss
8	Execution Time
9	Integer Branches
10	CALL Instructions
11	Regular Type 2 Instructions
12	LOAD and STORE Instructions
13	LOAD Instructions
14	STORE Instructions

However, not all the hardware events are good candidates for modelling as the occurrence of some events depends on the current system state, which continually changes. Thus, the number of occurrences during execution may vary significantly from one execution to another. Therefore, it is important to carefully select proper hardware events whose occurrences are consistent each time the application is executed [86].

Hardware performance counters like *Instruction MMU TLB Miss*, *Data MMU TLB Miss*, *Floating-Point Unit Instruction Count* are not considered for evaluation as a count of zero were obtain across all twelve benchmarks applications (listed in Table 5.3) during execution. In addition, *LOAD and STORE Instructions* are also not considered as they are the aggregation of *LOAD Instruction* event count and *STORE Instructions* event count. Differences of load and store events in the benchmark applications improve profiling.

The *Perf* tool in Linux is used to obtain the hardware performance counters for all 12 benchmark applications, mainly by merely reading the counter values from the appropriate registers. While infrequent reading of these register values does not

create significant overhead, reading them frequently, for example, to create a runtime profile of each performance counter, typically creates a significant overhead for the system [173][174]. Hence, the proposed technique refrains from reading the counter values too frequently to keep the resulting overhead to a minimum. Instead, it relies on the overall hardware performance counter values, obtained at the end of executing each benchmark application.

Each benchmark application is executed ten times, and a standard deviation is calculated for each hardware performance counter. The standard deviation values are averaged for each counter across all benchmark applications. As shown in Figure 5.3, the performance counters are then sorted from the least standard deviation (*Data Cache (Read) Miss*) to the performance counter with the most deviation (*Execution time*). Lastly, four performance counters that exhibit the most consistent behaviour are selected for the subsequent steps.

Figure 5.3: Averaged standard deviation of hardware performance counters across twelve benchmark applications

After careful analysis, from the existing set of available hardware events, four hardware performance counters were selected, namely *Instruction Cache Miss*, *Data Cache Miss*, *Branch Prediction Miss* and *Call Instructions* for anomaly detection. In our implementation, the LEON3 soft-core processor is configured to monitor these

four hardware performance counters concurrently. The hardware performance counters selected as features are *Instruction Cache Miss*, *Data Cache (Read) Miss*, *Branch Prediction Miss* and *Call Instructions*, which are the hardware performance counters with the least standard deviation. Hence, these hardware performance counters, exhibiting the least fluctuation, are chosen to characterize the benchmark application.

5.2.4 Support Vector Machine Classifier

Generally, machine learning classifiers observe the given samples and determine the class to which each sample belongs to. This technique often leaves a probability value or a degree of closeness of a particular sample to a particular predefined class. For this work, two classes have been defined: anomaly and benign.

This work adopts a supervised non-probabilistic binary linear classifier called Support Vector Machine (SVM) [175]. However, the classifiers such as Decision Tree (DT), Naive-Bayes (NB), and k-Nearest Neighbors (kNN) can also be used for this purpose [24] [39] [176]. Given a set of labelled training samples, an SVM training algorithm builds a model that assigns new samples to one class or the other. A model generated by SVM represents the training samples as points in space, mapped so that the training samples of the different class are distinctively divided by a gap. Test samples are then mapped into that same space and predicted to belong to a class based on which side of the gap they reside.

Compared to other machine learning approaches, SVM based approach has many advantages. As discussed in [177], SVM is based on the structural risk minimization principle instead of empirical risk minimization principle. This ensures that the model is tightly fitted to the trained dataset and once a solution is reached, it has

the optimum characteristic. Moreover, SVM also has stronger generalization ability [178].

5.3 Experiments and Results

This section evaluates the proposed method for detecting malicious applications by observing the selected (see Section 5.2.3) hardware performance counters at runtime and comparing them with the model generated offline using SVM classifier algorithm.

5.3.1 Experimental Setup

In this section, Terasic's SoCKit [172] (as shown in Figure 5.4) running Altera's hybrid FPGA/Cortex-A9 microprocessor is used to run the experiments. The reconfigurable space in the FPGA has been configured with a SPARC-based LEON3 soft-core processor [171] to act as the main processor. On the other hand, the ARM Cortex-A9 hard processor system is used to monitor and analyse the selected four hardware performance counters at run-time. The analysis code is run bare metal on the A9 processor.

To illustrate the concept, LEON3 SPARC-V8 soft-core processor, obtained from Terasic, is configured and executed on the Terasic SoCKit platform. The LEON Linux running on this soft-core processor is based on kernel version 3.10.106 and is maintained by Cobham Gaisler [179].

The host computer used to configure and debug the LEON3 soft-core processor on the FPGA uses the open source CentOS 6.5 (64-bit) [180] Linux operating system installed on a VMware Workstation [181]. Additionally, Altera's Quartus II 14.1 (64-bit) [182] is used to build and generate the bitstream of the LEON3 soft-core processor to be configured on the FPGA.

Figure 5.4: Terasic’s SoCKit used in the experiment

Moreover (as discussed in Section 5.2.3), four hardware performance counters, namely *Instruction Cache Miss*, *Data Cache Miss*, *Branch Prediction Miss*, and *Call Instructions* were selected for monitoring.

5.3.2 Benchmarks

As shown in Table 5.5, all twelve benchmark applications and twenty generated anomalous applications were selected as the training set. The twenty generated anomalous applications are separated into two categories; thirteen represents the combination of any two benchmark applications, and seven represents the *patch*-ed modification of the benchmark applications. The remaining eleven generated anomalous applications from both categories is used to test the model. The second column shows whether the application is benign or anomalous.

The four hardware performance counters discussed earlier in Section 5.2.3 are extracted for all these applications in both the training and test set.

Table 5.5: Benchmark Applications for Training and Testing Phase

APPLICATIONS	LABEL	TRAINING	TESTING
DFADD	benign	✓	✓
DFMUL	benign	✓	✓
DFDIV	benign	✓	✓
DFSIN	benign	✓	✓
MIPS	benign	✓	✓
ADPCM	benign	✓	✓
GSM	benign	✓	✓
JPEG	benign	✓	✓
MOTION	benign	✓	✓
AES	benign	✓	✓
BLOWFISH	benign	✓	✓
SHA	benign	✓	✓
ADPCM + AES	anomaly	✓	
ADPCM + DFADD	anomaly	✓	
AES + MIPS	anomaly	✓	
MOTION + SHA	anomaly	✓	
GSM + JPEG	anomaly	✓	
JPEG + BLOWFISH	anomaly	✓	
GSM + DFDIV	anomaly	✓	
ADPCM + DFMUL	anomaly	✓	
AES + DFSIN	anomaly	✓	
MOTION + BLOWFISH	anomaly	✓	
DFADD + BLOWFISH	anomaly	✓	
AES + MOTION	anomaly	✓	
GSM + BLOWFISH	anomaly	✓	
PATCH + AES	anomaly	✓	
PATCH + BLOWFISH	anomaly	✓	
PATCH + DFADD	anomaly	✓	
PATCH + DFDIV	anomaly	✓	
PATCH + DFMUL	anomaly	✓	
PATCH + DFSIN	anomaly	✓	
PATCH + GSM	anomaly	✓	
GSM + DFMUL	anomaly		✓
SHA + DFDIV	anomaly		✓
MOTION + DFMUL	anomaly		✓
SHA + DFADD	anomaly		✓
AES + DFDIV	anomaly		✓
JPEG + GSM	anomaly		✓
ADPCM + DFSIN	anomaly		✓
PATCH + JPEG	anomaly		✓
PATCH + MIPS	anomaly		✓
PATCH + MOTION	anomaly		✓
PATCH + SHA	anomaly		✓

These performance counter values are used for the training and testing of SVM classifier library implemented in a sci-kit [183] tool using the popular Python language. The training used an SVM classifier with the Radial Basis Function (RBF) kernel with an empirically obtained ‘C’ and ‘gamma’ values of 1.0 and 100 respectively. The python script used in the sci-kit library has been included in Appendix B.

All the benchmark applications and generated anomalous application are compiled and executed on the LEON3 soft-core processor. Each application is executed ten times, and the four selected event counts are recorded, and the average value is calculated. Next, the event counts for all the benchmark applications, and twenty of the generated anomalous applications (see column ‘Training’ in Table 5.5) are used to train the model using the SVM classifier (as explained in Section 5.2.3). The model is then stored in the Cortex-A9 monitoring processor. Then at runtime, when an application completes its execution, the monitoring processor extracts the four hardware performance counters and feeds it into the model (generated offline using the SVM classifier) to determine if the application is benign or anomalous. The next subsection evaluates the proposed strategy based on the selected benign and anomalous applications (see column ‘Testing’ in Table 5.5).

5.3.3 Test Results

Table 5.6 shows the averaged count for the four selected hardware performance counters for all the applications and generated anomalous applications used to test the SVM based model. Columns 2 through 5 of Table 5.6 tabulates these counter values for all the applications. As shown in Table 5.6 (row 1 to row 12), all the benign applications are used during the training phase as the model is meant to represent these applications. The anomalous applications created in this work are split into different training and testing sets. These benign applications are tested using the generated model, to confirm that they can indeed be correctly identified as benign applications.

It can be seen that the model represents the training set well as no training error is detected.

Moreover, the SVM generated model is able to differentiate between two similar applications where the event counts are very close to each other. This can be observed in Table 5.6, row 2, where DFMUL have an *Instruction Cache Miss*, *Data Cache (Read) Miss*, *Branch Prediction Miss* and *Call Instructions* of 5311, 123, 15039 and 5688 while the anomalous version of *patch + DFMUL* have an event counts of 5949, 193, 15035 and 5890 respectively. Likewise, the model is able to differentiate between *MOTION* benchmark application and its corresponding *patched* version, even though they have event counts that are close to each other.

Table 5.7 shows the results of the classification for the testing dataset. It is observed that none of the anomalies was identified as a benign application, and all the benign applications were correctly identified. This is an important indicator as identifying an anomalous application as a benign application is far more damaging than identifying the benign application as an anomalous one. This confirms the efficiency of the proposed anomaly detection technique using hardware performance counters.

5.4 Kernel Verification

The earlier section discussed how a set of benign applications could be modelled using a machine learning SVM classifier to distinguish it from anomalies. It is vital to detect a malicious activity early during runtime and mitigate the problem by immediately arresting the execution to minimize damage.

However, SYSCALLs are privileged operating system functions that perform specific work in the order of a few micro-seconds (Table 5.8). The table shows the execution time of a few SYSCALLs obtained from the ODROID-XU3 platform [103] clocked

Table 5.6: Training and Testing Application HPC Count

Applications	Instruction Cache Miss	Data Cache (Read) Miss	Branch Pre- diction Miss	CALL Instructions
DFADD	11450	251	33344	12083
DFMUL	5311	123	15039	5688
DFDIV	7142	104	18491	7254
DFSIN	26764	610	209972	176950
MIPS	333	67	2118	301
ADPCM	567	132	3143	4877
GSM	509	67	2052	1930
JPEG	15965	4332	130069	18607
MOTION	342	110	382	306
AES	1633	183	3877	1383
BLOWFISH	392	684	5998	1612
SHA	383	572	1139	558
ADPCM + AES	3687	279	9055	9460
ADPCM + DFADD	12396	226	377842	20057
AES + MIPS	2855	280	6542	1990
MOTION + SHA	534	687	2531	878
GSM + JPEG	5582	4708	87828	55374
JPEG + BLOWFISH	5591	4479	93904	54543
GSM + DFDIV	8809	255	19166	9644
ADPCM + DFMUL	6261	184	19506	13360
AES + DFSIN	45708	609	215859	190602
MOTION + BLOWFISH	554	793	8127	1649
DFADD + BLOWFISH	10497	873	35991	11923
AES + MOTION	2111	246	4722	2015
GSM + BLOWFISH	738	696	8833	3755
PATCH + AES	2447	198	4657	1994
PATCH + BLOWFISH	430	716	8078	1614
PATCH + DFADD	13779	229	33370	12633
PATCH + DFDIV	9547	141	18761	7530
PATCH + DFMUL	5949	193	15035	5890
PATCH + DFSIN	41541	870	211654	188967
PATCH + GSM	639	90	2048	2451
GSM + DFMUL	6280	199	15603	8026
SHA + DFDIV	8621	748	20784	8042
MOTION + DFMUL	411	120	430	347
SHA + DFADD	11916	731	35292	13121
AES + DFDIV	10041	340	22852	9184
JPEG + GSM	7099	4861	87827	55374
ADPCM + DFSIN	39188	650	215951	196401
PATCH + JPEG	7186	4274	86189	53239
PATCH + MIPS	396	80	2299	315
PATCH + MOTION	412	122	442	344
PATCH + SHA	453	590	2501	849

Table 5.7: Results of the Test Phase

Applications	Classification
DFADD	benign
DFMUL	benign
DFDIV	benign
DFSIN	benign
MIPS	benign
ADPCM	benign
GSM	benign
JPEG	benign
MOTION	benign
AES	benign
BLOWFISH	benign
SHA	benign
GSM + DFMUL	anomaly
SHA + DFDIV	anomaly
MOTION + DFMUL	anomaly
SHA + DFADD	anomaly
AES + DFDIV	anomaly
JPEG + GSM	anomaly
ADPCM + DFSIN	anomaly
PATCH + JPEG	anomaly
PATCH + MIPS	anomaly
PATCH + MOTION	anomaly
PATCH + SHA	anomaly

at 200Mhz running Linux kernel version 3.10.106 [184]. A typical application can contain hundreds if not thousands of SYSCALL invocations during the application's execution.

By taking advantage of HPC's fine-grained sensitivity in capturing the behaviour of SYSCALLs within short execution cycles, the detection of anomalous SYSCALLs and its subsequent remedial actions can be performed even before the application completes its execution.

Table 5.8: Duration of SYSCALL frequently targeted by a Rootkit

SYSCALL	Time (μs)
open	1.64
close	14.03
read	2.41
write	17.96

This section describes how SYSCALLs are verified using HPC and to further detect if the anomalies are due to Rootkit attacks. Figure 5.5 highlights the capability

of using hardware performance counters in detecting Rootkit. A benign application with and without 'suterusu' Rootkit is executed on the platform to verify that HPCs can be used to detect the presence of a Rootkit. In the figure, it can be seen that for cases, 'who' and 'pwd' the percentage difference in SYSCALLs is less than 0.5%. Work in [139], cannot detect this case. It is observed that in almost all the cases the HPCs could record a significant difference averaging more than 2.5%. It is seen in the figure that the SYSCALL trace is not affected by Rootkit while the HPC counters are affected by the presence of a Rootkit.

Figure 5.5: Hardware Performance Counter change when a Rootkit is present

5.4.1 Selection of Vulnerable System Calls

It was discussed in Section 2.1 that a Rootkit typically hides once it has infected the operating system. It then manipulates the operating system by modifying the SYSCALLs to perform the intended malicious activities while hiding within the kernel. Table 5.9 lists of nine SYSCALLs that are typically targeted by Rootkit [86].

Table 5.9: List of Monitored System Call

System Calls	Description
open	open and possibly create a file
close	close a file descriptor
read	read from a file descriptor
write	write to a file descriptor
lseek	reposition read/write file offset
mmap	map or unmap files or devices into memory
opendir	open a directory
readdir	read a directory
getdents	get directory entries

5.4.2 Selection of Hardware Performance Counters

In addition, the system calls are executed with various typical inputs so that the general behaviour of the system call can be captured. An important aspect of this method is the selection of HPC from the plethora of events. The same technique, as discussed in Section 5.2.3, was used to select the appropriate HPCs for the list of SYSCALLs (seen in Table 5.9). This is to ensure that only stable HPCs are selected for monitoring the SYSCALLs.

After careful analysis, from the existing set of available hardware events, four hardware performance counters were selected, namely *LOAD*, *STORE*, *Branch Prediction Miss* and *L1 Instruction Cache Refill* for anomaly detection. It is noted that the list of selected hardware events are different from the list in Section 5.2.3 as this section are focused on various SYSCALLs while the previous section looks at various applications.

5.4.3 Generating the Classifier Model

In Section 5.2.4, the SVM classifier is used to differentiate the benign applications from the anomalies. The same technique is used to distinguish between a benign SYSCALL from an anomalous one. Moreover, similar to the technique discussed

in Section 5.2, to distinguish benign SYSCALLs from anomalies, ‘tainted’ SYSCALLs are included in the training set to ensure the model only detects the typical operations of the SYSCALLs and rejects other behaviours due to changes to the SYSCALLs.

However, in this section, the anomaly is further classified to identify Rootkit. This is achieved by using the SVM classifier to distinguish the anomaly to identify trained Rootkits from other forms of anomalies. This is only possible as the number of existing Rootkit is limited [185], unlike user-space malware that spans in the millions [121].

It should be noted that the list of selected hardware performance counters in this section is different from the list of hardware performance counters selected in Section 5.2.3. The difference in the selected hardware performance counters arises from the fact that this section focuses on analysing ‘system calls’ and therefore selects suitable HPCs that are stable across different ‘system calls’ during multiple execution runs. On the other hand, Section 5.2.3 focuses on analysing various ‘applications’ and hence selects suitable HPCs that are stable across different applications during multiple execution runs.

5.4.4 Experiments and Results

This section evaluates the proposed method for detecting malicious SYSCALLs by observing the selected (see Section 5.2.3) hardware performance counters at runtime and comparing them with the model generated offline using SVM classifier algorithm. In addition, anomalies are further classified to identify Rootkits.

5.4.4.1 Experimental Setup

To run the experiments in this section, Hardkernel’s ODROID-XU3 (as shown in Figure 5.6) running a Samsung’s Exynos system on chip (SoC) comprising of Cortex-A7 and Cortex-A15 cores is used. In this setup, the Cortex-A7 has been configured to act

as the main processor while the other Cortex-A7 core is used to monitor and analyse the selected hardware performance counters at run-time. The ODROID-XU3 has a Linux running in it with a kernel distribution of 3.10.106. The Linux SYSCALLs on the ODROID-XU3 are used as the benign set while the Rootkit and ‘tainted’ SYSCALLs were compiled and executed on the Exynos SoC in the ODROID-XU3.

Figure 5.6: Image of ODroid-XU3 Platform

The work in earlier Section 5.3 focuses on detecting anomalies in ‘applications’ by observing the architecture-specific features, namely HPCs. Hence the LEON3 platform is used for experiments as it has the necessary tools to monitor the HPCs. On the other hand, this section focuses on detecting anomalies in an operating system. This is achieved by monitoring the HPCs of the OS SYSCALLs. Hence an operating system is required for the experiments to monitor its SYSCALLs. Unlike the LEON3 platform, the ODROID-XU3 platform incorporates a widely used Ubuntu Linux OS. Hence, the Odroid-XU3 platform is used in this section in order to validate the proposed techniques on a widely used OS and to confirm its usefulness on many similar platforms.

5.4.4.2 Benchmarks

As shown in Table 5.10, all eight SYSCALLs that would be executed in various conditions were selected for the training and testing set. The first condition is the original SYSCALL that is designated as a benign SYSCALL. Next, five sets of ‘tainted’ SYSCALLs are included in the dataset. Similar to the process described in Section 5.2.2, short snippets of code are inserted into these SYSCALLs. Finally, three Rootkits are included to evaluate the technique. The ‘✓’ represents the SYSCALLs that are in the training set, while the ‘O’ represents the SYSCALLs that will be in the testing set. HPC values are used for the training and testing of SVM classifier using scikit-learn [183] machine learning tool for Python. The training set used an SVM classifier with the Radial Basis Function (RBF) kernel with an empirically obtained ‘C’ and ‘gamma’ values of 1.0 and 100 respectively.

