VIRTUAL MACHINE INTROSPECTION BASED MALWARE DETECTION APPROACH AT HYPERVISOR FOR VIRTUALIZED CLOUD COMPUTING ENVIRONMENT

Thesis

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

AJAY KUMARA M.A.

DEPARTMENT OF INFORMATION TECHNOLOGY

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA,

SURATHKAL, MANGALORE - 575025

MARCH, 2018

DECLARATION

by the Ph.D. Research Scholar

I hereby declare that the Research Thesis entitled **VIRTUAL MACHINE INTROSPECTION BASED MALWARE DETECTION APPROACH AT HYPERVISOR FOR VIRTUALIZED CLOUD COMPUTING ENVIRONMENT** which is being submitted to the **National Institute of Technology Karnataka, Surathkal** in partial fulfilment of the requirements for the award of the Degree of **Doctor of Philosophy** in **Information Technology** is a *bonafide report of the research work carried out by me.* The material contained in this Research Thesis has not been submitted to any University or Institution for the award of any degree.

(IT13F01, AJAY KUMARA M. A.)
Department of Information Technology

Place: NITK, Surathkal.
Date:

CERTIFICATE

This is to *certify* that the Research Thesis entitled **VIRTUAL MACHINE INTROSPECTION BASED MALWARE DETECTION APPROACH AT HYPERVISOR FOR VIRTUALIZED CLOUD COMPUTING ENVIRONMENT** submitted by **AJAY KUMARA M.A**, (Register Number: IT13F01) as the record of the research work carried out by him, is *accepted as the Research Thesis submission* in partial fulfilment of the requirements for the award of degree of **Doctor of Philosophy**.

Dr. Jaidhar C.D
Research Guide

Prof. G. Ram Mohana Reddy
Chairman - DRPC

Acknowledgment

I would like to thank all those people who have made this research work possible. First and foremost, I would like to express my sincere thanks to my research guide *Dr. Jaidhar C.D.*, Information Technology Department, for his guidance, suggestion throughout my research work.

I express heartfelt thanks to my Research Progress Assessment Committee (RPAC) members *Dr. Pathipati Srihari* and *Dr. Jidesh P,* for their valuable suggestions and constant encouragement to improve my research work.

I sincerely thank all teaching, technical and administrative staff of the Information Technology Department for their help during my research work.

I would like to thank my parents and my brothers Mr. Kumar A, Mr. Shivraj M. A, and Mr. Shashi kumar A, for their exhaustive support, encouragement and inspiration. Without them, surely, this research work would not have been possible.

Place: Surathkal Ajay Kumara M.A

Date:

Abstract

Cloud computing enabled by virtualization technology exhibits a revolutionary change in information technology infrastructure. The hypervisor is a pillar of virtualization and it allows to abstract the host or bare hardware resources to the Virtual Machines (VMs) which are running on the virtualized environment. As the VMs are easily available for rent from the Cloud Service Provider (CSP) that are a prime target for malignant cloud user or an adversary to launch the attacks and to execute the sophisticated malware by exploiting the identified vulnerability present in it. In addition, the proliferation of unknown malware exposes the limitations of traditional and VM-based anti-malware defensive solutions. These motivated the development of secure hypervisor or Virtual Machine Monitor (VMM) based solutions. The Virtual Machine Introspection (VMI) has emerged as a fine-grained out-of-VM security solution to detect the malware by introspecting and reconstructing the volatile memory state of the live guest Operating System (OS) by functioning at VMM. However, VMM-based introspection solutions present a number of limitations, including the well-known semantic gap issue.

In this work, as a first proposed work (methodology) we study the limitation of existing host-based security solution. To address this issue, we proposed Virtual Machine and hypervisor based Intrusion Detection and Prevention System (VMIDPS) for virtualized environment to ensure the robust state of the VM by detecting and analyzing the rootkits as well as other attacks on live monitored guest OS. The VMIDPS leverages cross-view based technique for detection and identification of intrusion at VM. The experimental results showed that the VMIDPS successfully detected the Windows based rootkits, and Denial of Service (DoS) attack on Monitored VMs. However, the main limitation of this approach is that it uses an agent-based solution on each of the individual Monitored VM to obtain the run state of the guest OS.

In the second proposed work (methodology), we study the limitation of the prior VMI technique that is not intelligent enough to read precisely the manipulated semantic information on their reconstructed high-level semantic view of the live guest OS at VMM. To effectively address this issue, we proposed VMI-based real-time malware detection system called Automated-Internal-External (A-IntExt) system. It seamlessly introspects the untrustworthy Windows guest OS internal semantic view (i.e., processes). Further, it checks the detected, hidden as well as running processes (not hidden) as benign or malicious. The prime component of the A-IntExt system is the Intelligent Cross-View Analyzer (ICVA) that leverages the novel Time Interval Threshold (TIT) technique for detecting the hidden-state information from the internally and externally gathered run state information of the Monitored VM. Experimental results showed that, we can effectively detect and manually analyze the stealthy hidden activity of the malware and rootkits, including measurement with Windows benchmark programs.

In the third proposed work (methodology), we have further extended the A-IntExt system as an advanced VMM-based guest-assisted Automated Multi-level Malware Detection System (AMMDS) that leverages both VMI and Memory Forensic Analysis (MFA) techniques to predict early symptoms of malware execution by detecting stealthy hidden processes on a live guest OS. The AMMDS generalize the cyber physical system application that is functioning at introspected guest OS. More specifically, the AMMDS detects and classifies the actual running malicious executables from the semantically reconstructed executables (i.e., *.exe*) the process view of the guest OS. The two sub-components of the AMMDS are: Online Malware Detector (OMD) and Offline Malware Classifier (OFMC). The OMD recognizes whether the running processes are benign or malicious using its Local Malware Signature Database (LMSD) and OMS. The OFMC classifies unknown malware by adopting machine learning techniques at hypervisor. The AMMDS has been evaluated by executing large real-world malware and benign executables on to the live guest OSs. The evaluation results achieved full detection accuracy in classifying unknown malware with a considerable performance overhead.

In the fourth proposed work (methodology), we have systematically evaluated other shortcomings of our proposed A-IntExt system and AMMDS. In this work, we further extended the A-IntExt system by implementing Hybrid Feature (HF) selection technique that uses representative instances of other individual feature selection techniques of the corresponding feature set that were extracted from the detected hidden and dubious executables of infected memory dumps of the introspected guest OSs.