Table 5.10: Train and Test Dataset

Conditions	Label	System Calls							
		open	close	read	write	lseek	opendir	readdir	getdents
benign	benign	✓ O	✓ O	✓ O	✓ O	✓ O	✓ O	✓ O	✓ O
taint 1	anomaly	✓	✓	✓	✓	✓	✓	✓	✓
taint 2	anomaly	✓	✓	✓	✓	✓	✓	✓	✓
taint 3	anomaly	O	O	O	O	O	O	O	O
taint 4	anomaly	O	O	O	O	O	O	O	O
taint 5	anomaly	O	O	O	O	O	O	O	O
suterasu	rootkit	✓ O	✓ O	✓ O	✓ O	✓ O	✓ O	✓ O	✓ O
diamorphine	rootkit	O	O	O	O	O	O	O	O

Each SYSCALL is executed ten times, and the four selected HPC values are averaged. Next, the HPC values for all the SYSCALLs in varying conditions (see ‘✓’ in Table 5.10) are used to train the model using the SVM classifier (as explained in Section 5.2.3).

The next subsection evaluates the proposed strategy based on the selected benign and anomalous SYSCALLs (see ‘O’ in Table 5.10).

5.4.4.3 Experimental Results

As discussed in Section 5.4, four HPCs that are stable for the eight SYSCALL (shown in Table 5.9) were identified and selected. To demonstrate the proposed method, each time the selected SYSCALL is invoked, the four HPC values are obtained. The model is then used to verify the SYSCALL during the detection phase at runtime when any SYSCALL (shown in Table 5.9) is invoked. It will first differentiate the benign SYSCALL from the anomalous ones. Next, if the SYSCALL has been classified as anomalous, it goes one step further to determine if it is due to a Rootkit.

As shown in Table 5.10 (row 1, 'benign' condition), all the benign SYSCALLs are included during the training phase as the model is meant to represent these SYSCALLs. The different sets of 'tainted' SYSCALLs created in this work are split into different training and testing sets. In addition, one untrained Rootkit is included in the testing set to evaluate the technique. These benign SYSCALLs are tested using the generated model, to confirm that they can indeed be correctly identified as benign. It can be seen that the model represents the training set well as no training error is detected.

Table 5.11 shows the classification result for each SYSCALL in the varying conditions for the testing dataset. It is observed that none of the anomalies was identified as a benign application, and all the benign applications were correctly identified. This is an important indicator as identifying an anomalous SYSCALL as benign is far more damaging than identifying the benign SYSCALL as an anomalous one. Moreover, the two Rootkits that were included in the training set is classified correctly as a Rootkit. However, the untrained Rootkit is partially classified as an anomaly, specifically for the 'lseek' and 'getdents' SYSCALLs while the remaining SYSCALLs are classified correctly as a Rootkit. This confirms the efficacy of the proposed kernel anomaly detection technique using hardware performance counters.

Table 5.11: Test Results

Conditions	System Calls							
	open	close	read	write	lseek	opendir	readdir	getdents
benign	benign	benign	benign	benign	benign	benign	benign	benign
taint 3	anomaly	anomaly	anomaly	anomaly	anomaly	anomaly	anomaly	anomaly
taint 4	anomaly	anomaly	anomaly	anomaly	anomaly	anomaly	anomaly	anomaly
taint 5	anomaly	anomaly	anomaly	anomaly	anomaly	anomaly	anomaly	anomaly
suterasu	rootkit	rootkit	rootkit	rootkit	rootkit	rootkit	rootkit	rootkit
diamorphine	rootkit	rootkit	rootkit	rootkit	anomaly	rootkit	rootkit	anomaly

An additional ten tests where the ‘tainted’ SYSCALLs and Rootkit were randomised to observe the performance of the proposed method. In all tests, the method has consistently classified the benign SYSCALLs correctly. However, the ability to differentiate the Rootkit from anomalous ones occasionally results in the Rootkit SYSCALL being classified as an anomaly, which gives the overall average classification accuracy at 96%.

5.5 Summary

In this chapter, a lightweight technique was proposed to detect anomalies using hardware performance counters and an SVM classifier. Unlike existing methods, the proposed approach relies on a single SVM generated model to distinguish anomalous applications from a set of benign applications. Based on the benign application set, the appropriate hardware performance counters are monitored to generate the SVM model. A set of anomalous applications was also used in the training phase to improve the model. It has been shown that the proposed approach accurately differentiates between the benign and anomalous applications during the testing phase for all the test applications.

This work is then extended to apply the same technique to identify tainted SYSCALLs in the kernel-space that are maligned by a Rootkit. Instead of observing all the SYSCALLs in the operating system, the proposed method only focuses on the vulnerable ones to keep the methodology lightweight. Unlike existing work [86] that requires high monitoring overheads by deploying HPC-trace models, the proposed technique can be effectively deployed to achieve a classification accuracy rate of over 96% while incurring significantly less resources by requiring only a single SVM model.

While the proposed technique in this chapter has been shown to detect a compromised operating system successfully, it only observes the SYSCALLs. The next chapter presents an alternative technique for anomaly detection by using the hardware performance counters to extract the operating system behaviour in the presence of malware at runtime.

Lightweight Assessment of Malware for Embedded Architectures (LAMBDA)

6.1 Introduction

The previous chapters discussed various metrics for intrusion detection in embedded systems. Chapter 3 proposed using the power trace to detect an anomaly. Chapter 4 relied on SYSCALL trace to extract signatures to detect and classify user-space malware, while Chapter 5 delved deeper into the SYSCALLs themselves to detect kernel-space rootkit. This chapter proposes an alternative means to detect anomalies by focusing on specific indicators that represent operating systems. These indicators are observed during the program execution by monitoring their HPCs at the architecture level.

Recent semantic-based malware detection methods [139], [186] rely extensively on finding frequencies of specific SYSCALL patterns in program. Such methodologies may lead to false negatives, as any malware having lower deviations in SYSCALL access may easily bypass such schemes. This was discussed in Section 5.4, where it was shown that Rootkit does not affect the SYSCALL trace. However, malware detection methods using HPCs [122], [140], [141] give rise to high false positives as they are susceptible to small deviations. However, selecting the appropriate HPCs

is crucial for consistency, and high accuracy as monitoring arbitrary HPC can result in an increased false positive rate. It is also observed in Figure 5.5, that different HPCs have varying levels of sensitivity. Hence, it is noteworthy that instead of directly measuring the SYSCALL frequency or HPC for a program, measuring the effect of a program on an operating system resources is worth considering.

This chapter proposes an anomaly detection methodology that introduces a single metric λ , which measures the distance of a program from a set of known programs that are allowed to run on the OS. λ monitors the HPC to observe the interaction between the program and OS indicators that provides significant information than relying on only systems calls [139] or only HPCs [140]. LAMBDA employs statistically based *t*-tests which identify boundaries between malware from the set of benign programs. A *Sensitivity matrix* is also designed to highlight the important interactions between the OS *Indicators* and the selected HPC *Observers* for detecting potential malware. This ensures that all interactions are considered and given weight to them.

Section 6.2 introduces LAMBDA methodology, followed by a discussion of the proposed data extraction, training and detection stages for distinguishing a benign program from an anomaly. Section 6.2.1 describes the selection of Indicators and HPCs events used in this experiment. Section 6.3.6.2 provides the results and highlights the pros and cons of the approach. Finally, the proposed methodology is compared with multiple state-of-the-art, followed by a summary for this chapter.

6.2 The Anomaly Detection Methodology

This section proposes an approach of using HPC values of selected indicator program to generate a metric called λ for differentiating the benign program from anomalous ones.

The metric λ is designed by analysing the distributions of various tuples which are the combination of OS indicator programs and HPC events in a chosen platform. Figure 6.1 illustrates the training phase of LAMBDA to obtain the sensitivity matrix and threshold for the detection phase. It is observed that careful choice of these tuples provides an insight to detect the presence of potential malware on the platform. In the training phase, the tuples sensitivity are obtained by analysing the accuracy of each to classify an anomaly or benign program correctly using the t -test. The sensitivity value of these tuples is represented as a sensitivity matrix that is obtained after the training phase. Classification of a program as benign or malware, based on a specific tuple, is determined by a t -test to measure the difference between the distributions of benign programs and the given program.

Figure 6.1: Training Phase for LAMBDA

In the detection phase, when an unknown program is executed, LAMBDA classifies it based on the t -tests performed between the distributions of different program and benign. This evaluation results in the score matrix that is normalised with the sensitivity matrix to obtain a λ value. Normalizing the score matrix ensures that $0 \leq \lambda \leq 1$. The threshold, λ_t , is determined using a 3σ analysis on the λ values of the benign programs, so that when $\lambda > \lambda_t$, the program is considered as an anomaly, else a benign. Figure 6.2 illustrates the detection phase of LAMBDA that uses the sensitivity matrix and threshold for anomaly detection.

Figure 6.2: Detection Phase for LAMBDA

6.2.1 Association of Indicator Programs with HPCs

Indicators are native Linux utility programs such as `ls`, `pwd`, and `ps`. Similar to a typical program, when an *Indicators* is executed, a group of SYSCALLs are invoked as shown in Table 6.1 for ‘`ls`’ column. The frequency of invoked SYSCALLs increases when a program is executed in parallel to the *Indicators*. Table 6.1 shows the deviations in different SYSCALLs for a *benign* program ‘*Links*’ (a text-based web browser) and a ‘*Backdoor*’ malware running in the background along with *Indicators* `ls`.

However, unlike the SYSCALL trace used in Chapter 4 that extract short SYSCALL sequences, it is observed that the frequency of invoked SYSCALLs varies when any program is executed in parallel to the *Indicators*. The advantage of using an *Indicator* is that it allows the program to be analysed across multiple SYSCALLs in parallel, thus giving an improved perspective over single SYSCALL-based malware detection approach. Table 6.1 shows that the deviation in *read* SYSCALL is higher in the presence of ‘*Backdoor*’ malware for *Indicators* *ls*. Though the malware program *Trojan* does not deviate significantly for the same *Indicators*. Hence, multiple *Indicators* are deployed to extract information on the behaviour of an unknown program with respect to the effect on various SYSCALLs. For example, Table 6.1 shows that the Trojan affects the ‘*rt_sigaction*’ and ‘*close*’ SYSCALLs more if executed in parallel with *ps* instead of *ls*.

The choice of *Indicators* and its quantity are based on what OS resources need to be monitored. These OS resources are typically ‘common targets’ for malware attacks. The OS resources (file, network, process and privileged access) that were selected to be monitored are based on the findings reported in [187]. At the end of the preliminary investigation, the five *Indicators* selected are *ls*, *netstat*, *ps*, *who* and *pwd* that correspond to the chosen OS resources. Several other *Indicators*, such as *tcpdump*, *iotop*, and *lsof*, were indeed evaluated for LAMBDA. However, these indicators did not improve the detection accuracy as they were either too slow or utilise the same SYSCALLs as the selected *Indicators*. Furthermore, including more *Indicators* only leads to an increase in computational burden with minimum improvement to the framework.

In addition to choosing the most effective *Indicators*, the choice of selecting the right association between *Indicators* and *Observers* is also an important component for the detectability of potential malware. Figure 6.3 clearly shows the separability of different programs using the tuple (*ls*, *instructions*). The figure shows the

Table 6.1: Total number of different System Calls executed for the Indicator Programs *ls* and *ps* in the presence of a benign program and two different types of malwares

SYSCALLs	ls	ls + Benign	ls + Backdoor	ls + Trojan	ps	ps + Trojan
access	1	2	1	2	1	14
brk	3	4	3	4	3	14
close	7	9	9	8	145	252
execve	1	2	2	3	1	2
fstat64	5	8	5	5	5	11
getdents	-	-	-	-	2	2
getdents64	2	2	2	3	-	-
geteuid32	-	-	-	-	1	14
gettimeofday	-	-	-	-	2	4
ioctl	3	4	6	4	3	4
lseek	-	-	-	-	2	2
mmap2	12	19	12	12	12	23
mprotect	3	6	3	5	4	19
munmap	2	3	2	2	2	4
open	4	6	5	7	146	153
openat	1	1	1	2	1	1
read	3	4	260	19	145	164
readlink	-	-	-	-	8	8
rt_sigaction	-	-	-	-	23	135
set_tls	1	2	1	2	1	2
stat64	-	-	-	-	81	382
uname	1	2	1	3	1	3
Total	49	74	313	81	589	1213

Instruction Count HPC event count for three benign programs and three malware that is executed 50 times. Benign program (*flushdisk*, *gsm*, *sha*) can be seen clustered together near the bottom while the malware can be seen at the upper end of the graph. Instead then associating a higher HPC value to malware, it shows that (*ls*, instructions) is a favourable candidate tuple to include in the training phase to distinguish a benign program from a malicious one. It is also observed that 50 runs are sufficient to arrive at a consistent conclusion.

An single tuple may be insensitive thus not accurate to conclusively distinguish a benign program from an anomaly. However, LAMBDA aggregates and analyses these tuples to extract salient features to detect potential malware. Therefore, each

Figure 6.3: Behaviour of `ls`, instruction tuple

tuple is assigned a suitable weight that represents its sensitivity, which is used in the training phase. This sensitivity matrix provides a quantification of the goodness of the associations of the indicators with the *observers*.

These five selected *Indicators* are monitored at runtime by observing a set of low-level hardware events called Hardware Performance Counter (HPC) or *Observers*. The work in Section 5.2.3 focuses on selecting the HPCs, namely *Observers* based on a set of programs. However, this section focuses on selecting the appropriate *Observers* based on the five *Indicators*. This is achieved by monitoring the HPCs of the *Indicators* and using the same technique discussed in Section 5.2.3. The four selected *Observers* are *cycles*, *instructions*, *L1-cache* and *branches*.

6.2.2 LAMBDA Methodology

The proposed malware detection methodology consists of two phases, the training phase and the detection phase. In both phases, the selected *observers*' value is collected for all the *indicators* in the presence of the Program Under Test (PUT). This *observer* value is extracted using an open-source tool called PMCTrack [188] running Ubuntu's Linux OS, kernel 3.10.106 on the ODROID-XU3 platform. To obtain

all the *observer* values for each *indicator* concurrently, GNU's `parallel` tool is used in conjunction with the PMCTrack tool to support concurrency. Following subsections detail the data collection, training and detection phases. The algorithms in both the training and detection phases are implemented using python script.

6.2.2.1 Data Collection

This subsection details the process of obtaining the tuples, which are *observer* values for the *indicators*.

Let the set of indicators be denoted as $\mathcal{R} = \{r_1, r_2, \dots, r_p\}$ and the observers as $\mathcal{H} = \{h_1, h_2, \dots, h_q\}$. A total of $p \times q$ tuples which is the combination of observers and indicators, respectively. For any program, e , when the indicators r_i are executed in its presence, hardware performance counter h_j are observed.

This is repeated for s trials to minimize the effect of system noise. Let the value of h_j for k^{th} trial when indicator r_i is run in the presence of e be denoted as $\alpha_{i,j}^k(e)$. Let $\theta_{i,j}(e)$ be the distribution of h_j in the presence of both r_i and e , containing s observations, i.e., $\theta_{i,j}(e) = \{\alpha_{i,j}^1(e), \alpha_{i,j}^2(e), \dots, \alpha_{i,j}^s(e)\}$. When done for all $p \times q$ features, a distribution matrix can be obtained for executable, e :

$$\mathcal{D}(e) = \begin{bmatrix} \theta_{1,1}(e) & \theta_{1,2}(e) & \theta_{1,3}(e) & \dots & \theta_{1,q}(e) \\ \theta_{2,1}(e) & \theta_{2,2}(e) & \theta_{2,3}(e) & \dots & \theta_{2,q}(e) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \theta_{p,1}(e) & \theta_{p,2}(e) & \theta_{p,3}(e) & \dots & \theta_{p,q}(e) \end{bmatrix}$$

6.2.2.2 Training Phase

The training set is a collection of two groups of program; one contains malware, and the other contains benign programs. Let the malware set be denoted as $\mathcal{M} = \{m_1, m_2, \dots, m_k\}$ and the benign programs as $\mathcal{B} = \{b_1, b_2, \dots, b_l\}$.

In the training phase, the *Univariate t-test* that uses a two-sample *Welch's t-test* is deployed. This univariate *t-test* uses the indicator-observer tuples that detects more malware in the training phase described in Algorithm 2.

Univariate t-test: A malware m_x is detected against a benign program b_y by the tuple (r_i, h_j) , if the null hypothesis is rejected for the distributions $\theta_{i,j}(m_x)$ and $\theta_{i,j}(b_y)$. *Welch's t-test* is used for testing the null hypothesis.

A strategy to assign weights to each tuple is achieved by counting the correct classification of a malware by that tuple. If a tuple (r_i, h_j) correctly identifies malware, m_x against a benign program b_y , then its count is increased. Since, there are $p \times q$ tuples, a count matrix with p columns and q rows is obtained. The values of each cell in the count matrix range between 0 and the total number of benign and malware program combination, i.e., between $[0, k \times l]$. These counts are then normalized to obtain the univariate *sensitivity matrix* based on these tuples. An illustration of the *sensitivity matrix* ($\mathcal{W}_0$), is shown below.

$$\mathcal{W}_0 = \begin{bmatrix} w_{1,1}^0 & w_{1,2}^0 & w_{1,3}^0 & \dots & w_{1,q}^0 \\ w_{2,1}^0 & w_{2,2}^0 & w_{2,3}^0 & \dots & w_{2,q}^0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{p,1}^0 & w_{p,2}^0 & w_{p,3}^0 & \dots & w_{p,q}^0 \end{bmatrix}$$

Here, $w_{i,j}^0$ represents the weight of each indicator r_i and observer h_j association to detect a malware program at runtime.

6.2.2.3 Detection Phase

Data for an unknown program $\mathcal{T}$ is collected, and the distribution matrix $\mathcal{D}(\mathcal{T})$ is obtained as described in the previous section. This matrix is compared against all the l benign programs using the univariate *t-test* for all the $p \times q$ tuples for fast univariate analysis. This univariate analysis constructs a *Count Matrix* ($\mathcal{C}_0$) having p rows and

Data: Training Set Containing Benign ($\mathcal{B}$) and Malware ($\mathcal{M}$) Executables, List of Indicator Programs $\mathcal{R}$, List of HPC Events $\mathcal{H}$

Result: Sensitivity Matrices $\mathcal{W}_0$ for univariate analysis based on the Training Set

```

forall  $b_l$  in  $\mathcal{B}$  do
    forall indicators in  $\mathcal{R}$  do
        Collect HPC values for the events in  $\mathcal{H}$  with  $b_l$  in the background;
         $\mathcal{D}(b_l)$  = parsed values for the collected data;
    end
end
forall  $m_k$  in  $\mathcal{M}$  do
    forall indicators in  $\mathcal{R}$  do
        Collect HPC values for the events in  $\mathcal{H}$  with  $m_k$  in the background;
         $\mathcal{D}(m_k)$  = parsed values for the collected data;
    end
end
Initialize Sensitivity Matrix  $\mathcal{W}_0$  to zero for all  $(\mathcal{R}, \mathcal{H})$  tuple;
forall  $r_i$  in  $\mathcal{R}$  do
    forall  $h_j$  in  $\mathcal{H}$  do
        forall  $b_l$  in  $\mathcal{B}$  do
            forall  $m_k$  in  $\mathcal{M}$  do
                Calculate t-statistic for  $\mathcal{D}(b_l)$  and  $\mathcal{D}(m_k)$  corresponding to tuple  $r_i$  and  $h_j$ ;
                if t-statistic  $> t_{critical}$  then
                    Increment  $\mathcal{W}_0(r_i, h_j)$  by 1;
                end
            end
        end
    end
end
Normalize and return  $\mathcal{W}_0$ ;

```

Algorithm 2: Training Phase

q columns. Each cell of $\mathcal{C}_0$, i.e., the value of c_{ij} , equals the total number of times $\mathcal{T}$ is classified as known malware by the tuple r_i and h_j . Thus the value of c_{ij} ranges between $[0, l]$. An illustration of normalized $\mathcal{C}_0$ is given as below:

$$\mathcal{C}_0 = \begin{bmatrix} c_{1,1}^0 & c_{1,2}^0 & c_{1,3}^0 & \cdots & c_{1,q}^0 \\ c_{2,1}^0 & c_{2,2}^0 & c_{2,3}^0 & \cdots & c_{2,q}^0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{p,1}^0 & c_{p,2}^0 & c_{p,3}^0 & \cdots & c_{p,q}^0 \end{bmatrix}$$

Element-wise multiplication of $\mathcal{C}_0$ with the sensitivity matrix $\mathcal{W}_0$, obtained in the training phase, gives a univariate *Score Matrix*, $\mathcal{S}_0$, as shown below.

$$\mathcal{S}_0 = \mathcal{C}_0 \odot \mathcal{W}_0 \quad (\text{Eq. 6.1})$$

Data: Distributions of all Benign Executables used for training $\mathcal{D}(\mathcal{B})$, Sensitivity Matrices $\mathcal{W}_0$, List of Indicator Programs $\mathcal{R}$, List of HPC Events $\mathcal{H}$, Executable of the Program Under Test $\mathcal{T}$, Threshold λ_t

Result: Decision whether $\mathcal{T}$ is Malware or Benign

forall *indicators* in $\mathcal{R}$ **do**

    Collect *HPC* values for the events in $\mathcal{H}$ with $\mathcal{T}$ in the background;
     $\mathcal{D}(\mathcal{T})$ = parsed values for the collected data;

end

Initialize *Count Matrix* $\mathcal{C}_0$ to zero for all $(\mathcal{R}, \mathcal{H})$ tuple and *Count Matrix* $\mathcal{C}_1$ to zero for all indicators in $\mathcal{R}$;

forall r_i in $\mathcal{R}$ **do**

forall h_j in $\mathcal{H}$ **do**

forall b_l in $\mathcal{B}$ **do**

            Calculate *t*-statistic for $\mathcal{D}(b_l)$ and $\mathcal{D}(\mathcal{T})$ corresponding to features r_i and h_j ;

if *t*-statistic $> t_{critical}$ **then**

                Increment $\mathcal{C}_0(r_i, h_j)$ by 1;

end

end

end

end

Normalize $\mathcal{C}_0$;

Calculate *Score Matrix* $\mathcal{S}_0$ using Eq. 6.1;

Calculate λ_0 using Eq. 6.2;

if $\lambda_0 \geq \lambda_t$ and λ_0 not in the critical region **then**

    Return $\mathcal{T}$ as Malware;

end

else if $\lambda_0 < \lambda_t$ and λ_0 not in the critical region **then**

    Return $\mathcal{T}$ as Benign;

end

Algorithm 3: Detection Phase

Hence, $s_{ij}^0 = c_{ij}^0 w_{ij}^0$ for $i \in \{1, \dots, p\}$ and $j \in \{1, \dots, q\}$. Λ_0 is defined as the sum of all the elements in the matrix $\mathcal{S}_0$ as shown below.

$$\lambda_0 = \sum_{i=1}^p \sum_{j=1}^q s_{ij}^0 = \sum_{i=1}^p \sum_{j=1}^q c_{ij}^0 w_{ij}^0 \quad (\text{Eq. 6.2})$$

This λ_0 can be interpreted as an overall distance of the program $\mathcal{T}$ from the benign program in $\mathcal{B}$. If λ_0 is greater than some threshold value, λ_t , then $\mathcal{T}$ is classified as an anomaly. Otherwise, it is classified as benign.

An important metric in the detection phase (detailed in Algorithm 3) is to determine an appropriate threshold value, λ_t , which is used to classify if an unknown program is a benign or potential malware. In order to do that, λ_0 values for all programs is calculated in reference to benign set and a 3σ analysis is applied on the distributions

of these λ_0 values. The value of λ_t is set to $\mu + 3\sigma$, where μ and σ are the mean and standard deviation respectively of the distribution of λ_0 for benign programs. If for any program the value of λ_0 crosses λ_t , it is categorized as an anomaly.

6.3 Experimental Results

In this section, the experimental setup and the choice of benchmarks to evaluate the proposed malware detection methodology. Next, a comparative study of the proposed approach with other state-of-the-art along with various machine learning techniques is presented. Finally, the implementation overhead of the proposed method is also examined.

6.3.1 Benchmarks

162 latest ARM-based malware were collected from [121], [189] and used 50 standard Linux benchmark programs, such as CHStone [3], UnixBench [190], and LMBench [191] for our analysis. These different benchmark programs were used as a source for the benign reference point. Table 6.2 shows the separation of all the programs into *Train* and *Test* data for offline and online analysis, respectively. Next, the platform used in the proposed detection mechanism is discussed.

Table 6.2: List of Selected Malware and Benign Programs for Train and Test Sets

Category	Family	Train	Test	Total
Malware	Flooders, Trojan, Backdoor, Exploit	152	10	162
Benign	CHStone	10	2	12
	UnixBench	12	4	16
	LMBench	18	4	22
Total		192	20	212

6.3.2 Experimental Setup

In order to analyse the efficacy of the proposed method, the approach is implemented and tested on the same ODROID-XU3 platform used in Section 5.4.4.1.

The appropriate *observers* and *indicators* are selected as discussed in Section 6.2.1 that are available on the ARM Cortex-A7 processor [192] and Linux OS.

6.3.3 Test Results

This subsection experimentally validates the LAMBDA scheme based on the distributions of observers collected as mentioned above when executing indicators in the presence of selected benign program and malware.

6.3.3.1 Deriving the sensitivity matrix

The univariate t -test (described in Section. 6.2) is performed on all 40 benign programs and 152 malware during the *training phase*. If the t -value between a benign and malware distribution, corresponding to a particular tuple, exceeds 1.66 ($t_{critical}$), the null hypothesis is rejected and the sensitivity is incremented for the respective tuple. Following the procedure mentioned in Algorithm 2, the *Sensitivity Matrix* $\mathcal{W}_0$ is obtained for the training dataset, as shown in Table 6.3.

Table 6.3: Sensitivity Matrix $\mathcal{W}_0$ that is comprised of four *observers* and 5 *indicators*

Indicators	cycles	instructions	L1-cache	branches
ls	0.0274	0.0299	0.0285	0.0298
ps	0.0513	0.0515	0.0512	0.0513
who	0.0243	0.0382	0.0366	0.0405
netstat	0.0264	0.0352	0.0352	0.0351
pwd	0.0259	0.0414	0.0389	0.0392

6.3.3.2 Calculating λ value

In the detection phase discussed in Section 6.2.2.3, 20 programs were used from the dataset shown in Table 6.2. The *observers* for each program is compared against the datasets of 40 benign programs obtained during the training phase. The univariate analysis in Algorithm 3 is used to obtain the *Count Matrix* $\mathcal{C}_0$ for a program. Each cell of $\mathcal{C}_0$ represents the number of times the program is identified as malware by the corresponding tuple. The count matrices for the two programs, namely GSM (a CHStone benchmark program used as a benign) as shown in Table 6.4 and Agent (a backdoor malware) as seen in Table 6.5.

Table 6.4: Count Matrix for 'GSM'

Indicators	cycles	instructions	L1-cache	branches
ls	36	26	29	29
ps	37	32	31	40
who	41	22	31	31
netstat	33	33	14	28
pwd	37	35	29	10

Table 6.5: Count Matrix for 'Agent'

Indicators	cycles	instructions	L1-cache	branches
ls	44	39	49	33
ps	42	40	39	19
who	40	42	42	40
netstat	33	52	44	44
pwd	44	47	50	31

The λ_0 value is calculated using Eq. 6.2 for each of the program. For the executables GSM and Agent, the λ_0 values are 0.672 and 0.972, respectively. The threshold value λ_t , following the 3σ analysis, is calculated to be 0.726. Hence, GSM is classified as a benign application whereas Agent is classified as a malware.

There is a small probability that a benign program having lambda value close to λ_t may be classified as an anomaly and vice-versa. The closeness range (also known

as a *critical region*) is obtained by taking a $\pm 10\%$ limit around the threshold λ_t . The following subsection discusses the selection of an appropriate significance level α to deal with these irregularities.

6.3.3.3 Selection of Significance Level α

This subsection discusses the impact of varying significance level for the univariate t -test as an anomaly detection model. The significance level α is the probability of rejecting the null hypothesis when it is true and describes a range which defines the distance of the sample statistic from the null hypothesis value, which can be rejected. A lower significance level implies the accommodation of more benign behaviour within the critical section but with a risk of including some malware behaviour, and a higher significance level implies risking some benign programs as an anomaly. Hence, selecting a proper significance level for the t -test is important.

The detection technique was analysed with a varying confidence interval (equivalent to $(1 - \alpha)$ level) for the univariate t -test to determine the average distance between the λ values for the same set of program. Figure 6.4 shows an increase in confidence interval (i.e., by a decrease in significance level) the average distance of λ values between the same set of benign programs and malware get increased. The average distances for different confidence intervals as observed from the experiment are 0.4187 for 90%, 0.4308 for 95%, 0.4448 for 98%, 0.4536 for 99%, and 0.4727 for 99.99% confidence level. Higher distance signifies a better distinction between an anomaly and a benign program, but this does not present additional information on the rate of false positives and false negatives. Hence, the performance of the model is verified in terms of false positives and false negatives for different confidence interval before selecting one for the model.

Figure 6.4: Average gap between LAMBDA (λ_0) score for benign and malware programs under different confidence interval.

The univariate t -test is performed for 1000 iterations with randomly selected train and test data in accordance with Table 6.2. The resulting false positive and false negative percentage values are shown in Table 6.6. It is observed from the table that as the confidence interval increases, the false positives reduces accordingly, but at the expense of increased false negatives. For anomaly detection, obtaining low false negatives is more important than incurring false positives. Hence, the confidence interval of 90% is selected for the univariate t -test since it provides 0.11% false negatives with our dataset with a false positive of 2.21%.

Table 6.6: Model performance of univariate t -test for the different confidence interval

Confidence Interval	False Positives	False Negatives
90%	2.21%	0.11%
95%	2.06%	0.16%
98%	1.85%	0.19%
99%	1.75%	0.2%
99.99%	1.52%	0.22%

6.3.4 λ score for program proximity

Figure 6.7 shows the λ score for the benign programs and malware on a number line that start at 0 and ends at 1. The benign programs include some of from the training set (shown in Table 6.2) and some ‘functionally equivalent’ programs (discussed in Chapter 3). It can be observed these ‘functionally equivalent’ programs (e.g. ADPCM v1 and ADPCM v2) have lambda score that is close to each other while different programs tend to have λ scores that are further apart from each other. The programs that have been ‘patched’ (i.e. ‘patched’ jpeg and jpeg) also have λ score close to their original counterpart. These λ scores can be used to uniquely identify an unknown program is within the known benign set or is similar to them. The same method is applied to malware where each malware can be uniquely identified using their respective λ scores by comparing it with a database that lists all the λ score for the trained malware.

6.3.5 Runtime λ

Currently, LAMBDA is used to verify if a program is a benign or an anomaly as only the initial portion of the program is being tested. However, it is common for malware to infect a benign program at different stages of execution. Therefore, if the malware infects the program near the end of execution, the proposed technique classifies it as benign as the test was only performed during the ‘benign’ stages. Hence, a method to enable runtime detection is proposed by time-slicing the program at a fixed time interval and calculating the λ score as described earlier. The λ score is calculated at each time slice, as illustrated in Figure 6.5. The duration of each time interval is equivalent to the slowest *indicator* program. In this work, the slowest *indicator* program, *ps*, completes at 1ms. Hence, the time interval used is also set at 1ms and the run-time λ is calculated at each interval. This ensures that any malicious

Table 6.7: λ_0 score for benign programs and malware

behaviour that occurs at various stages of the program can be detected.

Figure 6.5: Illustrating Runtime λ

6.3.6 Evaluating the Proposed Detection Mechanism

This subsection first compares the effectiveness of our proposed approach *LAMBDA* with other state-of-the-art works for malware detection and later present the results which compare our proposed approach with other machine learning based classification methods and how these techniques perform on unknown malware.

6.3.6.1 Detecting Unknown Malware

The dataset mentioned in Table 6.2 contains ARM-based malware of type Backdoor, Trojan, Exploit, and Flooders. It does not contain any ARM-based malware of type Distributed Denial of Service (DDoS) and Virus. In this subsection, the efficiency of the proposed method is evaluated on this DDoS malware and Viruses which have not been observed previously. Five DDoS malware and five viruses that work in a Linux environment for ARM processor were selected [121], [189]. Table 6.8 presents the λ_0 values for all these malware. The threshold λ_t is determined to be at 0.6731. It is observed that all the DDoS and Virus malware are detected using the proposed approach, though $DDoS_3$, $Virus_1$, $Virus_2$, and $Virus_5$ is only classified in the later

stages of execution. The DDoS and Virus malware perform actions like increasing network activities, affecting file systems, etc. that is captured by LAMBDA to detect these malicious activities that are previously unseen and does not belong to the typical execution pattern within the set of benign programs.

Table 6.8: Analysis of *Unknown Malware*

Malware	λ	Detected Stage
$DDoS_1$	0.8743	first stage
$DDoS_2$	0.7957	first stage
$DDoS_3$	0.7848	second stage
$DDoS_4$	0.9741	first stage
$DDoS_5$	0.9146	first stage
$Virus_1$	0.8947	third stage
$Virus_2$	0.8179	third stage
$Virus_3$	0.8147	first stage
$Virus_4$	0.9319	first stage
$Virus_5$	0.7876	second stage

In addition, another type of malicious attack, Rootkit, is included to evaluate the proposed technique to detect kernel-based attacks. The dataset in Table 6.2, used in the detection phase, does not contain any Rootkit. Therefore, LAMBDA is evaluated in the presence of a Rootkit *suterasu* [193]. The λ_0 value for this Rootkit comes out to be **0.8321** that implies the presence of an anomaly. This is due to the nature of Rootkit that modifies the SYSCALL itself that makes it distinguishable by LAMBDA monitoring of *observers* and *indicators*. Hence, the LAMBDA is able to detect Rootkit that was not present in the training dataset.

6.3.6.2 Comparison with other state-of-art methods

To demonstrate the effectiveness of the proposed approach in this chapter, it is compared with the existing methods [139], [141], [194], [195], [196]. Table 6.9 shows the comparison in terms of false positives (column 2) and false negatives (column 3) between the approaches in [139], [141], [194], [195], [196] and the proposed techniques

in this chapter.

Table 6.9: Comparison with State-of-the-art

Techniques	False Positive	False Negative
GaurdOL [139]	2.7%	3%
MAP [194]	7%	6%
CONFIRM [141]	5%	8%
ROPSentry [195]	0%	0%
Elnaggar et. al [196]	9%	2%
LAMBDA (Univariate)	1.52%	0.22%

It can be observed that the proposed approach achieves better false positive and false negative as compared to the following approaches in [139], [141], [194], [195], [196]. However, the approach presented in [195] reported a better percentage of false positives and false negatives but only focused on detecting Return-oriented Programming (ROP) based attacks which are a small category of malicious threats.

6.3.6.3 Comparison with Machine Learning based Approaches

In this subsection, the approach is evaluated with existing machine learning based classifiers, like Multilayer Perceptron, Logistic Regression, Gaussian Naive Bayes, Support Vector Machine, and Random Forest Classifier. The `scikit-learn` tool on `python` was used to implement these classifiers for the comparative study. The selection of training and testing programs is done according to Table 6.2.

Figure 6.6 shows the average F_1 -scores and its variations for the different machine learning classifiers and *LAMBDA*. It is observed that *LAMBDA* has a high average F_1 score and the least variation as compared to other classifiers. This signify that *LAMBDA* performance is more deterministic in comparison to the other machine learning based methods. Table 6.10 presents the average F_1 -score and detection time for all the classifiers used in the comparison.

Table 6.10: Comparison based on Average F_1 -score, Detection time, Memory usage and CPU Utilization

Methods	Average F_1 -score	Detection time (μs)
Multilayer Perceptron	0.8654	18.01
Logistic Regression	0.8661	7.92
Gaussian Naive Bayes	0.7844	25.69
Support Vector Machine	0.8654	285.96
Random Forest	0.9795	40.09
LAMBDA	0.9854	34.43

Figure 6.6: Comparison of F_1 -score of the proposed method LAMBDA with other machine learning based classifiers in Box-plot representation

It is shown that the detection time of *LAMBDA* is not the fastest, as compared to Logistic Regression method. However, *LAMBDA* has significantly better F_1 -score. When compared to the Random Forest method that has similar average F_1 -score of above 0.9, *LAMBDA* detection time is better at $32.43\mu s$.

6.4 Summary

This chapter presented *LAMBDA*, an approach for analysing a program on an embedded platform at runtime. A new metric called λ was proposed to classify if an unknown program is benign or an anomaly. Unlike existing methods that mainly use machine learning, the proposed approach uses statistical hypothesis testing to distinguish between benign programs and an anomaly. Not only does the proposed method accurately detects anomaly at runtime, but is also able to identify them early before the malware completes its execution. As shown in Table 6.6, observing tuples, which are a combination of observers and indicators, makes λ an effective metric. It has been shown to identify benign program and malware using only the λ score. If *LAMBDA* is unable to identify the program, it can indicate if the program is close to a known benign program or malware based on the λ score. It has also demonstrated success in detecting trained and untrained malware, including Rootkit.

Unlike the existing work proposed in [139], [141], [194], and [196], the proposed method accurately detects various malware types at runtime. Additionally, the technique obtains significantly better false positive rate, at 1.52%, and a false negative rate of close to 0% when compared to the current state-of-the-art approaches. Table 6.10 shows that *LAMBDA* is also faster in the detection stages, taking only $32.43\mu s$ when compared to the widely used machine learning based techniques proposed in existing works. Not only does the method accurately detects malware at runtime, but is also able to identify them early before the malware has a chance to complete.

It should also be noted that the techniques proposed in Chapter 5, are only for SYSCALL validation. In contrast, the *LAMBDA* techniques proposed in this chapter can help to do a higher system-level validation that is effective in distinguishing a benign program from malware. Incorporating the other techniques discussed in previous chapters can help provide a holistic perspective of the entire system. The next

chapter integrates the LAMBDA methodology with all the techniques proposed in the previous chapters to design a lightweight hierarchical intrusion detection framework for real-time detection and classification of anomalies.