Further, the proposed approach has been validated with other public benchmarked datasets at VMM. The AMMDS also performs offline detection of malware, however, it fails to address the `over-fitting` issue that plagues many machine learning techniques. In this work, we precisely address the `over-fitting` issue by dividing both generated dataset (VMM level) and benchmarked datasets as training, testing and validation sets. The evaluation results showed that proposed approach is proficient in detecting unknown malware with high detection accuracy on both generated and benchmarked datasets.

In the fifth work, the execution time of the MFA tools such as Volatility and Rekall is measured and compared for the different RAM dump sizes. The motivation behind this works is that RAM dump capture time and its analysis time in real time are highly crucial if an IDS depends on data supplied by the MFA tool or VMI tool. Furthermore, analysis of malware based on the infected memory dump is also a primary for an IDS. In this context, the evaluation conducted on memory dumps of both Linux and Windows VMs that are captured using open source VMI tool called LibVMI.

Contents

List of Abbreviations viii

List of Figures xiii

List of Tables xvi

1 Introduction 1

1.1 Cloud Computing and Virtualization 1

1.2 Hypervisor 2

1.3 Intrusion Detection and Prevention System 3

1.3.1 Hypervisor-based Intrusion Detection System 3

1.4 Types of Malware and Rootkit 4

1.5 Virtual Machine Introspection 5

1.6 Memory Forensic Analysis 7

1.7 Motivation 8

1.8 Dissertation Statement and Contributions 9

1.8.1 Research Objectives 9

1.9 Research Contributions 10

1.10 Outline of the Thesis 13

2 Literature Survey 15

2.1 VMM-based Intrusion Detection System 15

2.2 VMI Perspective 17

2.3 MFA Perspective 22

2.4 Machine Learning Technique Perspective 25

2.5 Outcome of Literature Survey 29

3 Hypervisor based Intrusion Detection and Prevention System 31

3.1 System Design . . . 31

3.1.1 File Integrity Verification . . . 33

3.1.2 Signature based Intrusion Detection System . . . 34

3.1.3 Anomaly based Intrusion Detection System . . . 34

3.1.4 Cross-View Analysis . . . 35

3.2 Experimental Setup and Results . . . 35

3.2.1 Linux rootkits . . . 36

3.2.2 Windows Rootkit . . . 38

3.2.3 DoS attack . . . 40

3.2.4 Port Scanning Attack . . . 40

3.3 Discussion . . . 41

3.4 Limitation of the In-and-Out-of-the-Box Virtual Machine Based IDPS 41

3.5 Summary of the Work . . . 43

4 VMI-based Stealthy Malware and Rootkit Detection System 44

4.1 Introduction . . . 44

4.2 Assumption and Threat Model . . . 46

4.3 Overview . . . 47

4.3.1 GVM-Introspector . . . 48

4.3.2 Guest Assisted Module . . . 51

4.3.3 Intelligent Cross-View Analyser . . . 52

4.3.4 Online Malware Scanner . . . 56

4.4 Experimental Results and Evaluation . . . 56

4.4.1 Experimental Setup . . . 56

4.4.2 Implementation . . . 57

4.4.3 Experiments and Results Analysis . . . 57

4.5 Performance Overhead . . . 61

4.6 Discussion . . . 62

4.7 Summary of the Work . . . 62

5 VMM-based Automated Multi-level Malware Detection System 64

5.1 Introduction . . . 64

5.2 Overview of AMMDS 68
5.2.1 Malware Detector 70
5.2.2 Executable File Extractor 70
5.2.3 Online Malware Detector 70
5.2.4 Offline Malware Classifier 72
5.3 Implementation and Evaluation 74
5.3.1 Experimental Setup 74
5.3.2 Implementation 74
5.3.3 Dataset Creation and Use 76
5.3.4 Evaluation and Results Discussion 76
5.3.5 Experimental Methods 81
5.3.6 Evaluation Metrics 82
5.3.7 Results Analysis 83
5.3.8 Performance Overhead 87
5.4 Discussion 87
5.4.1 Comparison with Existing Work 89
5.5 Summary of the Work 92

6 Leveraging Machine Learning Techniques to Detect and Characterize Unknown Malware at VMM 93
6.1 Introduction 93
6.2 System Design and Implementation 95
6.2.1 Feature Vector Generator 95
6.3 Experiments and Datasets 100
6.3.1 Datasets and Dataset Collection 100
6.4 Evaluation 103
6.4.1 Performance Overhead 107
6.4.2 Experimental Methods 108
6.4.3 Evaluation Metrics 111
6.4.4 Machine Learning Techniques 112
6.5 Analysis of Results 114
6.5.1 Result Analysis of Generated Dataset 115

6.5.2 Result Analysis of Benchmarked Datasets 124
6.5.3 Comparison of Results 131
6.6 Discussion 132
6.6.1 Limitations 133
6.7 Summary of The Work 135

7 Execution Time Measurement of Volatile Artifacts Analyzers 137
7.1 Introduction 137
7.2 Motivation and Overview of HyIDS 138
7.3 Evaluation and Experimental Results 140
7.3.1 Detecting Kernel Level Rootkits 140
7.3.2 Virtual Machine RAM Dump Analysis using Volatility and Rekall 142
7.3.3 Summary 145

8 Conclusion and Future Work 146
Bibliography 149

References 149

List of Publications 161

List of Abbreviations

BFV_T	Benign Feature Vector Temporary
EXT_{psc}	External Process Count
INT_{psc}	Internal Process Count
MFV_T	Malware Feature Vector Temporary
SVM	*Semantic Value Manipulation*
A-IntExt	Automated-Internal-and-External System
AMMDS	Automated Multi-level Malware Detection System
APT	Advanced Persistent Threat
AUC	Area Under Curve
AUC	Area Under the Curve
BFV	Benign Feature Vector
BLINK	Backword Link
BNF	Benign N-gram Feature
CPS	Cyber Physical System
CS	Chi-Square
CSP	Cloud Service Provider
DKOM	Direct Kernel Object Manipulation
DKSM	Direct Kernel Structure Manipulation

DLL	Dynamic Link Library
DO	Descending Order
DoS	Denial-of-Service
DP	Dubious Process
DPC	Dead Process Count
DR	Detection Rate
EFE	Executable File Extractor
FFV	Final Feature Vector
FLINK	Forword Link
FN	False Negative
FP	False Positive
FPR	False Positive Rate
FVG	Feature Vector Generator
FVM	Forensic Virtual Machine
GAM	Guest Assisted Module
GVM	Guest Virtual Machine
HF	Hybrid Feature
HIDS	Host based Intrusion Detection System
HP	Hidden Process
HPC	Hidden Process Count
HTM	Hardware Transactional Memory
HyIDS	Hypervisor based Intrusion Detection System

IAT	Interrupt Address Table
ICVA	Intelligent Cross-View Analyzer
IDPS	Intrusion Detection and Prevention System
IDS	Intrusion Detection System
IDT	Interrupt Descriptor Table
IG	Information Gain
KVM	Kernel Virtual Machine
LMSD	Local Malware Signature Database
MCC	Matthews Correlation Coefficient
MD5	Message Digest 5
MFA	Memory Forensic Analysis
MFV	Malware Feature Vector
MNF	Malware N-gram Feature
Monitored VM	Monitored Virtual Machine
Monitoring VM	Monitoring Virtual Machine
N-F	Nothing-Found
NF	N-gram Frequency
OFMC	Offline Malware Classifier
OMD	Online Malware Detector
OMS	Online Malware Scanner
OOB	Out-of-Bag
OS	Operating System

OSSEC	Open Source Security Event correlator
P2P	Peer-to-Peer
PE	Portable Executable
PFR	False Positive Rate
PID	Process Identifier
PN	Process Name
PoC	Proof of Concept
PPID	Parent Process Identifier
PS	Process
RMS	Root Mean Square
ROC	Receiver Operating Curve
SCT	System Call Table
SHA-1	Secure Hash Algorithm-1
SHA-256	Secure Hash Algorithm-256
SI-Requester	State Information Requester
SIDS	Signature based Intrusion Detection System
SMO	Sequential Minimal Optimization
SVM	Support Vector Machine
TIT	Time Interval Threshold
TN	True Negative
TP	True Positive
TPR	True Positive Rate

VM	Virtual Machine
VMI	Virtual Machine Introspection
VMIDPS	Virtual Machine and Hypervisor Independent Intrusion Detection and Prevention System
VMM	Virtual Machine Monitor

List of Figures

1.1 Semantic gap of VMI 6

3.1 Architecture of the VMIDPS for virtualized environment 32

3.2 KBeast rootkit file compilation (a), rootkit hides port details (b), OSSEC alert message for rootkit injection(c) 37

3.3 `Hacker Defender` rootkit injection (a) OSSEC alert message for `Hacker Defender` rootkit (b) 39

3.4 OSSEC detected DoS attack as URI too long 40

4.1 The proposed VMI based A-IntExt system 47

4.2 Time interval threshold used by A-IntExt system 49

4.3 Hidden processes (a) dubious processes (b) details of Monitored VM externally introspected (left side) and internally acquired (right side) by the A-IntExt system after rootkit and stealthy malware injection on Windows guest OS 50

4.4 Simplified `_EPROCESS` structure of Windows system (Florio 2005) . . 52

4.5 Online malware scanner 54

4.6 The average time taken by the OMS to compute MD5, SHA-1, and SHA-256 hashes for different processes (5a). Time taken by OMS to detect malware by cross-checking with LMD based on it's computed hashes (5b) 60

4.7 Performance impact of A-IntExt system on PCMark05 in detecting hidden and malicious state information of Monitored VM for Windows guest OS 61

5.1 The proposed VMI based A-IntExt system 69

5.2 Flow chart of the AMMDS for detection of malware using OMD and OFMC components 71
5.3 The average time consumed by the OMD to generate SHA-256, SHA-1, and MD5 hash digest for execution of different types of malware (4a). Time taken by the OMD to identify the known malware by cross-checking with LMSD based on computed hash digest (4b) 79
5.4 Snapshot of OMD for identification of malicious (not hidden) process `kelihos_dec.exe` from OMS 80
5.5 Snapshot of OMD for the detection of known malware by cross-checking with LMSD 80
5.6 Malware detection accuracy achieved by different classifiers based on NGL CC and Odds Ratio feature selection techniques for three different feature length 84
5.7 Comparison of performance of the classifier under different performance metrics for the different feature length recommended by NGL CC and Odds Ratio feature selection techniques 86
5.8 Performance overhead of the AMMDS on PCMark05 in detecting hidden and dubious state information of Monitored VM 87

6.1 4 byte N-gram sequence extraction from an executable 96
6.2 Performance overhead of the A-IntExt system using PCMark05 benchmark 107
6.3 Training phase and prediction phase of the A-IntExt system 109
6.4 OOB error rate comparison with different tree size for HF selection technique on the validation set of generated dataset 119
6.5 RMS error rate comparison with different tree size for HF selection technique on the validation set of generated dataset 119
6.6 ROC curves for feature selection techniques used by different classifiers for L= 300 on the validation set of generated dataset 122
6.7 A-IntExt system malware detection evaluation using different performance metrics for L=300 on the validation set of generated dataset . 123

6.8 OOB error rate comparison with different tree size for HF selection technique on the validation set of benchmarked datasets 127
6.9 RMS error rate comparison with different tree size for HF selection technique on the validation set of benchmarked datasets 127
6.10 ROC curves for feature selection techniques used by different classifiers for L= 300 on the validation set of benchmarked datasets 128
6.11 A-IntExt system malware detection evaluation using different performance metrics for L=300 on the validation set of benchmarked datasets 131

7.1 High level view of HyIDS architecture for virtualized environment . . 139
7.2 `average coder` rootkit module hidden at Domu1 VM the same detected by out-of-the-box VMI solution LibVMI 141
7.3 `average coder` rootkit hidden module extracted by the Volatility from raw of physical memory dump 142
7.4 AC rootkit infected module extracted by the Rekall from raw of physical memory dump 142
7.5 RAM Dump Analysis Time 144

List of Tables

3.1 Experimental testbed using OSSEC HIDS 35

3.2 List of Windows and Linux rootkits used in this experiments and detected by our proposed approach 42

3.3 Security attack scenarios experimented and detected by our proposed approach 42

4.1 Detection of hidden, dead and DPs by the A-IntExt system for Windows guest OS 58

4.2 Identifying an actual malicious process from the detected hidden processes by the OMS of A-IntExt system on Windows guest OS 59