Hierarchical Framework for Intrusion Detection

7.1 Introduction

The primary objective of this research was to create a non-intrusive runtime multi-model intrusion detection system that can be realised on typical resource-constrained embedded systems. The previous chapters discussed various metrics such as power trace, SYSCALL trace and architecture-specific features techniques that can be used for real-time intrusion detection. These techniques presented in Chapters 3, 4, 5 and 6 rely on one of these features in isolation, and be highly effective during evaluation.

This chapter presents a sophisticated technique for integrating multiple features introduced in the earlier chapters, to propose a lightweight hierarchical multi-modal intrusion detection framework that can be realised on resource-constrained embedded platforms.

7.2 Metrics for Intrusion Detection

While the features introduced in Chapters 3, 4, 5 and 6 can be used to detect and classify malware, combining them offers immense benefit to satisfy various con-

straints such as detection accuracy, robustness and monitoring overhead. However, the large number of possible configurations pose a difficult challenge to propose an optimum methodology for integrating multiple metrics for better intrusion detection framework. The various metrics also need to be used in an ordered fashion to create an optimum detection and classification system. Moreover, the need to detect and classify at runtime is vital, since allowing the malware to have continued access can cause enormous damage to the system. Thus, all the techniques proposed in the thesis so far are geared towards runtime detection. Similarly, it is also essential for the proposed integrated framework to allow for runtime detection and classification.

Figure 7.1: Components in a Typical Embedded System

Figure 7.1 shows the three components in a typical embedded system and how the various techniques proposed in this thesis interact with these components. The

System Call Trace observes the interaction between an application and operating system while the Architecture-specific Features observes the interaction between a platform and the operating system. Power trace technique and LAMBDA captures all three components at a different perspective. Power trace observes at a ‘course-grain’ high-level while LAMBDA observes from the low-level architecture.

Next, a brief discussion of the individual metrics proposed in the earlier chapters is presented, followed by a brief discussion of their pros and cons.

7.2.1 Power Trace

In Chapter 3, a lightweight power trace based technique was proposed for anomaly detection on an embedded system. The method proposed in Chapter 3 relied on the power trace of the benign applications to train a model that can differentiate between trained benign applications and other unknown or untrained applications and malware. δ is a single metric score that is calculated at runtime using the power trace of the unknown executable and the trained model to classify the executable as either benign ($\delta \geq 1$) or anomalous ($\delta = 0$).

7.2.2 System Call Trace

In Chapter 4, an improved SYSCALL-based signature for **user-space** malware detection was proposed. The proposed method extracted a minimal set of signatures from the SYSCALL trace to classify malware into its respective family at runtime. A one-time training phase is required to extract the malware signatures needed for detection.

7.2.3 Architecture-specific Features

In Chapter 5, a technique to detect **kernel-space** anomaly was proposed that monitors the architecture-specific features through hardware performance counters (HPCs) and relies on SVM classifier for detection and classification. The proposed technique identifies the appropriate hardware performance counters, which can be monitored for specific architecture-specific features, to verify the integrity of the SYSCALLs. It has been shown that the proposed approach correctly differentiates between the ‘clean’ SYSCALLs and anomalous ones during the testing phase.

7.2.4 LAMBDA

In Chapter 6, a lightweight technique called *LAMBDA* was proposed to detect unknown anomaly on embedded systems during runtime. A new metric, λ was formulated to decide whether an application is benign or malicious. This approach shows that λ is an effective measure to observe the behaviour of a system by selecting the correct tuple of HPC events and select OS native applications. λ score can range from 0 to 1 where a score close to 0 signifies benign application and a score close to 1 denotes a malicious behaviour. The boundary threshold that separates benign from malware is derived at the training phase.

7.2.5 Pros and Cons of Features

Table 7.1 shows the strength and weaknesses of each feature-technique with respect to whether it can detect the different types of malware. It is categorised by the ability to classify or identify trained or unknown malware and Rootkit. In addition, it also divided to highlight if the malware resides in user-space or the Rootkit is present in kernel-space.

Table 7.1: Pros and Cons of Various Metrics

TECHNIQUE	OUTPUT					
	DETECTION				CLASSIFICATION	
	Trained Malware	Unknown Malware	Trained Rootkit	Unknown Rootkit	Trained Malware	Trained Rootkit
Power-Trace	✓	✓	✓	✓	X	X
System Call Trace	✓	X	X	X	✓	X
Architecture-specific Features	X	X	✓	✓	X	✓
LAMBDA	✓	✓	✓	✓	X	X

It can be clearly observed from Table 7.1 that none of the feature-technique is capable of handling all the scenarios by themselves in isolation. However, it can be seen from row 3 and row 4 that System Call Trace and Architecture-specific Features have complementary behaviour. As seen in Table 7.1, Power Trace technique has the same capabilities as LAMBDA technique. Though both are anomaly detection technique, it is seen in Table 7.2 that Power Trace incurs less overhead as compared to LAMBDA. Although, LAMBDA is capable of indicating an application similarity to the known benign set. Hence, Power Trace is selected to be used first for anomaly detection and LAMBDA at the end to classify an unknown benign application is similar to any known benign applications (i.e. patched application)

7.3 Power Trace as the First Line of Defence

Even though power is monitored frequently to obtain the executable's power trace or profile, the monitoring overhead incurred is significantly lower than a hardware performance counter (HPC)-based technique, as seen in Table 7.2. Among the various features discussed, power-trace has the least monitoring overhead and hence offers an effective means to detect a range of abnormal behaviours. Therefore, the power-trace based technique proposed in Chapter 3 was chosen to detect anomaly as the first line of defence. The proposed method samples the 'instantaneous' power of the runtime application and compares it with the model stored in the database, created

offline (see Section 3.2.2).

Table 7.2: Overhead of using HPC vs Power

Feature	Register Read	Time Interval	CPU Utilization
Hardware Performance Counter	4	1ms	15%
Power	1	300ms	2.5%

Next, the method calculates a single metric δ that is used to determine if the running application is benign or anomalous. This process is repeated periodically at a pre-defined interval. If the calculated δ score determines the application to be anomalous for three consecutive intervals, an alarm is triggered. This technique has been shown to not only differentiate between different applications but also detect subtle changes in applications, e.g. ‘functionally equivalent’ application (as discussed in Section 3.3.2) . The strategy not only allows detection of anomalous applications at runtime but also minimises the number of false alarms.

7.3.1 Monitoring power at runtime

There is an increasing number of platforms [1] [103] that are incorporating high sensitivity ‘current sensors’ to provide for a finer granularity of monitoring power during application runtime. Figure 7.2 shows the high-level block diagrams of ODROID-XU3 and EFM32 development platforms, both having integrated current sensors on-board that allows the processor to access the current and voltage measurement via reading the appropriate registers. Monitoring current and voltage on the main processor incurs lower overhead as typically a simple register read is required to obtain the power.

7.2.a: ODROID-XU3 power extraction

7.2.b: EFM32 power extraction

Figure 7.2: Power extraction on existing embedded platforms

The sampling frequency depends on the type of current sensor available on the platform as well as its interface with the processor. The EFM32 platform allows the sampling to be done every $100\mu s$ whereas ODROID-XU3 power profile can be sampled in the millisecond's range due to the latency of I2C communication to obtain these measurements.

7.4 Integrating Power Trace with System Call Trace for Resource-efficient Classification of User-space Malware

Any alarm triggered by the power-trace based anomaly detection system, as described in the previous section, indicates that an abnormal behaviour has been detected. However, using the power profile alone is insufficient to classify what kind of threat has occurred. Therefore, another feature is incorporated to allow the framework to classify the malware threat using signatures. Existing anti-virus solutions that use signatures for malware detection require significantly large storage space due to the sheer number of malware signatures that need to be stored. In contrast, Chapter 4 proposed a technique that allows malware identification via the SYSCALL trace at runtime, using only a limited set of signatures.

Figure 7.3 shows the usefulness of the proposed technique by evaluating it using a benign application (SHA) that has been infected with malware. As can be seen in this figure, the signature was detected well before the application has the chance to complete, thereby mitigating the potential damage that could have otherwise occurred. Section 4.4.1 details the proposed method of classifying malware based on their inherent common signatures. This is done by generating a unique set of signatures for each class of malware.

Moreover, Section 4.5.1 explains how additional signatures can be generated to classify malware at different malware stages of execution thus allowing early malware detection at runtime as seen in *point #4* in Figure 7.3.

The proposed dual-feature method as shown in Figure 7.3 can essentially detect and classify an application infected with malware at runtime. If the method detects an anomaly using the power profile, the SYSCALL trace would be enabled and begins

Figure 7.3: Malware Detection using Power Trace and System Call Trace at run-time

looking for malware signatures generated offline using the classification techniques detailed in Chapter 4.

7.4.1 Obtaining System Call Trace

System calls are common functions or libraries offered by the operating system for applications. They are basic functionality that all applications use extensively. To detect malware signatures at runtime, the SYSCALL trace of an application needs to be extracted at runtime. This sub-section discusses the various hardware and software methods to obtain the SYSCALL identification.

7.4.1.1 Software Techniques for System Call Identification

- (i) **Using Hook Function:** One method of identifying the SYSCALL is by utilising the 'hook' function available in operating systems. In this method, each time a SYSCALL is invoked, the 'hook' function calls a small custom module to extract the SYSCALL ID (which uniquely identifies the SYSCALL). The identified

SYSCALL can then be used to look for the malware signatures.

- (ii) **Using Trap Instruction:** This method extracts the SYSCALL ID with the assembly language opcode of `trap`. SYSCALLs in OS kernel are generally implemented using `trap` to execute code in a kernel-space to allow manipulation of critical resources. Therefore, when a `trap` occurs, the SYSCALL ID can be extracted that is used to identify the SYSCALL. The identified SYSCALL can then be used to look for the malware signatures.

Software technique can be used to obtain the SYSCALL trace. However, it requires instrumentation of the operating system. In the next sub-section, a hardware-based method is proposed that would allow SYSCALL extraction to be performed without the need to instrument the operating system nor applications.

7.4.1.2 Hardware Techniques for System Call Identification

- (i) **Using Soft-core Processor:** The Terasic's SoCKit (as discussed in Section 5.3.1), running Altera's hybrid FPGA/ Cortex-A9 microprocessor, has been configured to extract SYSCALL trace at runtime. The reconfigurable space in the FPGA has been configured with a SPARC-V8 LEON3 soft-core processor to act as the main processor that allows its execution trace to be extracted, filtered and buffered (as discussed in paragraph 4, Section 5.2.1). The ARM Cortex-A9 processor on this SoCKit platform is used to monitor and analyse the trace buffer for the SYSCALL trace. The monitoring algorithm is executed bare metal on the Cortex-A9 processor.
- (ii) **Using HPC:** In ARM-based platforms, HPCs are configured and managed by the performance management unit (PMU). The PMU requires the interrupt service routine to be registered to handle performance monitoring interrupts (PMI).

The interrupt frequency needs to be set by initialising a performance counter to the desired value. The event selection filter is set that specifies the event to count. Final process is to enable the HPC and interrupts for the selected HPC.

A technique in [2] propose using HPC that correspond to `EXC SVC` and `EXC SMC` events are leveraged to extract the SYSCALL. Both these counters are available in both the ARMv7 and ARMv8. A challenge that could occur is instruction skid that refers to the number of instructions that are executed past an instruction which triggers a counts overflow before the PMI halts execution. The interrupt shadow refers to these instructions collectively, and any HPC event that occurs within this shadow is missed. Fortunately, SYSCALL instructions occur far less frequently than many other codes, thus losing a single SYSCALL invocation due to an interrupt shadow is found to be rare occurrence which was verified by checking the counter value in the ISR upon overflow.

7.5 Integrating Power Trace with Architecture-specific Features for Lightweight Kernel Verification

Chapter 5 showed that architecture-specific features based anomaly detection and classification technique allows for better accuracy in detecting kernel-space malware. However, the significant monitoring overhead imposed on the CPU makes HPC based anomaly detection untenable in practice, especially in resource-constrained embedded systems. However, if HPC based verification occurs only when required (such as only when an anomaly is detected), then the monitoring overhead incurred can be reduced significantly. Such an outcome can be effectively achieved by combining the low overhead Power Trace based detection technique with the Architecture-specific Features based anomaly detection and classification technique. Together, the two metrics allow for highly effective anomaly detec-

tion and classification system while achieving a low overall monitoring overhead. The rest of the section explains this integrated technique.

As discussed in Section 7.3, if an alarm is triggered by the `Power Trace` based anomaly detection system, this indicates that an abnormal behaviour has been detected. However, using the power profile alone is insufficient to classify what kind of threat has occurred. Therefore, another feature is incorporated to allow a dual-feature technique to detect threats to the kernel-space using HPCs. The kernel is comprised of many SYSCALLs, which are essentially `common libraries` used by application to perform specific action. Malware attacks with an objective to compromise the OS's kernel, targets these SYSCALLs to allow the OS to perform the malicious action that the malware requires at an elevated privilege-level. Chapter 5 proposed a technique that performs anomaly detection and classify the malware using HPC. This is done by monitoring selected HPC events, each time a SYSCALL is executed. The HPC events are then collected and compared to the model trained offline to detect whether the SYSCALLs have been compromised.

Figure 7.4 shows the usefulness of the combining the `Power Trace` based technique to the HPC based anomaly detection and classification technique by evaluating it using a benign application (SHA). The operating system was infected with a 'suterusu' Rootkit at `time interval 21`. As can be seen in this figure, the anomalous SYSCALL was detected well before the application has the chance to complete, thereby mitigating the potential damage that could have otherwise occurred. In addition, using the techniques proposed Section 5.4, this malware can be classified using the machine learning classifier. This technique is viable as there is a limited 'families' of kernel-space malware (e.g. Rootkits) [185], [197].

Figure 7.4: Detecting Rootkit using Power Trace and Architecture-specific Features at runtime

7.5.1 Obtaining Architecture-specific Features

Architecture-specific features are obtained through hardware performance counters (HPC) that are accessed by configuring and reading specific processor registers. Therefore, HPC can be controlled by any method that allows registers to be configured that includes operating system and hypervisor. This subsection discusses the various hardware and software methods to configure the HPCs.

As discussed earlier in Chapter 5, HPC counts are obtained by simply reading the appropriate registers in the processor. However, the frequency of reading and processing these counters for runtime usage may incur significant overhead [173][174].

Section 7.4 described how to obtain SYSCALL trace using HPC. Building on this, SYSCALLs that are vulnerable to kernel-based malware can be monitored. All modern microprocessor, i.e. ARM Cortex-A series, have a wide range of HPC events to choose from via registers.

7.5.1.1 Software Techniques for Monitoring Architecture-specific Features

- (i) **Using PMCTrack:** A tool called PMCTrack [188] is used to obtain event counts for the program under test (PUT). PMCTrack is an OS-oriented PMU tool for the Linux kernel. It monitors module abstraction, a platform-specific component that is responsible for collecting the necessary high-level metrics that a given OS scheduling algorithm requires to function. This abstraction makes it possible to create architecture-independent implementations of OS scheduling algorithms that leverage PMU data. Mostly, the scheduler does not access or deals with performance counters or hardware events directly, but instead uses the PMCTrack kernel API to retrieve the necessary per-thread or per-application performance metrics from the underlying monitoring module. Despite being an OS-oriented tool, PMCTrack is also equipped with a set of command-line tools and userspace components to assist OS-scheduler designers during the entire development process. These user-space tools complement existing kernel-level debugging tools with PMU-related offline analysis and tracing support. Figure 7.5 illustrates the monitoring algorithm and PMCTrack in the OS.

Figure 7.5: High-level structure of incorporating HPC into OS

7.5.1.2 Hardware Techniques for Monitoring Architecture-specific Features

- (i) **Using Soft-core Processor** : The ARM Cortex-A9 processor is used to monitor and analyse the selected HPCs at runtime. Extending the LEON3 soft-core processor design (illustrated in the previous sub-section) of using a dedicated hardware processor to extract SYSCALL ID. Once the SYSCALL has been identified, the corresponding HPC registers can be configured and enabled to start monitoring the events. This monitoring process imposes no overhead onto the soft-core processor as the monitoring algorithm is executed on a separate Cortex-A9 processor. Moreover, a dedicated monitoring system ensures anomaly detection even if the main processor being monitored is compromised. Figure 7.6 illustrates how the proposed multi-processor framework is designed in such a way that the Cortex-A9 processor via the HPS2FPGA bridge has access to the LEON3 system address space through the AHB2AXI/AXI2AHB bus and can extract the events count at runtime with minimally overhead to the LEON3 soft-core processor. The applications are executed on the LEON3 soft-core processor that becomes the main processor while the Cortex-A9 processor is configured for runtime anomaly detection.

Figure 7.6: Cortex-A9 processor monitors hardware events of LEON-3 processor

- (ii) **Using HPCs:** Extending the HPC-based method described in Section 7.4 that allows the SYSCALL to be identified at runtime. Upon the supervisor call instruction being trapped by the PMU (e.g. a *read* SYSCALL), the saved program counter (PC) can either ‘land’ within vector swi, the entry point of *read* SYSCALL, or is somewhere past the *read* SYSCALL entry point. Figure 7.7 shows that about 95% of the time, the PC would land before the SYSCALL allowing time for the HPC to be configured and enabled. In total, trapping system-wide supervisor call instructions via performance interrupts added 2% overhead.

Figure 7.7: The frequency associated with three possible values of PC upon trapping SVC instruction via PMU and isolating the *read* SYSCALL [2].

7.6 Lightweight Integrated Technique for Intrusion Detection

Sections 7.4 and 7.5 highlight the benefits of integrating the ultra-lightweight Power-trace based anomaly detection technique with SYSCALL and HPC based user-space and kernel-space anomaly detection and classification techniques respectively. In both these sections, the main aim was to use power-trace based detection technique perpetually at runtime and only call the more complex SYSCALL and HPC based techniques, when an anomaly was detected. This technique helped to reduce the overall monitoring overhead of the anomaly detection and classification framework.

This section proposes to combine these three features, namely power trace, SYSCALL and HPC for a more comprehensive intrusion detection framework, while leveraging on the advantages of each of these features.

Next, the complementary nature of SYSCALL-based and architecture-specific features-based techniques is highlighted, before describing the integrated framework combining the three features to bolster the intrusion detection system from both user-space and kernel-based malware while still being lightweight.

7.6.1 System Call Trace vs Architecture-specific Features

As shown in Chapter 4, SYSCALL trace, when used as a feature, can be effectively used to detect and classify user-space malware, even at runtime. However, such features cannot be used to detect anomalies that modify the SYSCALL itself, e.g. Rootkits. Figure 7.8 shows the power trace of a benign application being executed and at time interval 21, a ‘suterusu’ Rootkit was injected into the system. As seen in the figure, power trace method detects the Rootkit and triggers an alarm after a pre-defined interval, as discussed in previously in Section 7.3. Once the alarm

is triggered, the framework starts logging the SYSCALLs. The SYSCALL trace is used to look for known malware signatures (as described in section 4.3.4). However, since Rootkit do not change either the SYSCALL IDs or their flow, the SYSCALL trace remains unchanged. In other words, the program trace continues to emulate a benign application. This behaviour is highlighted in Point 4 of Figure 7.8 to show that the SYSCALL-based malware detection and classification system is unable to identify any malware in this example application that has been modified by a Rootkit.