4.3 Identifying an actual malicious processes from detected and classified DPs by OMS of A-IntExt system on Windows guest OS 59

4.4 List and functionality of Windows rootkit 60

5.1 Execution of malware and benign executables on live Monitored VM in different experimental test cases 78

5.2 Identifying an actual malicious process from test-I of detected hidden processes by the OMD of AMMDS 78

5.3 Identifying an actual malicious process from test-I of detected dubious executables by the OMD of AMMDS 78

5.4 Confusion matrix 83

5.5 TPR and FPR of different classifiers on different feature length 85

5.6 Comparison of AMMDS with previous VMI-based malware detection approaches 88

5.7 Comparison of results with other VMM-based and non-introspection based malware detection approaches that used machine learning techniques 90

6.1 Types of malware used in the experiments 101
6.2 The composition of generated dataset 102
6.3 The composition of benchmarked datasets 102
6.4 Execution of benign and malicious executables on live Monitored VMs in different experimental test cases for generating testing dataset at VMM 105
6.5 Execution of benign and malicious executables on live Monitored VMs in different experimental test cases for generating validation dataset at VMM 105
6.6 List of default (D) parameters used by the classifiers on WEKA. The Increased (I) values of default parameters are highlighted in bold . . . 115
6.7 Detection accuracy (%) of malware using different feature selection techniques on both testing and validation sets experiments of generated dataset 116
6.8 Experiment results of malware detection with different feature selection techniques for validation set of generated dataset 120
6.9 Detection accuracy (%) of malware using different feature selection techniques on both testing and validation sets experiments of benchmarked datasets 125
6.10 Experiment results of malware detection with different feature selection techniques for validation set of benchmarked datasets 129
6.11 Comparison of accuracy (%) for validation sets of generated and benchmarked datasets for L=300 of HF selection technique 132

7.1 Experimental setup 140
7.2 Real-world rootkit experiments under VMs 141
7.3 RAM dump analysis time of the Ubuntu VMs 143
7.4 RAM dump analysis time of Windows VMs 143

Chapter 1

Introduction

1.1 Cloud Computing and Virtualization

Cloud computing improves resource utilization while reducing infrastructural cost. It is based on existing technologies such as service-oriented architecture, virtualization, and utility computing. Virtualization is a key underlying technology for cloud computing architecture. It facilitates sharing of physical machine resources such as CPU, Memory, I/O and Network interface etc., among several VMs running on the same physical machine using a special software layer called hypervisor or VMM. Sharing of resources increases the critical security challenges for CSP. Protecting virtualized resources of guest OS against sophisticated malware such as a virus, spyware, stealthy rootkit, Trojan, polymorphic and metamorphic variants is a massive challenge for the CSP (Takabi et al. 2010).

Virtualization technology allows to create and run multiple guest OSs concurrently using physical hardware of the host system or bare hardware resources in a virtualized infrastructure. Its function is to create virtual resources such as virtual CPU, virtual network interface card, virtual memory etc., to assign for different dedicated VMs (Vollmar et al. 2014). Virtualization provides tremendous benefits and sharing of hardware resources. However, it increases greater security risk and significant challenges for the CSP. Ensuring security portfolio among different dedicated VM resources is a massive challenge for CSP. With the advent of virtualization, traditional security solutions are not designed by keeping virtualization in mind. Most of the solutions are signature dependent that is not feasible to ensure the complete security of virtualized resources on a virtualized cloud computing environment (Pearce et al. 2013).

The virtualization is classified into two types that are full virtualization and para virtualization. In case of full virtualization, the virtualization layer completely abstracts VM or guest OS from the underlying hardware. The guest OS is utterly unaware of its virtualized state and no modifications are needed. Further, virtualization of sensitive and privileged instructions do not require hardware assistance as well as OS assistance. All OS instructions are translated by the hypervisor and cache them for future usage, whereas user instructions run without any modification at native speed. It keeps VMs in an isolated environment which ensures security of the VMs. Examples for full virtualization are VMware vSphere ESXi 6.5 and Microsoft Virtual Server.

In para virtualization, the guest OS (the one being virtualized) is aware of its virtualized state. The modification is required at the OS kernel level to replace non-virtualizable instructions into hyper-calls in order to establish direct communication with a hypervisor. Interrupt handling and memory management are examples of kernel operations. The open source Xen project is an example of para virtualization (Vollmar et al. 2014; Horne 2007).

1.2 Hypervisor

Hypervisor or VMM is a low-level software program that emulates or abstract physical hardware resources such as CPU, Memory, Disk, Network Interface Card, I/O to multiple guest OSs in real-time which are running on the same host OS or bare hardware. The hypervisor has full control over entire virtual resources of the guest OSs. In general, the hypervisor acts as a bedrock between the VMs and the physical hardware. The hypervisor broadly classified into two types: Type-1 hypervisor and Type-2 hypervisor.

The Type-1 hypervisors run directly on the bare system hardware. For example Xen Hypervisor, VMware vSphere ESXi 6.5, Citrix-Xen Server, Microsoft-Hyper-V 2008 R2 etc. Type-2 hypervisors run on a host OS to emulate host machine hardware resources to VMs. For example, KVM hypervisor, Oracle Virtual Box and VMWare Player are Type-2 hypervisors (Vollmar et al. 2014).

1.3 Intrusion Detection and Prevention System

The term "Intrusion" can be defined as an unauthorized attempts to access or compromising confidentiality, integrity and availability by bypassing the existing security defenses solution to disrupt normal operation of a computer system or network system (Scarfone and Mell 2007). Intrusion Detection System (IDS) is a type of security measure used to detect intrusions automatically and alert administrators. Intrusion prevention is also security measure used to prevent the identified intrusion before the intrusion trespass. Intrusion Detection and Prevention Systems (IDPS) is a process of continuous monitoring of incoming and outgoing traffic and events occurring on an OS or network to detect and prevent security threats, attacks in a timely manner without human intervention (Modi et al. 2013).

1.3.1 Hypervisor-based Intrusion Detection System

The hypervisor is a software component used to ensure sharing of host system resources in a virtualized environment among VMs. Computing security in hypervisor provides secure virtualization throughout its life cycle, including development, implementation, provisioning, de-provisioning, and management. Securing hypervisor, VMs, and virtual switch is an active research area. Recent attempts made in the direction of computing security in hypervisor are discussed in this Section (Patel et al. 2012).

Hypervisor creates VMs as per the specifications chosen by the CSP on the host system. The VMs running on the hypervisor access the hardware resources of host system via hypervisor in their life time. Communication between any VM with any other VM in a virtualized environment is also through the hypervisor. Because the hypervisor has full control over host system resources allocation to VMs and it can directly access the memory allocated to guest VMs. This enables malicious VMs to exploit vulnerabilities of the hypervisor to attack the hypervisor or another VM. The hypervisor vulnerabilities are most dangerous because the malignant user can leverage the vulnerabilities of the hypervisor to launch an attack or compromise the entire host system (Jin et al. 2011). By compromising the hypervisor, the malignant user can directly steal the confidential data present in the host system or modify the software to disrupt the normal operation of the entire virtualized environment.

1.4 Types of Malware and Rootkit

Due to the propagation of cloud computing, VMs are increasingly becoming attractive targets for cyber crooks due to easy access from CSP (Pearce et al. 2013). The current generation of malware uses code obfuscation techniques (Lin and Stamp 2011) and rootkit functionalities (Goudey 2012b) to subvert most of the existing in-host or VM-based anti-malware security solutions to gain privileged access to the targeted machine. With successful penetration, malware operates by leveraging uncovered vulnerabilities of the guest OS to perform illegitimate activities, and also attack other VMs running on the same virtualized infrastructure. Preserving the VMs by detecting such sophisticated malware is a challenging task for the CSP (James 2010). Malware are classified into several types based on their different behavioral characteristics. The taxonomy of discriminative malware is classified as follows:

- **Trojan horse:** It is a piece of the program, which looks legitimate, but is malicious. Its objective is to install other malicious applications on the victim computer to be controlled by cyber-crooks to steal confidential information.
- **Virus:** A virus is a program, that propagates by interpolating a copy of itself to another program and becomes a part of it.
- **Worm:** A worm is a self-replicating malicious program which exploits the vulnerabilities on a target system.
- **Spyware:** It secretly gathers Internet usage and confidential information from the victim machine.
- **Metamorphic and Polymorphic:** Malware writer of these types use multiple transformation techniques like code permutation, shrinking and renaming, etc. Once this malware infects the victim machine, it has the ability to change its code as it propagates.
- **Backdoor:** Once a backdoor infected, it additionally installs other spyware; allows an attacker to control the entire victim machine.

- **Ransomware:** It prevents usage of accessing the system from an end user either by encrypting or holding user confidential files for ransom.
- **Botnet:** Botnet uses rootkit-based threads to hide its presence once installed. It acts as malware downloader by establishing its own unstructured P2P networks to perform click fraud activity.
- **Stealthy rootkits:** Rootkits are one type of malware that runs at highest privileges access of OS (kernel or ring '0') by exploiting the security weakness of the system. Rootkits tamper the integrity of the OS by stealthy modifying critical kernel data structure of the OS. They maintain a persistent and undetectable presence on the victim system. Stealthy rootkits are able to hide processes or programs or files or ports or directories etc., (Sparks and Butler 2005). The rootkit deviates the normal behavior of the system by injecting malicious code into an OS or application software. Some of the rootkits hide themselves to evade the security check of anti-malware solution (Kim et al. 2012).

  Rootkits are mainly classified into user-mode rootkits (application-level) and kernel-mode rootkits (Hwang et al. 2013). The user-mode rootkits modify or intercept OS critical files to hide themselves since they unable to modify kernel structure directly. For instance, executable files, system libraries, and Dynamic Link Library (DLL) file on Windows OS. Kernel-mode rootkits can alter the code of the core OS i.e. kernel device driver, System Call Table (SCT), kernel data structure etc. The severity of the kernel-mode rootkits is increasingly high as compared to user-mode rootkits as they modify OS source code and they are difficult to detect due to their hidden existence. Some rootkits achieve Direct Kernel Structure Manipulation (DKSM) attack (Prakash et al. 2015; Bahram et al. 2010). They have the capacity to foil or bypass In-the-box solution and the rootkits are often directly alter the kernel memory using the /dev/kmem memory device.

1.5 Virtual Machine Introspection

Traditional in-host or VM-based malware defensive solutions are not virtualization-aware, as these solutions often rely on the signature-based technique, and are vulnerable to unknown malware that uses zero-day exploits. These techniques achieve poor performance when attempting to identify new or unknown malware. The growing evasion capability of new and unknown malware needs to be countered by analyzing the behavior of the malware dynamically to address this issue. The VMI (Garfinkel et al. 2003) has evolved as a promising security solution to introspect any untrustworthy guest OS by operating at the VMM. It performs an indirect investigation of the untrustworthy monitored VM in real-time by operating at the hypervisor in a virtualized cloud environment. The useful isolation and inspection properties of the VMM ensure that the VMI-based security solutions are secure and tamper resistant. For example, an `isolation` property leveraged by the VMI confirms that the malicious software running on the introspected VM cannot access or tamper the VMI solution running on the VMM, even though the malware has completely corrupted the guest OS. The `inspection` property facilitates the VMI to examine the entire run state of the guest OS such as memory, CPUs, registers, I/O, etc., (Zhao et al. 2009).