Figure 7.8: Power Trace of an Application during a Rootkit Attack

The inability of the SYSCALL-based techniques to detect anomalies caused by Rootkits can also be confirmed by counting the number of SYSCALLs in the presence and absence of Rootkits. It can also be observed in Table 7.3, which shows each application's total SYSCALL count, that the total SYSCALL count remains unchanged in the presence of a 'suterusu' Rootkit.

Table 7.4 lists the different possibilities that can be achieved by combining the power trace and SYSCALL-based techniques. It can be observed that by using the power trace and SYSCALL-based methods, the framework can detect an anomaly

Table 7.3: Total System Call with and without 'suterusu' Rootkit

Application	without Rootkit	with 'suterusu' Rootkit	change (%)
LS	259	259	0
PS	1516	1516	0
WHO	168	168	0
NETSTAT	22	22	0
PWD	92	92	0

Table 7.4: Detection and Classification Truth Table

Power	System Call	Output
✓	X	User-space and Kernel-space Malware Detected
X	✓	User-space Malware Detected & Classified
✓	✓	User-space and Kernel-space Malware Detected, User-space Malware Classified, Kernel-space Malware not Classified

and even classify user-space malware. As discussed in Section 4.4.1, user-space malware is malicious code that executes at user-privileged-level in the operating system, as opposed to kernel-privileged-level. However, as power detects an anomaly, yet no signatures were found leads to a high probability of an attack occurring within the kernel-space (i.e. Rootkit).

Chapter 5 showed that HPC-based technique could be used effectively for operating system verification. This was achieved through verification of the SYSCALLs in the kernel. Hence, in the next subsection, the architecture-specific feature-based method with the anomaly detection framework using power trace and SYSCALL trace is combined, as discussed previously in Section 7.4.

7.6.2 Combining Architecture-specific Features with System Call Trace and Power Trace Metric

Figure 7.9 demonstrates the flow of the proposed multimodal intrusion detection framework. The following points details the actions are taken when a trigger event occurs:

Figure 7.9: The Proposed Framework using Multiple Metric

1. When the application is started, the trained power trace model is obtained from the database and compared with the current runtime power profile.
2. If an anomaly is detected, the algorithm verifies it by calculating the δ score for three consecutive time interval before triggering an alarm if all three confirms the anomalous behaviour.
3. The alarm triggers the search for malware signatures using SYSCALL trace and SYSCALL verification using architecture-specific features monitoring each time a SYSCALL is invoked.
4. Malware signatures found in the SYSCALL trace indicates that a malware has been detected and classified.
5. To establish the operating system is not compromised, each SYSCALL invoked is verified. Different set of HPCs needs to be monitored for each SYSCALL. Thus, once the SYSCALL is identified, the corresponding HPCs

are enabled and monitored. At the end of the SYSCALL execution, the HPC values are tallied and compared with the model generated offline to classify it as either 'clean', anomalous, or Rootkit.

By capturing the different kernel-based malware behaviours, the proposed method is capable of not only detecting a 'clean' SYSCALL but also a compromised SYSCALL and identify the malware. If the model is unable to determine the malware, it is still categorised as an anomalous behaviour (abnormal behaviour) which suggests an unknown attack that can be included in future training model.

7.7 Incorporating LAMBDA for Robust Anomaly Detection and Classification

It has been shown in the previous sections that using a systematic approach to integrate power trace, SYSCALL trace and hardware performance counters can provide for detection and classification of attacks coming from user-space as well as the kernel-space.

However, the approach is very tightly modeled for *known* benign applications and malware. This can potentially lead to increased miss-classification due to emerging new forms of attacks that may not have been modelled. In Chapter 6, a novel approach called LAMBDA is introduced that captures the intrinsic behaviours of an application by observing the HPC values of OS indicators at runtime. Leveraging this tuple, the proposed technique introduces a metric λ , to decide whether an unknown application is benign or anomalous at runtime.

In addition, as shown in Section 6.3.4, the λ score of an unknown application could also be compared to the database of the λ scores of known applications and malware. This helps to identify if the unknown application matches any benign application or

malware used during the training phase. In situations, where the λ score does match any benign or malware from the training set, it can also help to detect if the behaviour of an unknown application is similar to any trained benign application or malware (i.e. patched application, variant of a trained malware). This additional information could not be obtained from techniques discussed in the other chapters.

Figure 7.10: Holistic Systems Monitoring with Robust Malware Classification and System Fault Detection

Figure 7.10 illustrates an example on how combining the two metrics proposed in the previous chapters, namely, the System Call Trace and LAMBDA can be used to robustly detect and confirm the presence of malware in the system. As explained in Chapter 6, the LAMBDA technique can effectively detect and identify malware during

runtime. Similarly, as described in Chapter 4, SYSCALL trace method can also be relied upon to detect and classify malware at runtime. Hence, the outcome from both these techniques can be leveraged simultaneously to validate the presence of malware in the system with a very high degree of confidence. Hence, the combined approach provides for a robust malware detection and identification while still being lightweight. Additionally, integrating the technique offers an insight that goes beyond benign and malware detection. For example, if the LAMBDA technique highlights the presence of an anomaly while the System Call Trace based technique is unable to find any malware signature, then this may indicate a system fault. Similarly, if the System Call Trace based technique can detect a malware but unable to classify into a specific category and the LAMBDA technique can only confirm an anomaly but not specific malware, then this may indicate the presence of probably a malware that is unknown or untrained.

7.7.1 Obtaining HPC Concurrently for Multiple Indicators

Section 7.5.1.1 details how HPC events can be monitored using PMCTrack. This tool is also capable of concurrently monitoring multiple applications in a Linux OS environment. PMCTrack's monitoring allows fine-grained control with regards to in-kernel event multiplexing. Moreover, when the number of hardware events to be monitored exceeds the number of performance counters available in a processor core, time multiplexing is used. Therefore, different subsets of events can be monitored in a round-robin fashion.

7.8 Framework for an Integrated Intrusion Detection System

This section presents a systematic approach for integrating multiple metrics for a comprehensive intrusion detection system. Figure 7.11 illustrates the proposed real-time multi-modal intrusion detection framework that combines all the metrics discussed in this thesis. The previous sections have illustrated how each technique incorporated into the framework improves its detection accuracy and coverage. This section describes the integrated framework that incorporates all the metrics of intrusion detection presented in this thesis and show its effectiveness through several case studies.

Figure 7.11: Framework Overview for an Integrated Intrusion Detection System

7.8.1 Leveraging HPCs for Non-intrusive Monitoring

It was described in the previous sections the various ways to monitor and obtain the corresponding features. Some of these technique require instrumenting the operating system and/or the application while others require dedicated code to use the HPC. However, besides power trace, the other features can be obtained using HPCs. In Section 7.4 details how SYSCALL ID can be extracted using HPC events that cor-

respond to ‘exception taken for supervisor call’ (EXC_SVC) and ‘exception taken for secure monitor call’ (EXC_SMC) events.

Moreover, SYSCALLs that are observed for kernel verification are then monitored meticulously using the selected HPCs (as described in Section 5.4) during training phase. Finally, Section 7.7 details how LAMBDA uses time-multiplexing to obtain the HPC count for five indicators programs concurrently. These HPC counts are used to calculate the λ score to determine if the running application is benign or an anomaly.

Implementing these three techniques with HPCs assist in eliminating OS and application instrumentation. This makes the proposed framework scalable because the OS and applications does not need to be instrumented.

Table 7.5 lists the various HPCs used in the different stages of the integrated framework. The selection of these HPCs is based on the techniques proposed in Section 5.2.3 of Chapter 5.

Table 7.5: List of HPCs used in the integrated framework

Techniques	Selected HPCs
System Call Trace	EXC_SVC (Exception Taken, Supervisor Call)
	EXC_SMC (Exception Taken, Secure Monitor Call)
Architecture-specific Features	LD_RETIRE (Instruction architecturally executed, condition code check pass, load)
	ST_RETIRE (Instruction architecturally executed, condition code check pass, store)
	BR_MIS_PRED (Mispredicted or not predicted branch speculatively executed)
	L1I_CACHE (Level 1 instruction cache access)
LAMBDA	CYCLES (CPU Cycles)
	SW_INCR (Instruction architecturally executed, condition code check pass, software increment)
	L1I_CACHE (Level 1 instruction cache access)
	BR_IMMED_RETIRE (Instruction architecturally executed, immediate branch)

7.8.2 Multiplexing HPCs Features for Non-intrusive Monitoring

It was discussed in the previous section that HPC is a good idea for non-intrusive monitoring. Ideally, the three techniques, namely, SYSCALL trace, architecture-specific features and LAMBDA should be performed concurrently. However typically many existing embedded systems do not allow monitoring of many HPC concurrently at runtime. For example, ARM Cortex-A processor allows only four HPCs to be monitored concurrently. Hence, in order to obtain the HPCs for monitoring SYSCALL trace, architecture-specific features and LAMBDA simultaneously, a multiplexing technique is proposed to obtain all the required HPCs at runtime.

Figure 7.12: Multiplexing Multiple HPC-based Techniques

Figure 7.12 illustrates how multiplexing is used to obtain the necessary HPCs counts. This implementation is further simplified as the SYSCALL trace and LAMBDA techniques are also time-sliced to support real-time detection.

Once power trace detects an anomaly, the other techniques are scheduled in a round-robin pattern to obtain the necessary features via HPCs. Overall analysis can

be performed once all four results (as shown in Figure 7.12, point A), one for each technique are reported. Subsequent overall analysis can be performed at each time interval where the result for each technique is updated.

7.8.3 Integration for Ease of Deployment

The proposed scheme can be implemented on a multi-core processor, where one of the core is dedicated to monitor the OS and applications running on the other cores. Figure 7.13 shows a possible implementation of the proposed intrusion detection system on a Samsung Exynos 5422 architecture where one core is monitoring the power trace as well as the HPCs for the remaining seven cores.

Figure 7.13: A single core dedicated for monitoring the HPCs and Power Trace for the other cores

In an effort to avoid the need for instrumentation of the operating system or the applications, an embedded hypervisor that resides in between the hardware layer

and the operating system can be deployed for the monitoring task. Several critical systems such as automotive, medical and industrial systems [198] already employ embedded hypervisor to create an abstraction layer between the underlying hardware and the operating systems running on top. A type-1 embedded hypervisor executes in privileged mode and has full control of the hardware. It allows multiple guest OS and bare-metal application to execute on the hypervisor in unprivileged mode. All instructions from the guest operating system go through the embedded hypervisor as hypercalls to gain access to the underlying hardware.

As shown in Figure 7.14, the intrusion detection framework runs on the monitoring core via the embedded hypervisor to observe and analyse the specific metrics such as power trace and SYSCALL trace, LAMBDA etc. through HPCs as discussed earlier. As the monitoring core is managed by the hypervisor and not the guest OS, any attack originating from the guest OS does not circumvent the intrusion detection capabilities of the system. In addition, since the application and guest OS need not be instrumented in the proposed setup, new guest OS can be added easily in the system if needed. This makes the solution scalable and straightforward to implement.

Figure 7.14: Hypervisor used to realise a monitoring solution for Intrusion Detection Framework

The next subsection presents a detailed discussion with case studies on reaching an outcome given the various metric from each technique.

7.8.4 Multi-modal Analysis

It was discussed in the Section 7.3 that power trace is capable of detecting anomalies while incurring a low monitoring overhead. Therefore it has been selected as the primary feature for anomaly detection. Next, Section 7.4 proposed a methodology using SYSCALLs that generates a smaller number of signatures to identify as well as classify malware at runtime. Subsequently, in Section 7.6, it was shown that HPC-based models can be used to identify kernel-based attacks, i.e. Rootkit. Lastly, the LAMBDA technique discussed in Section 7.7, can be used to cast a system-wide ‘net’ to monitor for any unwanted behaviours and highlight if a benign application or malware is similar to any known benign application or malware.

Table 7.6: Extended Truth Table for Multi-Modal IDS

#	Power Trace	System Call Trace	Architecture-specific Features	LAMBDA	Outcome
1	benign	-	-	-	trained benign
2	anomaly	no signature	clean	benign	patched benign or unknown benign
3	anomaly	malware	clean	malware	malware identified
4	anomaly	unclassified malware	clean	anomaly	unknown malware
5	anomaly	no signature	anomaly	anomaly	unclassified rootkit
6	anomaly	no signature	rootkit	anomaly	rootkit identified
7	anomaly	malware	clean	anomaly	malware identified
8	anomaly	no signature	clean	anomaly	system fault

Table 7.6 summarises the various permutations of features that may be possible in the proposed multi-model intrusion detection system. It shows the decision outcome of the monitoring module if one or more of these features detect malicious activities

in the system. Since, power trace is selected as the primary feature, the next stage of the monitoring module is only triggered when the power trace detects an anomaly. SYSCALL trace are used to classify user-space malware while HPCs are specifically targeted to identify kernel-space (i.e. Rootkit) malware. Finally, the LAMBDA methodology is relied upon to detect unknown malware that were not seen during the training phase. In addition, the LAMBDA process helps to detect if an unknown application at runtime is closely resembles any of the known benign or malware used during the training phase.

Each of the features described in the earlier sections, provide different and complementary information on the intrusion's behaviour to the overall Intrusion Detection System. *Power Trace* method categorised if the running application is a benign or an anomaly. *System Call Trace* technique classifies the application as malware or probable malware. *Architecture-specific Features* verifies if the SYSCALL is clean or if it is an anomaly or a known Rootkit. Finally, *LAMBDA* categorised the application as either benign, benign-like, a trained malware or an anomaly. Therefore, combining these techniques offer a broader view that provides a deeper insight into the intrusion and the precise actions to be taken.

7.8.5 Performance Evaluation

In this sub-section, many case studies will be undertaken to evaluate the performance of the various algorithms proposed in this thesis to detect anomaly, malware, Rootkit and system faults in embedded systems. Table 7.7 lists the applications and malware that were used for the training and testing phases.

The rest of this subsection discusses how the proposed framework reacts to each test case and its corresponding scenarios listed in Table 7.6. As discussed in Section 7.3, the δ refers to power trace score that would classify if the executable is

Table 7.7: Testing and Training Sets for Performance Evaluation

TYPE		APPLICATION
Training	benign	AES, SHA V1, PNS, CRC32, ADPCM, GSM, DFADD, MIPS
	malware	Viruses, Trojans, diamorphine & suterusu Rootkit
Testing	benign	AES, BITCOUNT, patched SHA V1, SHA V2
	malware	infected SHA V1, Viruses, Worms, umbreon & suterusu Rootkit

benign ($\delta > 1$) or anomalous ($\delta = 0$). Also, as discussed in Section 7.7, λ refers to a score that classifies the executable as benign or anomalous. λ score can range from 0 to 1 where a score close to 0 signifies benign application and a score close to 1 denotes a malicious behaviour. The boundary threshold that separates benign from malware is derived during training phase.

Table 7.8: Training Phase Dataset

TECHNIQUE	BENIGN	MALWARE
Power Trace	✓	X
System Call Trace	✓	✓
Architecture-specific Features	✓	✓
LAMBDA	✓	✓

Table 7.8 shows the dataset used during the training phase for each technique. Power-trace based technique generates a model that tightly represents the known benign applications. Thus, it only uses the benign application dataset during the training phase. On the other hand, the SYSCALL-based anomaly detection and classification system extracts the malware signatures, but requires benign application data to reduce false positive. Hence, both benign applications and malware are used in this technique. HPC based technique models the operating system's SYSCALL before and after being infected by the kernel-space malware. Therefore, it requires both the 'clean' SYSCALL and the Rootkit. Finally, LAMBDA requires both benign applications and malware to generate the model and threshold score that is used to differentiate if the running executable is benign or malware.

- **Case Study 1:** As shown in Table 7.7, the AES application was used during the

training phase of the proposed IDS. During runtime, when the AES application is executed, the corresponding power profile matches the one stored in the database during the training phase. Hence, the δ score does not trigger an alarm in the system. This behaviour corresponds to Scenario 1 of Table 7.6. Therefore, as expected, the other three stages are not triggered. This indicates that the executable is a ‘known benign’ application.

- Case Study 2:** As shown in Table 7.7, the BITCOUNT application was not included during the training phase of the proposed IDS. During runtime, when the BITCOUNT application is encountered, the corresponding power profile does not match any profile stored in the database. Hence, δ score triggers the alarm early in the execution. The alarm triggers the other three stages to start monitoring the system. As this is an ‘unknown’ but ‘benign’ application, no malware signatures and anomalies at kernel-space can be detected. In addition, λ score indicates that the application is a benign, but is unable to identify the application. This behaviour corresponds to Scenario 2 of Table 7.6. This indicates that the executable is an ‘unknown benign’ application.
- Case Study 3:** As shown in Table 7.7, the patched SHA application was not included during the training phase of the proposed IDS. However, the original SHA application was indeed used during the training phase. At runtime, when the patched SHA application is encountered, the corresponding power profile mismatches the SHA profile midway through execution. Hence, δ score triggers the alarm midway through execution which triggers the monitoring of the other three stages. However, as the patch is not malicious, no malware signatures nor kernel-space anomaly is detected. In addition, the calculated λ score signify that the patched SHA is a benign application whose score is close to the SHA λ score stored in the database. This behaviour corresponds to Scenario 2 of

Table 7.6. This indicates that the executable is an unknown benign application that exhibit similar behaviour to the known SHA application.

- **Case Study 4:** As shown in Table 7.7, the SHA V2 application was not included during the training phase of the proposed IDS. However, the original SHA application was used during the training phase. At runtime, when the SHA V2 application is encountered, the corresponding power profile does not match any of the power profiles stored in the database. This case study follow closely to Case Study 3 as described earlier. The only difference is that the λ score is further away from the trained SHA application. This is because the SHA V2 application is implemented differently, even though the output is the same as the original SHA given the same input. This behaviour corresponds to Scenario 2 of Table 7.6. This indicates that the executable is classified as an ‘unknown benign’.
- **Case Study 5:** As shown in Table 7.7 and explained in Table 7.8, the Virus malware application was used during the training phase of the proposed IDS for the SYSCALL trace technique, but not for the power profile phase. Hence, during runtime, when the Virus malware is encountered, the corresponding power trace does not match any profile in the database. Hence, δ score triggers the alarm early in the execution. The alarm triggers the other three stages to start monitoring the system. As this is a known malware, malware signatures were detected that identifies the malware, while no anomaly were found at kernel-space. The calculated λ score classifies the executable as malicious and the corresponding λ score is used to identify the malware from the database. This behaviour corresponds to Scenario 3 of Table 7.6. This indicates that the executable is a known malware.
- **Case Study 6:** As shown in Table 7.7, the Worm malware was not included during the training phase of the proposed IDS. During runtime, when the Worm

malware application is encountered, the corresponding power profile does not match any profile in the database. Hence, δ score triggers the alarm early in the execution. The alarm triggers the SYSCALL method that is unable to classify the malware. The kernel-space verification assures that the SYSCALLs are 'clean' to signify it is not compromised. However, λ score classifies it as an anomaly. As both δ score and λ score classifies this executable as anomaly and SYSCALL trace can only detect it as an unclassified malware, this indicates that it is an unknown user-space malware. This behaviour corresponds to Scenario 4 of Table 7.6.