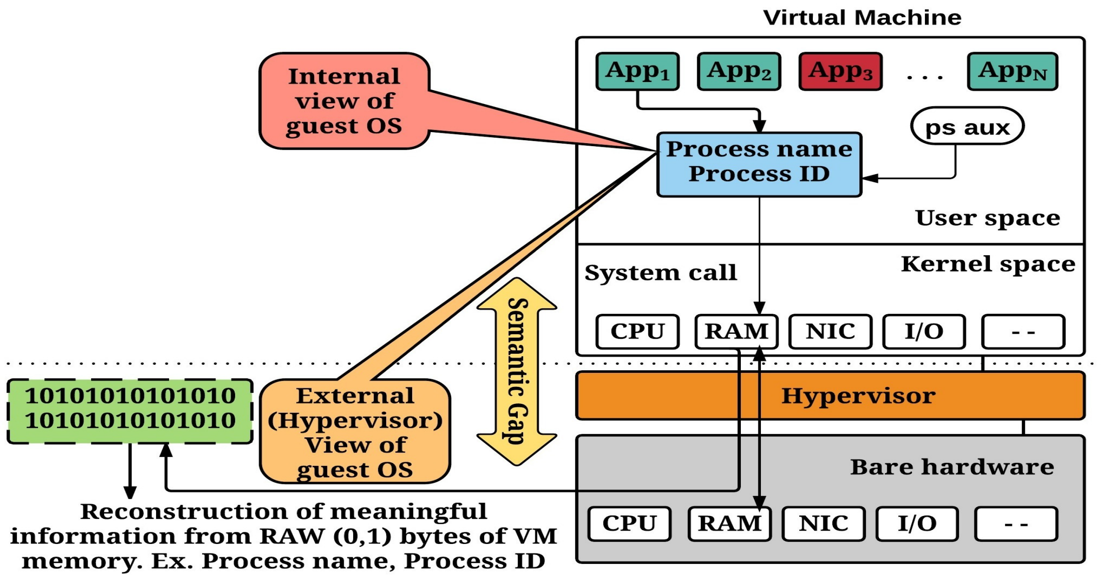


Figure 1.1: Semantic gap of VMI

When the VMI introspects a guest OS, it intercepts its memory state to reconstruct the guest OS abstraction from the raw memory. However, intercepting low-level details of the current run state of the guest OS and converting it into a meaningful form (e.g., processes, modules, system calls, etc.) presents an obstacle called as the semantic gap (Dolan-Gavitt et al. 2011; Jain et al. 2014; Fu and Lin 2013). Figure 1.1 shows the semantic gap of the VMI. The existing VMI-based techniques are not efficient enough to reconstruct the high-level rich semantic views of the large kernel data structure (e.g., registry, file system, kernel object, etc.) of the introspected VM that is constantly manipulated by the sophisticated malware and rootkits (Prakash et al. 2013). To develop such a proficient introspection system, the VMI developer requires tremendous manual effort and an extensive knowledge of the guest OS (Bauman et al. 2015a). The prime function of the VMI is to perform early detection and prediction of malicious executables so as to prevent it from tampering with other essential data structures by examining and analyzing memory content of the live Monitored VM. However, the existing VMI techniques are not intelligent enough to read precisely the manipulated semantic information on their reconstructed high-level semantic views of the live Monitored VM.

1.6 Memory Forensic Analysis

The VMI has evolved as a base for various novel methodologies in the fields of digital forensics and cyber security (Poore et al. 2013). In digital forensics field, digital investigator prime concern is to detect and analyze the digital artifacts in order to investigate the damage caused by the unauthorized or malicious activities. Live investigation and dead investigation are two basic categories of a digital forensics investigation. Basically, a live investigation is performed on a system while the system is in running state. Inspecting a system while it is in the active state provides vital information in light of the proof that is available. For example, data related to opened files, active processes, opened ports, network connections statistics, and volatile malware etc., can be retrieved from a running machine (Poisel et al. 2013). It plays a crucial role in digital investigations and incident response. Thus, live investigation (memory forensics) sometimes referred as a live incident response. In addition, it can also be used to spot remnants of kernel objects. For example, exited processes details can be extracted even after they have departed from the active process list. Apart from extracting and analyzing aforesaid artifacts, it can also assist in identifying the malware, analysis of malware and reverse engineering.

The dead investigation is done while a system is in off state. It needs an image of the system storage media to analyze and to extract data from it. Since dead analysis deals with static data it is easier to investigate than a live investigation in which state of the system constantly running and changing. Dead investigation on VM image is as same as an investigation of a physical system image. However, the dead investigation does not provide crucial information like a live investigation. In our work live investigation techniques such as VMI and Memory Forensics Analysis (MFA) are employed.

The possibility of leveraging the MFA with the VMI provides more flexibility to bridge the semantic gap by examining the rich semantic views of the kernel data structures based on the memory dump of different live guests' OS (Hay and Nance 2008; Dolan-Gavitt et al. 2011). Advantages of using VMI with digital forensics are described in (Poore et al. 2013; Nance et al. 2009). However, most of the MFA tools require a memory dump to perform memory forensics. In a virtualization environment, frequent acquisition of live VM's memory dump, which requires temporary suspension of the running VM to seize the consistent memory state, leads to performance degradation. Furthermore, there is a lack of digital evidence on the captured volatile artifacts, possibly due to malware infection. In addition, one of the main drawbacks of the MFA-based technique is that it involves extensive manual effort for malware analysis.

1.7 Motivation

The proliferation of modern malware or virulent software like a virus, worms, trojans, botnets, rootkits etc., are designed by a cyber criminal to perform nefarious tasks in the normal run state of the computer system by evading traditional in-host based anti-defensive solution (Srinivasan et al. 2011). Example Agobot variant rootkit contains spiteful logic to detect and remove more than 105 antivirus (F-Secure 2003). Propagation of these intrusive programs precisely target individuals CSP, governments and their data to perform disruptive impact on their business by initiating a variety of attacks such as Distributed Denial of Service attacks (DDoS), code injection attack etc. A recent survey states that over 500 chief executives and IT managers are hesitant to move their business to the cloud due to fear of security threats. Further, they lose their control of their data once transferred to cloud (Goudey 2012a). In addition, based on the 2016 Symantec threat report (Symantec 2016), Symantec discovered more than 430 million new unique pieces of malware by the end of 2017, this number has grown up to 36% more from their previous year report. However, detecting a volume, acceleration of malware and Advanced Persistent Threats (APT) is ever present challenges in the field of information security. Malware writer expose the vulnerability of the target OS by adopting code obfuscation techniques such as polymorphism and metamorphism during the development of malicious software.

1.8 Dissertation Statement and Contributions

The goal of this research work is to design, implement, evaluate a real time VMM-based an advanced VMI system to detect real-world malware and rootkits on the live introspected system by leveraging MFA techniques and adopting machine learning techniques at VMM to secure the run state of the VM on virtualized cloud computing environment.

1.8.1 Research Objectives

- Propose hypervisor and virtual machine dependent intrusion detection and prevention system for detection of rootkits on guest OS.

- Propose a VMI-based stealthy malware and rootkit detection system by analyzing semantic views of the guest OS.

- Propose an automated multi-level malware detection system to detect and classify unknown malware based on a reconstructed semantic views of the executable on the live introspected guest OS in a virtualized cloud computing environment.

- Introduce novel VMM-based malware detection system by leveraging machine learning techniques and measure its proficiency by using both generated and benchmarked malware datasets.

- Measure and compare the execution time of volatile artifacts analyzers (MFA and VMI) of the live introspected guest OSs.

1.9 Research Contributions

Accordingly, the following contributions of this research study are available to the research community in the form of journal and conference publications.

- **Hypervisor based Intrusion Detection and Prevention System (Chapter 3).** In this work, we proposed In-and-out-of-the-box Virtual Machine and hypervisor based Intrusion Detection and Prevention System (VMIDPS) for virtualized cloud environment to ensure robust state of the VMs by detecting followed by eradicating rootkits as well as other attacks on the VMs. The main aim of this approach is to safeguard VMs and hypervisor by detecting and eradicating the intrusions at VM level. The proposed approach leverages cross-view based technique for real-time identification of intrusions by using the file integrity verification and signature-based intrusion detection techniques. It uses a popular open source Host-based Intrusion Detection System (HIDS) called Open Source Security Event Correlator (OSSEC) (Bray et al. 2008) to detect and analyze the events in real-time. The experiments were conducted by executing real-world Linux and Windows based rootkits. Denial of Service (DoS) attack and files integrity verification tests are successfully detected by the proposed approach while recording the identified intrusion details onto the log files.

- **VMI-based Stealthy Malware and Rootkit Detection System (Chapter 4).** We designed, implemented and evaluated the VMI-based A-IntExt system for a virtualized environment that address the semantic gap of the VMI. It seamlessly introspects the untrustworthy Windows guest OS internal semantic views (e.g., processes) to detect the hidden, dead, and malicious processes on the live introspected guest OSs. Further, it checks the detected, hidden as well as running processes (not hidden) as benign or malicious. The prime component of the A-IntExt system is the ICVA, which is responsible for detecting hidden state information from internally and externally gathered state information of the monitored guest OS. The A-IntExt system has been evaluated by using publicly available Windows malware and real-world rootkits to measure the detection proficiency as well as execution speed. The experimental results demonstrated that A-IntExt system is effective in detecting malicious and hidden state information rapidly with maximum performance overhead of 7.2%.

- **VMM-based Automated Multi-level Malware Detection System (AMMDS) (Chapter 5).** Next, we extended the A-IntExt system as AMMDS to detect and classify malware on the live introspected system at VMM. The presented AMMDS considers the Cyber Physical System (CPS) as an application which is functioning in the virtualized environment (e.g., guest OS) targeted by malicious executables. In order to secure the critical infrastructure of the virtualized cloud environment, the AMMDS leverages both the VMI and MFA techniques to perform three levels of investigation. As the first level of investigation, it performs introspection and precisely detects the semantic view of the hidden and malicious process to estimate the perfect infection state of the live introspected guest OS. It seizes the execution state of the introspected guest OS by capturing the memory dump of the monitored guest OS soon after it identifies the unusual behavior and then instantly reconstructs and extracts executables from the acquired memory dump to carry out next level of investigations. As a second level of investigation, the OMD component of the AMMDS examines the extracted executables to ascertain the malicious one. The OFMC component of the AMMDS analyses the extracted executables in order to identify unknown or zero-day malware using machine learning techniques as a third level of the investigation. The AMMDS has been evaluated by using real malware datasets on the virtualized environment established using the Xen hypervisor. Our empirical results showed that AMMDS is robust in detecting and classifying unknown malware that can evade VM-based security solution, and it only incurs the acceptable moderate run-time overhead of 5.8%.

- **Leveraging Machine Learning Techniques to Detect and Characterize Unknown Malware at VMM (Chapter 6).** In this approach, we have systematically evaluated other shortcomings of our proposed A-IntExt system and AMMDS. For example, the reconstructed semantic details by the A-IntExt system are available in a combination of benign and malicious states at the hypervisor. In order to distinguish between these two states, an extensive manual analysis is required by the early A-IntExt system when a large number of malware infected on the live guest OSs. Even though this issue was tackled by AMMDS by performing both static online and offline malware detection and classification, AMMDS is inadequate to address the other common issue called *over-fitting* that plagues many machine learning techniques that degrade the detection accuracy of the classifier. In this approach, we have validated the A-IntExt system with other tested benchmarked malware datasets not generated by us. This validation strengthens the trustworthiness and confirms how well our proposed A-IntExt system worked on other public benchmarked malware datasets at VMM. Further, the extended A-IntExt system implemented with HF selection technique that uses representative instances of other individual feature selection techniques such as Information Gain (IG), N-gram Frequency (NF), and Chi-Square (CS). The HF selection technique uses the corresponding feature set of other individual feature selection techniques that were extracted from the detected hidden and dubious executables of infected memory dumps of the introspected guest OSs. Further, we used six well-known machine learning techniques at VMM from VMI perspective to precisely detect unknown malware from the semantically reconstructed executables. In addition, we precisely address *over-fitting* issue by dividing both generated dataset (VMM level) and benchmarked datasets as training, testing, and validation sets. The evaluation results showed that our proposed approach is proficient in detecting unknown malware on both the datasets by achieving the detection accuracy 99.55% with 0.004 False Positive Rate (FPR) on a generated dataset including 6.3% of performance overhead.

- **Execution Time Measurement of Volatile Artifacts Analyzers (Chapter 7).** The VMI has emerged as a promising approach that monitors run the state of the VM externally from the hypervisor. However, the main limitation of the VMI lies with the semantic gap. An open source tool called LibVMI address the semantic gap. The MFA tool such as Volatility can also be used to address the semantic gap. But, it needs a memory dump (RAM) as input. Memory dump acquires time and its analysis time is highly crucial if the IDS depends on the data supplied by the MFA or VMI tool. Furthermore, memory analyzer accuracy is also a primary for the IDS. Thus, our aim is to 1) Measure the time required to capture a live VM RAM dump using VMI tool. 2) Measure the performance of the MFA tool such as Volatility in terms of execution time elapsed to analyze the RAM dumps of different size. 3) Compare the performance of the Volatility with another open source MFA tool called Rekall in terms of execution speed. In this work, live VM RAM dump acquires time by the LibVMI is measured. In addition, memory dump analysis time consumed by the Volatility is measured and compared with Rekall. Further, Volatility and Rekall are compared with LibVMI. It is noticed that examining the volatile data through LibVMI is faster as it eliminates memory dump acquire time.