- Case Study 7:** As shown in Table 7.7, the AES application was used during the training phase of the proposed IDS. However, an Umbreon Rootkit that was not included during the training phase of the proposed IDS is 'inserted' into the OS immediately after AES application is started. At runtime, when the AES application is encountered, the corresponding power profile does not match the AES profile during execution. Each time AES uses a SYSCALL, the Umbreon Rootkit intercept it during execution and call the malware's modified SYSCALL. Hence, δ score triggers the alarm that enables the monitoring of the other three stages. As AES is a benign application, no malware signatures can be found. However, the kernel verification detects an anomaly that indicates the attack originates from the OS. This is verified by the λ score that indicates an anomaly. The λ score do not correspond to any known malware on the database. This behaviour corresponds to Scenario 5 of Table 7.6. This indicates that the attack is an unknown Rootkit.
- Case Study 8:** As shown in Table 7.7, the AES application and suterusu malware were included during the training phase of the proposed IDS. At runtime, when the AES application is encountered, the corresponding power profile does not match the AES profile during execution. Hence, δ score triggers the

alarm that enables the monitoring of the other three stages. As AES is a benign application, no malware signatures can be found. However, the kernel verification classifies the anomaly as a Rootkit. This is verified by the λ score that detects an anomaly. The λ score do not correspond to any known malware on the database. This behaviour corresponds to Scenario 6 of Table 7.6. This indicates that the attack is a known Rootkit.

- Case Study 9:** As shown in Table 7.7, the `infected_SHA` application was not included during the training phase of the proposed IDS. The `infected_SHA` application was created for this case study by adding a known user-space malicious code snippet into the SHA executable. However, only the original SHA application was used during the training phase. At runtime, when the `infected_SHA` application is encountered, the corresponding power profile does not match the SHA profile from the database. Hence, at runtime, the δ score triggers the alarm during execution that activates the monitoring of the other three techniques. Malware signature were detected that identifies the malware while no anomaly were found at the kernel-space. The calculated λ score classifies the application as anomaly. This behaviour corresponds to Scenario 7 of Table 7.6. This indicates that the executable contains ‘user-space malware’.
- Case Study 10:** As shown in Table 7.7, the AES application was used during the training phase of the proposed IDS. However, a `system_fault` that was not included during the training phase of the proposed IDS is ‘inserted’ into the OS immediately after AES application is started. At runtime, when the AES application is encountered, the corresponding power profile deviates from the AES power trace during execution. Hence, δ score triggers the alarm that enables the monitoring of the other three stages. As AES is a benign application, no malware signatures and kernel-space malware can be found. However, λ

score detects an anomaly. This behaviour corresponds to Scenario 8 of Table 7.6. This indicates of the possibility of a system fault if the λ score does not correspond to any of the known malware.

7.9 Summary

This chapter first described a lightweight technique to implement the power trace based anomaly detection system on existing embedded systems. Such a technique can be leveraged upon to detect anomalies in a non-intrusive fashion that can be especially useful for systems that cannot be instrumented with any anomaly detection code.

In order to classify user-space malware, the SYSCALL trace based malware detection and classification technique, as described in Chapter 4, was incorporated. The combined technique allows for a resource-efficient malware detection and classification system, by only invoking the more complex SYSCALL trace based technique when the power trace based technique detects an anomaly.

Next, the combination of power trace and architecture-specific features based techniques from Chapters 3 and 5 allowed for a lightweight methodology to verify the integrity of SYSCALLs in an operating system. Section 7.6 combined three features, power trace, SYSCALL trace and architecture-specific features to propose an efficient technique that can not only be deployed for lightweight malware detection and classification but also kernel verification.

The LAMBDA technique proposed in Chapter 6 was integrated with the earlier features to allow for a robust malware detection and classification by relying on SYSCALL trace simultaneously with the score obtained from LAMBDA. The confirmation on presence of a malware from both the SYSCALL trace and LAMBDA based techniques provides for a high degree of robustness in the malware detection system.

Lastly, a comprehensive real-time multi-modal intrusion detection framework was proposed that can be realised on existing embedded systems without being intrusive to the operating system or the applications running on it. The overall framework provides a comprehensive means to detect and classify attacks at various layers of embedded systems such as the application, operating system and the underlying architecture. The reliance on multiple features for intrusion detection allows us to leverage on the strengths of all the features while overcoming the limitations of an individual feature as discussed with several case studies in Section 7.8.5. In addition, the proposed framework can also differentiate these attacks from system faults. Finally, it was also shown that unlike existing works that require each feature to be extracted and monitored separately, SYSCALLs feature in this thesis could be extracted directly from the HPCs, thereby obviating the need to obtain them separately. This leads to further reduction in the monitoring overheads in the system.

Conclusions And Future Research

8.1 Summary of Research

In this thesis, several novel techniques have been proposed that culminated into a real-time lightweight multi-modal intrusion detection in embedded systems. The proposed method is non-intrusive and can be systematically incorporated into existing platforms to provide for multi-level monitoring at runtime to facilitate early detection.

The first line of defence for anomaly detection relies on the real-time monitoring of ‘power trace’. Unlike existing methods, the proposed approach has been shown to be lightweight and capable of differentiating anomalous applications from known (or benign) applications. It has been shown that ‘power trace’ can be used to accurately detect anomalies within $1ms$, making it very suitable for early detection, unlike existing works [39], [99] that relies on coarse-grained energy metrics. The sensitivity of the proposed method makes it possible to trigger prompt investigations for potentially malicious behaviour due to malware or Rootkit.

In order to classify user-space malware, a real-time ‘system call’ signature-based technique was proposed. Unlike existing methods, the proposed approach relies on a minimal set of ‘system call’ signatures for detecting malware. This was made possible

by identifying a smaller set of malware signatures that effectively represent the most prominent behaviours of a large family of malware. The proposed methodology was used to successfully extract 10 signatures to detect 420 malware with 93.4% detection accuracy. In addition, the proposed method only needs a total of 23 signatures to classify the 420 malware samples into their respective families with an average classification accuracy of 83%. The proposed method has been evaluated and shown to perform as well as existing methods [164], [165] [166] while lending well for real-time classification. It can be deployed with minimal intervention by relying solely on hardware performance counters (HPC) to monitor the 'system call'. Recommendations to further improve the classification accuracy is discussed as a work package in the future work section.

It was noted that the system call based approach discussed above is incapable of detecting Rootkit in the kernel space that affects and taint the system call itself. An SVM-based classification model was devised to differentiate tainted system calls from benign ones. HPC was monitored to analyse the characteristics of each system call to isolate malicious behaviour due to a Rootkit. It has been shown that the proposed approach is capable of detecting anomalous system calls with high accuracy while incurring lower monitoring overheads as compared the work in [86]. This method can also be integrated with that devised to detect malware in the user space to provide for a comprehensive approach to observe malicious behaviour both at the user and kernel spaces.

An alternative approach for detecting user-space malware by monitoring at the architecture level has been proposed to enhance the robustness of the malware detection process. The proposed approach is based on statistical hypothesis testing to distinguish between benign programs and malware. A new metric called λ is proposed to detect if an unknown program is benign or malicious. It is evident that combining both the architectural performance counters and OS indicators makes λ

an effective metric to differentiate between a benign and malicious program that has been shown to achieve a detection accuracy of 90% successfully. Additionally, the technique obtains significantly better false positive rate, at 1.52%, and a false negative rate of close to 0% when compared to the current state-of-the-art approaches in [139], [141], [194], and [196]. LAMBDA is also faster in the detection stages, taking only $32.43\mu s$ when compared to the widely used machine learning based techniques proposed in existing works. The lightweight approach of LAMBDA makes it well-suited for resource-constrained embedded systems as compared to the complex machine learning based approaches. Not only does the method accurately detect malware at runtime, but is also able to identify them early before the malware has a chance to complete. This method has paved the way for detecting malware and malware-like behaviour as well as system faults.

Finally, the integrated framework consisting of the above-mentioned methods has led to a robust detection of malware with the help of complementary techniques that monitor both the 'system call' and malware-specific indicators at the architecture level using LAMBDA. The framework also makes it possible to detect anomalies that exhibit malware-like behaviour to isolate potential zero-day attacks. In addition, the LAMBDA based method, in conjunction with the 'system call' trace, can also be relied upon to affirm system faults. Most importantly, the proposed methods can be deployed on existing platforms for real-time monitoring by merely observing HPC outputs without intruding into the system under observation to not impede on system performance.

8.2 Recommendations for Future Research

This thesis presented various techniques and strategies to detect and classify anomaly by using multiple metrics in a systematic approach. The characteristics of these selected metrics were carefully analyzed and optimized to strike a balance

between detection sensitivity and monitoring overhead. Leveraging the extensive customizability of the individual technique, future research can be undertaken to strengthen the ideas proposed in this thesis. The following subsections identify avenues for future research by extending the work proposed in this thesis:

- (i) **Incorporating System Call Frequency for Improving Malware Classification Accuracy:** The malware detection and classification technique proposed in Chapter 4 analyses the patterns of sequential system calls to achieve a high detection accuracy of over 90%. However, it can only categorize malware into respective families such as Trojan, Worm, etc. with an accuracy of just over 80%. Noting that each malware family exhibits different levels of correlation with the frequency of specific system calls, it would be of interest to incorporate the frequency of system calls with the techniques proposed in Chapter 4 to extract additional information for improved malware classification.
- (ii) **Rootkit Classification for Early Detection and Management:** In Chapter 5, a method is proposed to detect compromised system calls using a lump-sum hardware performance counter (HPC) value at the end of each system call execution. However, this method does not consider the HPC trace during system call execution that provides additional information regarding the Rootkit to allow for its classification. Similar to the technique proposed in Chapter 3 that uses power trace to capture the behaviour of benign applications, HPC trace can be relied upon to represent Rootkit behaviour. This can provide for additional information to categorize Rootkit to better differentiate them for early detection and management
- (iii) **LAMBDA for Classifying Malware-like Anomalies and System Faults:** In Chapter 6, the metric λ is effectively used to differentiate if the running program is benign or abnormal. Noting that the method proposed relies on the

monitoring of indicators that are being influenced by malware behaviour, it will be of interest to leverage machine learning based techniques to examine how the detected abnormality correlates with malware-like behaviour. This will be of help to categorize abnormal behaviours into potential-malware and system faults. This approach will also require access to additional data representing comprehensive coverage of system faults. Combined with the power trace, an integrated framework for classifying system faults can also be developed to allow appropriate remedial actions for specific system faults.

- (iv) **Evaluation of the Hierarchical Framework:** The proposed intrusion detection framework in Chapter 7 relies on a hierarchical approach involving using various metrics to detect malware, Rootkit, system faults and even potential zero-day malware attacks. These techniques can be integrated into an automated, unified framework and extensively evaluated for its effectiveness in robust malware detection using multiple complementary metrics such as λ and 'system call'. The proposed metrics can also be combined intelligently to thoroughly assess their usefulness in identifying and even classifying system faults as well as malware-like behaviour to isolate potential zero-day attacks.
- (v) **Adaptive Real-time Monitoring of HPC:** In Chapter 7, a proposed framework that combines the various anomaly detection and classification techniques was presented in this thesis. The Hardware Performance Counters were heavily relied upon to provide a non-intrusive means to obtain the various metrics needed for lightweight intrusion detection. To obtain the different set of hardware performance counters needed for the various metrics, such as HPC to extract system call ID, HPC counts for kernel verification and HPC for indicators used in LAMBDA technique, a round-robin time slicing approach was proposed. However, the round-robin scheduling approach may not be efficient if only one of the metrics detects an anomaly while the others stay silent. In future, it will

be interesting to propose an adaptive real-time scheduling policy to obtain the HPCs in succession for the metric or metrics that detect anomaly instead of waiting for other metrics that do not detect any anomaly. This will further improve the responsiveness of the proposed intrusion detection system so that remedial actions can be taken promptly.

- (vi) **Scalability of the Framework for Multi-OS Environments:** Chapter 7 proposed the use of a hypervisor to monitor the hardware, thereby allowing a guest OS and multiple applications to be executed without requiring any instrumentation for intrusion detection. The presence of hypervisor also allows for multiple guest operating systems to be seamlessly incorporated in future, especially on multi-core platforms. Hence, the proposed intrusion detection framework in this thesis needs to be evaluated for its scalability on a multi-core, multi-OS environment. One of the key strengths of the proposed framework is to only rely on monitoring different sets of HPCs to obtain the various metrics needed for anomaly detection. In future, the framework needs to be evaluated and extended to allow it to monitor not only multiple cores but also keep track of the multiple guest OSes by extracting the HPCs from all the cores under observation. The time-slicing concept proposed in Chapter 7 to obtain different sets of HPC for the various metrics can be extended for this purpose to allow the monitoring of different cores

References

- [1] S. Labs, “Simplicity Studio and Advanced Energy Monitoring.” www.silabs.com/products/mcu/32-bit/32-bit-microcontroller-technology, accessed 2019-8-01.
- [2] M. Spisak, “Hardware-assisted rootkits: Abusing performance counters on the arm and x86 architectures.” in *WOOT*, 2016.
- [3] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, “Chstone: A benchmark program suite for practical c-based high-level synthesis,” in *2008 IEEE International Symposium on Circuits and Systems*, pp. 1192–1195, May 2008.
- [4] R. Hasan *et al.*, “How secure is the healthcare network from insider attacks? an audit guideline for vulnerability analysis,” in *IEEE Annual Computer Software and Applications Conf. (COMPSAC)*, June 2016.
- [5] Y. Zhou and X. Jiang, “Dissecting android malware: Characterization and evolution,” in *2012 IEEE Symposium on Security and Privacy*, pp. 95–109, May 2012.
- [6] P. Duessel, C. Gehl, U. Flegel, S. Dietrich, and M. Meier, “Detecting zero-day attacks using context-aware anomaly detection at the application-layer,” *Int. J. Inf. Secur.*, vol. 16, pp. 475–490, Oct. 2017.
- [7] A. Moser *et al.*, “Exploring multiple execution paths for malware analysis,” in *2007 IEEE Symp. on Security and Privacy (SP '07)*, May 2007.
- [8] S. S. Response, “lot devices being increasingly used for ddos attacks,” 2016. [Online; accessed 28-March-2017].

- [9] B. Snell, "Mobile threat report: What's on the horizon for 2016," 2016. [Online; accessed 28-March-2017].
- [10] A. Biswas, "Scary insights into the security of smart things: What the iot startups don't pitch about," 2016. [Online; accessed 28-March-2017].
- [11] L. Greenemeier, "Iot growing faster than the ability to defend it," 2016. [Online; accessed 28-March-2017].
- [12] M. S. Iqbal *et al.*, "SAM: A secure anti-malware framework for the smartphone operating systems," in *IEEE Wireless Communications and Networking Conf.*, 2016.
- [13] S. Greengard, "Cybersecurity gets smart," *Commun. ACM*, 2016.
- [14] M. Fauzi, A. Prakash, and T. Srikanthan, "Power profile based runtime anomaly detection," in *2017 TRON Symposium (TRONSHOW)*, pp. 1–9, Dec 2017.
- [15] M. Fauzi and T. Srikanthan, *Low-Complexity Signature-Based Malware Detection for IoT Devices*, pp. 181–189. Singapore: Springer, 2017.
- [16] M. Fauzi, S. P. Kadiyala, A. Prakash, T. Srikanthan, and Y. L. Aung, "Hardware performance counters based runtime anomaly detection using svm," in *TRON Symposium (TRONSHOW), 2017*, pp. 1–9, IEEE, 2017.
- [17] M. Fauzi, T. Srikanthan, and S. K. Lam, "Flexible and scalable power management framework for rtos based embedded systems," in *Embedded and Networking Systems: Design, Software, and Implementation* (G. N. Khan and K. Iniewski, eds.), ch. 5, pp. 105–210, Florida: CRC Press, 2014.
- [18] J. A. Yong, "Info on 1.5m singhealth patients stolen in worst cyber attack," 2018. <https://www.straitstimes.com/singapore/info-on-15m-singhealth-patients-stolen-in-worst-cyber-attack>, accessed 2018-12-01.
- [19] "ThreatMetrix: Q1 2018 Cybercrime Report." <https://www.threatmetrix.com/info/q1-2018-cybercrime-report/>, accessed 2018-12-01.

- [20] C. S. S., "Comodo threat research labs," 2018. <https://www.comodo.com/GTR/Q3/2018/Comodo-Cybersecurity-Global-Threat-Report-Q3-2018.pdf>, accessed 2018-12-01.
- [21] R. Z. A. Mohd, M. F. Zuhairi, A. Z. A. Shadil, and H. Dao, "Anomaly-based nids: A review of machine learning methods on malware detection," in *2016 International Conference on Information and Communication Technology (ICICTM)*, pp. 266–270, May 2016.
- [22] P. I. Radoglou-Grammatikis and P. G. Sarigiannidis, "Flow anomaly based intrusion detection system for android mobile devices," in *2017 6th International Conference on Modern Circuits and Systems Technologies (MOCASST)*, pp. 1–4, May 2017.
- [23] Y. Imamverdiyev and L. Sukhostat, "Anomaly detection in network traffic using extreme learning machine," in *2016 IEEE 10th International Conference on Application of Information and Communication Technologies (AICT)*, pp. 1–4, Oct 2016.
- [24] P. Alaei and F. Noorbehbahani, "Incremental anomaly-based intrusion detection system using limited labeled data," in *2017 3th International Conference on Web Research (ICWR)*, pp. 178–184, April 2017.
- [25] G. Wagener, A. Dulaunoy, *et al.*, "Malware behaviour analysis," *Journal in computer virology*, vol. 4, no. 4, pp. 279–287, 2008.
- [26] R. Canzanese, S. Mancoridis, and M. Kam, "Run-time classification of malicious processes using system call analysis," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 21–28, Oct 2015.
- [27] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pp. 62–75, IEEE, 2003.
- [28] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14, SSYM'05*, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2005.

- [29] G. McGraw and G. Morrisett, "Attacking malicious code: A report to the infosec research council," *IEEE software*, vol. 17, no. 5, pp. 33–41, 2000.
- [30] V. V. Phoha, *Internet security dictionary*, vol. 1. Taylor & Francis, 2007.
- [31] M. Johns and M. Jodeit, "Scanstud: a methodology for systematic, fine-grained evaluation of static analysis tools," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pp. 523–530, IEEE, 2011.
- [32] I. Pashchenko, S. Dashevskyi, and F. Massacci, "Delta-bench: Differential benchmark for static analysis security testing tools," in *Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE International Symposium on*, pp. 163–168, IEEE, 2017.
- [33] H. Chen, D. Dean, and D. Wagner, "Model checking one million lines of c code.," in *NDSS*, vol. 4, pp. 171–185, 2004.
- [34] D. Wagner and R. Dean, "Intrusion detection via static analysis," in *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pp. 156–168, IEEE, 2001.
- [35] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-sequence-based malware detection," in *Engineering Secure Software and Systems* (F. Massacci, D. Wallach, and N. Zannone, eds.), (Berlin, Heidelberg), pp. 35–43, Springer Berlin Heidelberg, 2010.
- [36] A. Walenstein and A. Lakhotia, "The software similarity problem in malware analysis," in *Duplication, Redundancy, and Similarity in Software* (R. Koschke, E. Merlo, and A. Walenstein, eds.), no. 06301 in Dagstuhl Seminar Proceedings, (Dagstuhl, Germany), Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
- [37] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*, pp. 290–299, ACM, 2003.

- [38] F. Guo, P. Ferrie, and T.-C. Chiueh, "A study of the packer problem and its solutions," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 98–115, Springer, 2008.
- [39] E. Viegas, A. O. Santin, A. Franca, R. Jasinski, V. A. Pedroni, and L. S. Oliveira, "Towards an energy-efficient anomaly-based intrusion detection engine for embedded systems," *IEEE Transactions on Computers*, vol. 66, pp. 163–177, Jan 2017.
- [40] R. Moskovitch, I. Gus, S. Pluderman, D. Stopel, C. Glezer, Y. Shahar, and Y. Elovici, "Detection of unknown computer worms activity based on computer behavior using data mining," in *2007 IEEE Symposium on Computational Intelligence in Security and Defense Applications*, pp. 169–177, April 2007.
- [41] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs," in *Proceedings of the sixth ACM workshop on Scalable trusted computing*, pp. 71–76, ACM, 2011.
- [42] M. B. Bahador, M. Abadi, and A. Tajoddin, "Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition," in *2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*, pp. 703–708, Oct 2014.
- [43] E. Totel, M. Hkimi, M. Hurfin, M. Leslous, and Y. Labiche, "Inferring a distributed application behavior model for anomaly based intrusion detection," in *2016 12th European Dependable Computing Conference (EDCC)*, pp. 53–64, Sept 2016.
- [44] M. Sheikhan and H. Bostani, "A hybrid intrusion detection architecture for internet of things," in *2016 8th International Symposium on Telecommunications (IST)*, pp. 601–606, Sept 2016.
- [45] M. Mdini, A. Blanc, G. Simon, J. Barotin, and J. Lecoeuvre, "Monitoring the network monitoring system: Anomaly detection using pattern recognition," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 983–986, May 2017.

- [46] C. Li, D. Srinivasan, and T. Reindl, "Hardware-assisted malware detection for embedded systems in smart grid," in *2015 IEEE Innovative Smart Grid Technologies - Asia (ISGT ASIA)*, pp. 1–6, Nov 2015.
- [47] M. K. Yoon, S. Mohan, J. Choi, M. Christodorescu, and L. Sha, "Learning execution contexts from system call distribution for anomaly detection in smart embedded system," in *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pp. 191–196, April 2017.
- [48] S. A. Maske and T. J. Parvat, "Advanced anomaly intrusion detection technique for host based system using system call patterns," in *2016 International Conference on Inventive Computation Technologies (ICICT)*, vol. 2, pp. 1–4, Aug 2016.
- [49] S. Brady, A. Hava, P. Perry, J. Murphy, D. Magoni, and A. O. Portillo-Dominguez, "Towards an emulated iot test environment for anomaly detection using nemu," in *2017 Global Internet of Things Summit (GloTS)*, pp. 1–6, June 2017.
- [50] C. Vaas and J. Happa, "Detecting disguised processes using application-behavior profiling," in *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*, pp. 1–6, April 2017.
- [51] A. Cuzzocrea, E. Mumolo, and R. Cecolin, "Runtime anomaly detection in embedded systems by binary tracing and hidden markov models," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, pp. 15–22, July 2015.
- [52] H. Debar, M. Dacier, and A. Wespi, "Towards a taxonomy of intrusion-detection systems," *Computer Networks*, vol. 31, no. 8, pp. 805–822, 1999.
- [53] D. Dasgupta and F. Nino, "A comparison of negative and positive selection algorithms in novel pattern detection," in *Systems, man, and cybernetics, 2000 IEEE international conference on*, vol. 1, pp. 125–130, IEEE, 2000.
- [54] S. Forrest *et al.*, "A sense of self for unix processes," in *IEEE Symp. on Security and Privacy*, May 1996.

- [55] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *J. Comput. Secur.*, vol. 19, pp. 639–668, Dec 2011.
- [56] M. Atallah, W. Szpankowski, and R. Gwadera, "Detection of significant sets of episodes in event sequences," in *Data Mining, 2004. ICDM'04. Fourth IEEE International Conference on*, pp. 3–10, IEEE, 2004.
- [57] R. Gwadera, M. Atallah, and W. Szpankowski, "Markov models for identification of significant episodes," in *Proceedings of the 2005 SIAM International Conference on Data Mining*, pp. 404–414, SIAM, 2005.
- [58] I. S. Thaseen and C. A. Kumar, "Intrusion detection model using fusion of chi-square feature selection and multi class svm," *Journal of King Saud University-Computer and Information Sciences*, vol. 29, no. 4, pp. 462–472, 2017.
- [59] Z. Wang, J. Yang, Z. ShiZe, and C. Li, "Robust regression for anomaly detection," in *2017 IEEE International Conference on Communications (ICC)*, pp. 1–6, May 2017.
- [60] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: theory and applications," *Neurocomputing*, vol. 70, no. 1-3, pp. 489–501, 2006.
- [61] V. Jyothsna and V. R. Prasad, "Fcaais: Anomaly based network intrusion detection through feature correlation analysis and association impact scale," *ICT Express*, vol. 2, no. 3, pp. 103–116, 2016.
- [62] S. M. Hussein, "Performance evaluation of intrusion detection system using anomaly and signature based algorithms to reduction false alarm rate and detect unknown attacks," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 1064–1069, Dec 2016.
- [63] CISCO, "Snort," 2019. <https://www.snort.org/>, accessed 2019-9-01.
- [64] A. McCallum, K. Nigam, *et al.*, "A comparison of event models for naive bayes text classification," in *AAAI-98 workshop on learning for text categorization*, vol. 752, pp. 41–48, Citeseer, 1998.

- [65] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: alternative data models," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No.99CB36344)*, pp. 133–145, 1999.
- [66] X. A. Hoang and J. Hu, "An efficient hidden markov model training scheme for anomaly intrusion detection of server applications based on system calls," in *Proceedings. 2004 12th IEEE International Conference on Networks (ICON 2004) (IEEE Cat. No.04EX955)*, vol. 2, pp. 470–474 vol.2, Nov 2004.
- [67] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the detection of anomalous system call arguments," in *Computer Security – ESORICS 2003* (E. Sneekenes and D. Gollmann, eds.), (Berlin, Heidelberg), pp. 326–343, Springer Berlin Heidelberg, 2003.
- [68] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, pp. 381–395, Oct 2010.
- [69] J. T. Giffin, S. Jha, and B. P. Miller, "On effective model-based intrusion detection," tech. rep., University of Wisconsin, Madison, 2005.
- [70] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, AISec '13*, (New York, NY, USA), pp. 45–54, ACM, 2013.
- [71] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, pp. 381–395, Oct 2010.
- [72] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012*, (New York, NY, USA), pp. 122–132, ACM, 2012.

- [73] M. Chandramohan, H. B. K. Tan, L. C. Briand, L. K. Shar, and B. M. Padmanabuni, "A scalable approach for malware detection through bounded feature space behavior modeling," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 312–322, Nov 2013.
- [74] D. Bilar, "Opcodes as predictor for malware," *International Journal of Electronic Security and Digital Forensics*, vol. 1, no. 2, pp. 156–168, 2007.
- [75] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, *Idea: Opcode-Sequence-Based Malware Detection*, pp. 35–43. Springer Berlin Heidelberg, 2010.
- [76] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Information Sciences*, vol. 231, no. Supplement C, pp. 64 – 82, 2013. Data Mining for Information Security.
- [77] N. Runwal, R. M. Low, and M. Stamp, "Opcode graph similarity and metamorphic detection," *Journal in Computer Virology*, vol. 8, pp. 37–52, May 2012.
- [78] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: Towards efficient real-time protection against malware," *IEEE Transactions on Information Forensics and Security*, vol. 11, pp. 289–302, Feb 2016.
- [79] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. A. Ghazaleh, "Signature-based protection from code reuse attacks," *IEEE Transactions on Computers*, vol. 64, pp. 533–546, Feb 2015.
- [80] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS '11*, (New York, NY, USA), pp. 30–40, ACM, 2011.
- [81] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07*, (New York, NY, USA), pp. 552–561, ACM, 2007.

- [82] A. K. Kanuparthi, R. Karri, G. Ormazabal, and S. K. Addepalli, "A high-performance, low-overhead microarchitecture for secure program execution," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pp. 102–107, Sept 2012.
- [83] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection.," in *USENIX Security Symposium*, vol. 2014, 2014.
- [84] D. Gao, M. K. Reiter, and D. Song, "Beyond output voting: Detecting compromised replicas using hmm-based behavioral distance," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, pp. 96–110, April 2009.
- [85] X. Zhai, K. Appiah, S. Ehsan, G. Howells, H. Hu, D. Gu, and K. D. McDonald-Maier, "A method for detecting abnormal program behavior on embedded devices," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 8, pp. 1692–1704, 2015.
- [86] X. Wang and R. Karri, "Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters," in *Proceedings of the 50th Annual Design Automation Conference, DAC '13*, (New York, NY, USA), pp. 79:1–79:7, ACM, 2013.
- [87] M. Ozsoy, C. Donovanick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 651–661, Feb 2015.
- [88] X. Zhai, K. Appiah, S. Ehsan, G. Howells, H. Hu, D. Gu, and K. D. McDonald-Maier, "A method for detecting abnormal program behavior on embedded devices," *IEEE Transactions on Information Forensics and Security*, vol. 10, pp. 1692–1704, Aug 2015.
- [89] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13*, (New York, NY, USA), pp. 559–570, ACM, 2013.

- [90] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 109–129, Springer, 2014.
- [91] D. Krishnankutty, R. Robucci, N. Banerjee, and C. Patel, "Fiscal: Firmware identification using side-channel power analysis," in *2017 IEEE 35th VLSI Test Symposium (VTS)*, pp. 1–6, April 2017.
- [92] C. Luo, Y. Fei, and A. A. Ding, "Side-channel power analysis of xts-aes," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 1330–1335, March 2017.
- [93] E. J. Vergara, *Energy Modelling and Fairness for Efficient Mobile Communication*. PhD thesis, Department of Computer and Information Science, Linkoping University, 2016.
- [94] M. Kar, A. Singh, S. Mathew, A. Rajan, V. De, and S. Mukhopadhyay, "8.1 improved power-side-channel-attack resistance of an aes-128 core via a security-aware integrated buck voltage regulator," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 142–143, Feb 2017.
- [95] L. Xiangyu and S. Yihe, "A low power side channel attack resistant data flow rindael encryption asic," in *2006 IEEE Asian Solid-State Circuits Conference*, pp. 227–230, Nov 2006.
- [96] D. Jayasinghe, A. Ignjatovic, J. A. Ambrose, R. Ragel, and S. Parameswaran, "Quadseal: Quadruple algorithmic symmetrizing countermeasure against power based side-channel attacks," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 21–30, Oct 2015.
- [97] P. Luo, L. Zhang, Y. Fei, and A. A. Ding, "Towards secure cryptographic software implementation against side-channel power analysis attacks," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 144–148, July 2015.

- [98] S. M. D. Pozo, F. X. Standaert, D. Kamel, and A. Moradi, "Side-channel attacks from static power: When should we care?," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 145–150, March 2015.
- [99] H. Kurniawan, Y. Rosmansyah, and B. Dabarsyah, "Android anomaly detection system using machine learning classification," in *2015 International Conference on Electrical Engineering and Informatics (ICEEI)*, pp. 288–293, Aug 2015.
- [100] P. Arjunan, H. D. Khadilkar, T. Ganu, Z. M. Charbiwala, A. Singh, and P. Singh, "Multi-user energy consumption monitoring and anomaly detection with partial context information," in *Proceedings of the 2Nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments, BuildSys '15*, (New York, NY, USA), pp. 35–44, ACM, 2015.
- [101] J.-S. Chou and A. S. Telaga, "Real-time detection of anomalous power consumption," *Renewable and Sustainable Energy Reviews*, vol. 33, pp. 400 – 411, 2014.
- [102] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," in *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pp. 239–252, ACM, 2008.
- [103] Hardkernel, "Odroid-xu3," 2017. https://wiki.odroid.com/old_product/odroid-xu3/odroid-xu3, accessed 2019-07-22.
- [104] H. Sedjelmaci, S. M. Senouci, and M. Al-Bahri, "A lightweight anomaly detection technique for low-resource iot devices: A game-theoretic methodology," in *2016 IEEE International Conference on Communications (ICC)*, pp. 1–6, May 2016.
- [105] H.-S. Ham and M.-J. Choi, "Analysis of android malware detection performance using machine learning classifiers," in *ICT Convergence (ICTC), 2013 International Conference on*, pp. 490–495, IEEE, 2013.
- [106] T. Badgujar and P. More, "An intrusion detection system implementing host based attacks using layered framework," in *Innovations in Information, Embedded and Communication Systems (ICIIECS), 2015 International Conference on*, pp. 1–4, IEEE, 2015.

- [107] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *computers & security*, vol. 28, no. 1-2, pp. 18–28, 2009.
- [108] U. Bashir and M. Chachoo, "Intrusion detection and prevention system: Challenges & opportunities," in *Computing for Sustainable Global Development (INDIACom), 2014 International Conference on*, pp. 806–809, IEEE, 2014.
- [109] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 421–430, Dec 2007.
- [110] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS '08*, (New York, NY, USA), pp. 51–62, ACM, 2008.
- [111] G. Creech and J. Hu, "A semantic approach to host-based intrusion detection systems using contiguous and discontinuous system call patterns," *IEEE Transactions on Computers*, vol. 63, pp. 807–819, April 2014.
- [112] Q. Song, W. Hu, and W. Xie, "Robust support vector machine with bullet hole image classification," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 32, no. 4, pp. 440–448, 2002.
- [113] W. Hu, Y. Liao, and V. R. Vemuri, "Robust support vector machines for anomaly detection in computer security," in *ICMLA*, pp. 168–174, 2003.
- [114] R. A. Maxion and K. M. Tan, "Benchmarking anomaly-based detection systems," in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pp. 623–630, IEEE, 2000.
- [115] K. Bache and M. Lichman, "Uci machine learning repository," 2013. [Online; accessed 30-May-2018].
- [116] N. E. I. Karabadji, H. Seridi, F. Bousetouane, W. Dhifli, and S. Aridhi, "An evolutionary scheme for decision tree construction," *Knowledge-Based Systems*, vol. 119, pp. 166–177, 2017.

- [117] H. Sarvari, C. Domeniconi, and G. Stilo, "Graph-based selective outlier ensembles," *arXiv preprint arXiv:1804.06378*, 2018.
- [118] A. F. Emmott, S. Das, T. Dietterich, A. Fern, and W.-K. Wong, "Systematic construction of anomaly detection benchmarks from real data," in *Proceedings of the ACM SIGKDD workshop on outlier detection and description*, pp. 16–21, ACM, 2013.
- [119] V. Total, "VirusTotal-free online virus, malware and url scanner," *Online*: <https://www.virustotal.com/en>, 2012.
- [120] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket.," in *Ndss*, vol. 14, pp. 23–26, 2014.
- [121] J.-M. Roberts, "VirusShare.com - Because Sharing is Caring," 2017. <https://virusshare.com>, accessed 2018-12-01.
- [122] X. Wang, S. Chai, M. Isnardi, S. Lim, and R. Karri, "Hardware performance counter-based malware identification and detection with adaptive compressive sensing," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, p. 3, 2016.
- [123] T. E. M. B. C. (EEMBC), "Benchmark software," 2018. [Online; accessed 29-May-2018].
- [124] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pp. 3–14, Dec 2001.
- [125] K. Hamandi, A. Chehab, I. H. Elhajj, and A. Kayssi, "Android sms malware: Vulnerability and mitigation," in *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*, pp. 1004–1009, IEEE, 2013.

- [126] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Network-level polymorphic shellcode detection using emulation," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 54–73, Springer, 2006.
- [127] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo, "On the infeasibility of modeling polymorphic shellcode," in *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 541–551, ACM, 2007.
- [128] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime intrusion prevention evaluator," in *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*, ACM, 2011.
- [129] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *Workshop on the evaluation of software defect detection tools*, vol. 5, 2005.
- [130] P. E. Black, "Software assurance metrics and tool evaluation.," in *Software Engineering Research and Practice*, pp. 829–835, 2005.
- [131] N. N. Tran, R. Sarker, and J. Hu, "An approach for host-based intrusion detection system design using convolutional neural network," in *Mobile Networks and Management* (J. Hu, I. Khalil, Z. Tari, and S. Wen, eds.), (Cham), pp. 116–126, Springer International Publishing, 2018.
- [132] T. Hamed, J. B. Ernst, and S. C. Kremer, *A Survey and Taxonomy of Classifiers of Intrusion Detection Systems*, pp. 21–39. Cham: Springer International Publishing, 2018.
- [133] K. Singh, M. Bhadauria, and S. A. McKee, "Real time power estimation and thread scheduling via performance counters," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 46–55, 2009.
- [134] S. Moore, M. Yampolskiy, J. Gatlin, J. T. McDonald, and T. R. Andel, "Buffer overflow attack's power consumption signatures," in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, p. 6, ACM, 2016.

- [135] D.-K. Kang, D. Fuller, and V. Honavar, "Learning classifiers for misuse and anomaly detection using a bag of system calls representation," in *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, pp. 118–125, June 2005.
- [136] B. Singh, D. Evtushkin, J. Elwell, R. Riley, and I. Cervesato, "On the detection of kernel-level rootkits using hardware performance counters," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 483–493, ACM, 2017.
- [137] N. Patel, A. Sasan, and H. Homayoun, "Analyzing hardware based malware detectors," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 25, ACM, 2017.
- [138] Y. Molchan, "New mining malware for xmr evolves: Research," 2018. <https://u.today/new-mining-malware-for-xmr-evolves-research>, accessed 2019-06-1.
- [139] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: towards efficient real-time protection against malware," *IEEE transactions on information forensics and security*, vol. 11, no. 2, pp. 289–302, 2016.
- [140] X. Wang and R. Karri, "Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 485–498, 2016.
- [141] X. Wang, C. Konstantinou, M. Maniatakis, and R. Karri, "Confirm: Detecting firmware modifications in embedded systems using hardware performance counters," in *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*, pp. 544–551, IEEE, 2015.
- [142] S. Nari *et al.*, "Automated malware classification based on network behavior," in *Int. Conf. on Computing, Networking and Communications (ICNC)*, Jan 2013.

- [143] A. M. Kosek and O. Gehrke, "Ensemble regression model-based anomaly detection for cyber-physical intrusion detection in smart grids," in *2016 IEEE Electrical Power and Energy Conference (EPEC)*, pp. 1–7, Oct 2016.
- [144] A. M. Kosek and O. Gehrke, "Ensemble regression model-based anomaly detection for cyber-physical intrusion detection in smart grids," in *Electrical Power and Energy Conference (EPEC), 2016 IEEE*, pp. 1–7, IEEE, 2016.
- [145] W. Chen, F. Kong, F. Mei, G. Yuan, and B. Li, "A novel unsupervised anomaly detection approach for intrusion detection system," in *Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS), 2017 IEEE 3rd International Conference on*, pp. 69–73, IEEE, 2017.
- [146] J. Wang, "Zero-day vulnerabilities: How do you stop a threat you can't see coming?," 2017. [Online; accessed 27-August-2017].
- [147] Y. Zhang, W. Chen, and J. Black, "Anomaly detection in premise energy consumption data," in *2011 IEEE Power and Energy Society General Meeting*, pp. 1–8, July 2011.
- [148] R. D. Keane and R. J. Adrian, "Theory of cross-correlation analysis of piv images," *Applied Scientific Research*, vol. 49, pp. 191–215, Jul 1992.
- [149] S. Labs, "Efm32 giant gecko starter kit," 2017. www.silabs.com/products/mcu/32-bit/efm32-giant-gecko, accessed 2018-12-01.
- [150] E. Bressert, *SciPy and NumPy: an overview for developers*. " O'Reilly Media, Inc.", 2012.
- [151] S. Labs, "uc/os-ii on efm32 giant gecko starter kit," 2017. www.silabs.com/products/development-tools/software/micrium-os/micrium-real-time-kernel, accessed 2018-12-01.
- [152] É. Filiol and F. Raynal, "Malicious cryptography... reloaded," in *CanSecWest Conference, Vancouver, Canada.[online] http://cansecwest.com/csw08/csw08-raynal.pdf*, 2008.

- [153] N. W. Paton, M. H. Williams, K. Dietrich, O. Liew, A. Dinn, and A. Patrick, *VESPA: A Benchmark for Vector Spatial Databases*, pp. 81–101. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000.
- [154] Kokke, “Tiny AES GitHub.” <https://github.com/kokke/tiny-AES-c>, accessed 2018-12-01.
- [155] B. Conte, “Crypto-Algorithm GitHub.” <https://github.com/B-Con/crypto-algorithms>, accessed 2018-12-01.
- [156] D. Suhubdy, “C implementation of the Blowfish algorithm GitHub.” <https://github.com/dendisuhubdy/encryption/blob/master/BFSH-KOC/Blowfish.c>, accessed 2018-12-01.
- [157] M. Barr, “Crc series,” 1999. <https://barrgroup.com/Embedded-Systems/How-To/Additive-Checksums>, accessed 2018-12-01.
- [158] B. Arslan *et al.*, “A review on mobile threats and machine learning based detection approaches,” in *Int. Symp. on Digital Forensic and Security*, April 2016.
- [159] B. Kolosnjaji *et al.*, “Deep Learning for Classification of Malware System Call Sequences,” in *Australasian Joint Conf. on Artificial Intelligence*, 2016.
- [160] Z. A. Othman *et al.*, “Improving signature detection classification model using features selection based on customized features,” in *Int. Conf. on Intelligent Systems Design and Applications*, Nov 2010.
- [161] A. Saracino *et al.*, “Madam: Effective and efficient behavior-based android malware detection and prevention,” *IEEE Trans. on Dependable and Secure Computing*, 2016.
- [162] B. N. Narayanan *et al.*, “Performance analysis of machine learning and pattern recognition algorithms for malware classification,” in *IEEE National Aerospace and Electronics Conf. (NAECON) and Ohio Innovation Summit (OIS)*, July 2016.
- [163] N. Islam, S. Das, and Y. Chen, “On-device mobile phone security exploits machine learning,” *IEEE Pervasive Computing*, no. 2, pp. 92–96, 2017.

- [164] A. Hellal *et al.*, “Maximal frequent sub-graph mining for malware detection,” in *Int. Conf. on Intelligent Systems Design and Applications (ISDA)*, Dec 2015.
- [165] M. Sun *et al.*, “Monet: A user-oriented behavior-based malware variants detection system for android,” *IEEE Trans. on Information Forensics and Security*, 2017.
- [166] E. Gandotra, D. Bansal, and S. Sofat, “Malware analysis and classification: A survey,” *Journal of Information Security*, vol. 5, no. 02, p. 56, 2014.
- [167] B. Kolosnjaji, G. Eraisha, G. Webster, A. Zarras, and C. Eckert, “Empowering convolutional networks for malware classification and analysis,” in *Neural Networks (IJCNN), 2017 International Joint Conference on*, pp. 3838–3845, IEEE, 2017.
- [168] M. Rahmatian, H. Kooti, I. G. Harris, and E. Bozorgzadeh, “Hardware-assisted detection of malicious software in embedded systems,” *IEEE Embedded Systems Letters*, vol. 4, no. 4, pp. 94–97, 2012.
- [169] J. Zhang, K. Zhang, Z. Qin, H. Yin, and Q. Wu, “Sensitive system calls based packed malware variants detection using principal component initialized multi-layers neural networks,” *Cybersecurity*, vol. 1, no. 1, p. 10, 2018.
- [170] ARM, “Axi to ahb-lite bridge cycle model,” 2019. http://infocenter.arm.com/help/topic/com.arm.doc.dui1055a/cycle_models_AMBA_AXI_AHBLite_Bridge_User_DUI1055A_en.pdf, accessed 2019-8-30.
- [171] C. Gaisler, “Leon3 processors,” 2017. <https://www.gaisler.com/index.php/products/processors/leon3>, accessed 2019-8-30.
- [172] Terasic, “Socikit - the development kit for new soc device,” 2019. <https://www.terasic.com.tw>, accessed 2019-8-30.
- [173] T. Röehl, J. Treibig, G. Hager, and G. Wellein, “Overhead analysis of performance counter measurements,” in *Parallel Processing Workshops (ICCPW), 2014 43rd International Conference on*, pp. 176–185, IEEE, 2014.

- [174] A. Nowak and G. Bitzes, "The overhead of profiling using pmu hardware counters, july 2014," URL [http://dx. doi. org/10.5281/zenodo](http://dx.doi.org/10.5281/zenodo), vol. 10800.
- [175] L. Boero, M. Marchese, and S. Zappatore, "Support vector machine meets software defined networking in ids domain," in *2017 29th International Teletraffic Congress (ITC 29)*, vol. 3, pp. 25–30, Sept 2017.
- [176] M. Anandapriya and B. Lakshmanan, "Anomaly based host intrusion detection system using semantic based system call patterns," in *2015 IEEE 9th International Conference on Intelligent Systems and Control (ISCO)*, pp. 1–4, Jan 2015.
- [177] B. Hammer, M. Strickert, and T. Villmann, "Relevance lvq versus svm," in *International Conference on Artificial Intelligence and Soft Computing*, pp. 592–597, Springer, 2004.
- [178] J. Shawe-Taylor, P. L. Bartlett, R. C. Williamson, and M. Anthony, "Structural risk minimization over data-dependent hierarchies," *IEEE transactions on Information Theory*, vol. 44, no. 5, pp. 1926–1940, 1998.
- [179] C. Gaisler, "Linux for leon processors," 2017. <http://www.gaisler.com/index.php/products/operating-systems/linux>, accessed 2018-12-01.
- [180] P. Membrey, T. Verhoeven, R. Angenendt, B. de Bruijn, and K. Singh, *The definitive guide to Centos*. Springer, 2009.
- [181] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang, "Bringing virtualization to the x86 architecture with the original vmware workstation," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, p. 12, 2012.
- [182] W. Kleitz, *Digital Electronics with VHDL, Quartus II Version*. Pearson Prentice Hall, 2006.
- [183] S. kit learn, "Support Vector Machine (SVM)." <https://scikit-learn.org/stable/modules/svm.html>, accessed 2019-07-22.
- [184] ODROID-Wiki, "Step-by-step guide to building an odroid-xu3/4 kernel," 2018. https://wiki.odroid.com/odroid-xu4/software/building_kernel, accessed 2019-07-22.

- [185] D. Lobo, P. Watters, and X. Wu, "Rbacs: Rootkit behavioral analysis and classification system," in *Knowledge Discovery and Data Mining, 2010. WKDD'10. Third International Conference on*, pp. 75–80, IEEE, 2010.
- [186] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 122–132, ACM, 2012.
- [187] X. Wang and R. Karri, "Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 485–498, 2015.
- [188] J. C. Saez, A. Pousa, R. Rodriguez-Rodriguez, F. Castro, and M. Prieto-Matias, "Pmctrack: Delivering performance monitoring counter support to the os scheduler," *The Computer Journal*, vol. 60, no. 1, pp. 60–85, 2017.
- [189] "Offensive Security." <https://www.offensive-security.com/>, accessed 2018-12-01.
- [190] K. Lucas, "BYTE UNIX benchmark suite." <https://github.com/kdlucas/byte-unixbench>, accessed 2018-12-01.
- [191] L. McVoy and C. Staelin, "LMBench - Tools for Performance Analysis." <http://www.bitmover.com/lmbench/>, accessed 2018-12-01.
- [192] A. Cortex, "A9 mpcore technical reference manual," *June rev r4p1*, 2012.
- [193] S. R. GitHub, "An lkm rootkit targeting linux 2.6/3.x on x86(_64), and arm." <https://github.com/mncoppola/suterusu>, accessed 2018-12-01.
- [194] M. Ozsoy, K. N. Khasawneh, C. Donovan, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Hardware-based malware detection using low-level architectural features," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3332–3344, 2016.

- [195] S. Das, B. Chen, M. Chandramohan, Y. Liu, and W. Zhang, "Ropsentry: runtime defense against rop attacks using hardware performance counters," *Computers & Security*, 2017.
- [196] R. Elnaggar, K. Chakrabarty, and M. B. Tahoori, "Run-time hardware trojan detection using performance counters," in *Test Conference (ITC), 2017 IEEE International*, pp. 1–10, IEEE, 2017.
- [197] D. Lobo, P. Watters, and X.-W. Wu, "Identifying rootkit infections using data mining," in *2010 International Conference on Information Science and Applications (ICISA 2010)*, pp. 1–7, IEEE, 2010.
- [198] C. Walls, "Embedded hypervisors," 2016. <https://www.embedded.com/design/operating-systems/4441695/2/Embedded-hypervisors>, accessed 2019-8-01.

Appendix A

The following are bash shell scripts used in Chapter 4. These scripts were executed on an Ubuntu Linux environment.

Time Slicing

```
1 #!/bin/bash
2
3 if [ $# -lt 1 ]
4 then
5     echo "$0 [folder-name]"
6     exit 0
7 fi
8 a=0;
9 localfolder=~ /SYSCALL/$1
10 cd $localfolder
11 for file in *.log
12 do
13     awk '{print substr($0,0,11)}' "$file" | uniq > timeslice.temp
14     for line in $(cat timeslice.temp)
15     do
16         let "a += 1"
17
18         if [ $a -lt 17 ]
19         then grep "$line" "$file" > slicer/"$a"/"$file"_"$a"
20         elif [ $a -gt 16 ]
21         then grep "$line" "$file" > slicer/etc/"$file"_"$a"
22         fi
23     done
24     rm timeslice.temp
25     a=0;
26 done
```

Script to segment SYSCALL trace by time

Signature Extractor

```
1 #!/bin/bash
2
3 if [ $# -lt 2 ]
4 then
5     echo "$0 [folder-name] [percentage]"
6     echo "    folder-name = where all the .log file resides"
7     echo "    percentage = 0.01 --> features found in 1% of apps;
8     0.99 --> features found in 99% of the apps"
9     exit 0
10 fi
11
12 localfolder=~ /SYSCALL/$1
13 rm common_$1.feats
14 cd $localfolder
15 rm $localfolder/freq/*.freq
16 rm $localfolder/feat/*.feats
17
18 for file in *.log_1
19 do
20     awk '{print $2}' "$file" > infile2
21     awk -f ../clean.awk infile2 > infile
22     ../extract.awk > mw.temp
23     awk -F '\t' '{print $1}' mw.temp | sort | uniq -c | sort -nr >
24     freq/"$file".freq
25     sort -u mw.temp > feat/"$file".feats
26     rm infile infile2 mw.temp
27 done
28
29 awk -v percentage=$2 'FNR==1{nfiles++; delete fseen} !($0 in fseen)
30 {fseen[$0]++; seen[$0]++; END{for (key in seen) if (seen[key]>(
31 nfiles*percentage)) print key}}' $localfolder/feat/*.feats > ../
32 common_$1.feats
```

Script to extract signature from SYSCALL trace

```
1 #!/bin/awk -f
2
3 if ( $0 ~ /\(\/ ) {
4     FS="(";
5     $0=$0
6 }
7
8 if($1 !~ /(---|killed|exited)/ ) {
9     !seen[$0]
10    print $1
11 }
```

clean.awk script


```

1 awk 'BEGIN{i=1}{line[i++]= $0}END{j=1;
2 while (j<i) {
3     print line[j],line[j+1];
4     j+=1}}'
5 infile
6
7 awk 'BEGIN{i=1}{line[i++]= $0}END{j=1;
8 while (j<i) {
9     print line[j],line[j+1],line[j+2]; j+=1}}'
10 infile
11
12 awk 'BEGIN{i=1}{line[i++]= $0}END{j=1;
13 while (j<i) {
14     print line[j],line[j+1],line[j+2],line[j+3]; j+=1}}'
15 infile
16
17 awk 'BEGIN{i=1}{line[i++]= $0}END{j=1;
18 while (j<i) {
19     print line[j],line[j+1],line[j+2],line[j+3],line[j+4]; j+=1}}'
20 infile

```

extract.awk script

Common Signature Extractor

```

1 #!/bin/bash
2
3 if [ $# -lt 2 ]
4 then echo "$0 [benign-folder] [malware-folder]"
5 exit 0
6 fi
7
8 rm syscallmodel.feats
9
10 bfile="common_"$1".feats"
11 mfile="common_"$2".feats"
12
13 sort $bfile > infile
14 sort $mfile > infile2
15
16 #find malware signatures that is not found in benign set
17 diff -u infile infile2 | grep '+' | sed 's/^[ \t]*//;s/[ \t]*$//' |
18     awk -F\+ '{ print $2 }' | sed '/^[0-9]/d' | uniq > syscallmodel
19     .feats
20
21 #count number of found unique malware signatures that is not found
22     in benign set
23 wc -l syscallmodel.feats
24
25 rm infile infile2 #file clean up

```

Script to extract common signatures from SYSCALL trace

Signature Verification

```
1 #!/bin/bash
2
3 IFS=$'\n'          # make newlines the only separator
4
5 for line in $(cat < "syscallmodel.feat")
6 do
7     echo $line
8     echo $line > infile5
9     awk 'NR==FNR{a[$0];next}
10    { for (word in a) if ($0 ~ word) print FILENAME, $0 }' infile5 m1
11    /feat/* |
12    awk -F'log_1' '{print $1}' | uniq | wc -l
13    rm infile5
14 done
```

Script to check if signature can detect trained malware

Appendix B

The following is the Python script used in Chapter 5. The script was executed on an Ubuntu Linux environment.

Training SVM Model

```
1 import pandas as pd
2 from sklearn import svm
3
4 #Read from train dataset
5 traindata = pd.read_csv("train.txt", sep=" ", header=None).values
6
7 features = traindata[:, :-1] #Separating the features
8 category = traindata[:, -1] #and classes of the train data
9
10 clf = svm.SVC(gamma='auto') #Define a SVM Classifier
11 clf.fit(features, category) #Fit the train data to the classifier
12
13 #Read from test dataset
14 testdata = pd.read_csv("test.txt", sep=" ", header=None).values
15
16 f = testdata[:, :-1] #Separating the features
17 c = testdata[:, -1] #and classes of the test data
18
19 decision = clf.decision_function(f)
20
21 #Predict the unknow test data based on the trained model.
22 pred = clf.predict(f)
23 #Get the classification accuracy based on the test data
24 score = clf.score(f, c)
25
26 print(decision)
27 print(score) #Print the classification accuracy
```

Script to train the model using SVM

1	9.925	1.750	7.003	7.620	8.490	1.092	9.690	2.460	4.373	4.832	0
2	7.620	9.555	1.642	7.281	3.932	8.481	4.309	1.053	1.430	8.482	2
3	2.966	2.872	7.932	7.392	4.665	6.550	1.412	7.184	4.729	1.441	1
4	7.505	2.912	1.977	7.206	8.699	7.096	9.576	3.535	8.299	4.617	0
5	8.011	8.099	8.259	1.459	9.874	9.238	3.375	2.268	1.635	2.412	0
6	6.201	9.223	7.350	8.495	7.755	3.955	3.519	3.635	3.477	9.709	2
7	5.307	5.516	2.851	2.527	9.027	3.792	9.542	4.862	7.802	1.366	0
8	4.706	1.155	1.957	3.723	4.930	5.924	4.245	4.309	6.530	6.429	2
9	3.396	7.766	2.538	3.198	5.707	2.564	3.854	1.886	6.055	4.657	0
10	7.182	6.837	7.833	9.930	9.052	5.389	5.253	4.878	8.283	5.616	2
11	1.090	9.152	1.580	2.998	7.347	6.207	7.914	2.317	5.626	4.019	0
12	5.083	6.245	8.594	4.916	6.827	9.306	5.846	2.076	5.363	2.527	2
13	3.290	7.534	9.609	9.187	8.152	7.764	1.898	7.917	7.166	8.945	2
14	9.477	3.072	8.920	7.727	6.253	3.715	4.224	3.894	8.003	6.246	0
15	2.284	7.719	4.711	3.400	3.085	9.182	3.821	6.288	7.743	1.938	0
16	1.154	8.721	8.797	1.754	1.070	1.797	9.701	8.576	3.232	4.450	1
17	5.413	9.128	7.821	3.050	9.070	7.076	9.160	3.330	2.805	1.693	0
18	1.424	5.844	3.471	8.382	5.639	7.376	6.286	3.688	8.569	2.932	2
19	2.176	5.188	5.760	3.848	2.336	5.174	7.664	6.868	9.125	9.315	0
20	9.964	7.663	7.391	2.874	6.977	7.812	1.603	9.600	6.233	7.343	0
21	9.537	8.647	2.040	7.503	8.991	8.812	3.757	1.635	1.053	7.331	2
22	5.822	1.611	6.718	4.583	9.161	6.036	5.582	1.913	9.231	9.438	1
23	9.396	5.663	8.827	8.553	4.006	2.710	7.197	9.167	6.765	1.601	0
24	8.324	9.897	4.521	6.465	8.893	2.138	6.250	5.403	3.542	3.958	1
25	8.214	1.789	5.933	7.759	3.636	7.519	2.847	1.540	3.363	9.299	2
26	8.040	2.898	8.863	7.461	9.814	3.245	9.598	9.898	2.704	9.415	1
27	8.625	8.253	6.749	4.730	7.330	7.072	8.200	9.839	1.544	1.547	1

Sample of train.txt

1	7.846	6.564	4.421	3.124	6.736	1.501	9.905	6.450	5.364	4.045	1
2	4.052	8.393	7.678	4.183	2.947	9.294	4.401	2.630	6.156	1.498	1
3	7.406	9.397	2.774	1.377	8.487	8.266	9.737	3.863	1.311	6.057	1
4	2.140	7.712	9.974	2.555	5.045	1.039	6.685	7.976	7.388	7.033	1
5	6.226	7.491	4.200	1.972	9.576	2.187	4.110	8.367	4.229	4.805	1
6	4.975	3.998	7.317	4.008	1.455	4.708	7.987	3.537	8.596	8.965	1
7	1.975	8.888	9.589	8.665	3.107	8.560	8.348	3.356	3.286	4.020	1
8	5.720	7.094	4.871	3.163	5.771	1.836	1.681	1.163	1.074	5.864	0
9	7.392	7.100	6.096	9.345	7.476	7.793	3.701	8.244	1.109	3.474	1
10	1.049	3.127	1.770	1.132	1.903	8.682	8.642	8.167	6.113	4.008	1
11	7.620	9.555	1.642	7.281	3.932	8.481	4.309	1.053	1.430	8.482	2
12	6.201	9.223	7.350	8.495	7.755	3.955	3.519	3.635	3.477	9.709	2
13	1.424	5.844	3.471	8.382	5.639	7.376	6.286	3.688	8.569	2.932	2
14	8.214	1.789	5.933	7.759	3.636	7.519	2.847	1.540	3.363	9.299	2

Sample of test.txt