1.10 Outline of the Thesis

This dissertation is organized into eight chapters, including this introductory chapter. In chapter 2, we present a detailed literature survey of Hypervisor based IDS, VMI, MFA, and machine learning techniques. Chapter 3 presents Hypervisor based Intrusion Detection and Prevention System for rootkit detection. In chapter 4, we presented the design, implementation, and evaluation of the VMI based A-IntExt system and proposed approach is evaluated using publicly available rootkits and malware. In chapter 5, we further extended A-IntExt System as AMMDS that performs detection of malware using both OMD and OFMA using machine learning techniques. In chapter 6, we further extended A-IntExt system by leveraging machine learning techniques at VMM. In this approach we precisely addressed the `over-fitting` issues of machine learning techniques by dividing dataset as training, testing and validation set. The evaluation of this proposed approach is experimented by executing a large real-world malware and benign executables at different versions of Windows guest OS to generate VMM based malware dataset (generated dataset). In addition, the feasibility of the proposed approach is evaluated with other benchmarked datasets. In chapter 7, we presented HyIDS framework to perform execution time measurement of volatile artifacts analyzer for detection and analysis of malware and rootkit. Finally, we conclude this dissertation and describe future directions of this research work in chapter 8.

Chapter 2

Literature Survey

In this chapter, we present a detailed literature survey of proposed approaches specific to the concept of Hypervisor based IDS, VMI, MFA, machine learning techniques for malware detection.

2.1 VMM-based Intrusion Detection System

The VMs constantly interact with the hypervisor during their life time. This enables malicious VMs to exploit vulnerabilities of the hypervisor to attack hypervisor or to attack another VMs. Attacking hypervisor through the direct exploitation of hypervisor vulnerabilities is known as hypervisor attacks. In order to detect hypervisor attack the IDS needed to positioned at hypervisor level to detect targeted attacks and to safeguard VM at the hypervisor level.

Bharadwaja et al. (2011) proposed collaborative IDS named as Collabra. Malicious applications running on the VM make use of hyper-call to misuse hyper-call interfaces to compromise guest OS kernel followed by host OS kernel. It is integrated with VMM for dynamic filtering of malicious hyper-calls to defeat sophisticated attacks. Collabra classifies the hyper-calls raised by the VM as malicious or non-malicious by performing integrity check and hypervisor classification mechanism.

Nikolai and Wang (2014) proposed a Hypervisor-based Cloud Intrusion Detection System (HCIDS) that leverages the virtualization technology at the cloud environment. The main advantage of this approach is the detection of attacks can be achieved by monitoring the activities outside the Monitored VM. Moreover, it also efficient to detect insider attacks. The HCIDS can be easily integrated with the existing IDSs to make them advance in security aspects within the cloud environments. The comparative analysis justified that the developed architecture does not need any additional software to examine the state of the VM based on the details accumulated in memory, registry, and I/O devices.

Wang and Karri (2013) proposed a VMM based prototype called NumChecker. Its main task is to identify the control-flow alteration made by the kernel-mode rootkits to a system call in a guest VMs. It does this task by making use of Hardware Performance Counters (HPCs). These HPCs are employed to count automatically the number of specific hardware events that occur during system call execution. HPCs reduce the checking cost and also strengthen the tamper-resistance. NumberCheker prototype implemented and evaluated on Linux with Kernel based Virtual Machine (KVM) hypervisor.

Xie and Wang (2013) proposed a rootkit detection technique for VMs demonstrates that extracting significant information and reconstruction of extracted information at the hypervisor enables the identification of rootkits. Since VMs are managed by the hypervisor, VM kernel symbol table can directly access from hypervisor which aids in reconstructing execution state of the VMs. Typical examples of such reconstructed state information are: active network connections, opened ports, running processes, and opened files. Examining the various reconstructed running state information allows to ascertain both the concealed information as well as the abnormal state. Similarly, (Schmidt et al. 2011) proposed an another approach that has dual features. It has been designed and implemented for the malware detection and kernel rootkit prevention inside the cloud computing environments. The proposed method provides a safe environment to identify malware during the runtime and thwarts hazard by not allowing it to get installed in the operating system kernel. However, to detect the malware system calls are captured in real-time upon efficiently utilizing the existing cloud resources associated with different analysis engines. Thus, loading of unauthorised modules are restrained.

Hwang et al. (2013) proposed out-of-the-box rootkit detection system based on the hypervisor for cloud computing environments. This approach makes use of vIPS platform to operate as a virtual security appliance. The detection system can investigate the rootkits within VM without deteriorating for the liability caused by the rootkits. The designed method overcomes the drawbacks of in-the-box rootkit detection system. Since it is agent-less, no need to install anything inside the target VM to identify the rootkits. Rhee et al. (2009) presented a system which efficiently deals with the inhibiting dynamic data kernel attacks. This is accomplished by employing VMM-based monitoring to permit an unauthorized memory accesses on protected kernel data. Further, essential VM introspection techniques were considered to track the protected data dynamically in real-time. Besides, the QEMU (Quick Emulator) VMM was also implemented to defeat dynamic data kernel attacks successfully. Therefore the designed method manifest to have the ability to fortify against the rootkit attacks without the modification occurred to the under-layered operating system.

CXPInspector monitoring system uses out-of-the-box approach to observe and analyze the VM state (Willems et al. 2013). It is constructed based on the idea of Currently eXecutable Pages (CXP), which has the ability to observe the activities of a program or even an entire OS. CXPInspector has the capacity to analyze the behavior of both user-mode and kernel-mode rootkits execute on 64-bit processor machine. Further, it has the ability to provide a performance profile of a single program or a complete OS. CXPInspector leverages EPT technology in order to virtualize the memory management unit and ensures address space partitioning.

To detect the malicious drivers, a simple static analysis approach has been designed and evaluated (Musavi and Kharrazi 2014). This method was proposed based on two observed facts. The first fact is that generally, rootkits penetrate into Windows OS through kernel drivers. The second fact is that usually genuine kernel-level code uses slight obfuscation techniques as compared to the malicious kernel-level code. The authors have described the modern developments in implementing the of kernel-level rootkits. In addition, the authors have proposed a set of features to measure the kernel drivers malicious behavior.

2.2 VMI Perspective

The VMI was pioneered by Garfinkel and Rosenblum by developing a prototype called Liveware (Garfinkel et al. 2003) for detection of an intrusion by examining the low-level state of the guest OS from outside the VM. Since then, numerous efforts have focused on the significant adoption of the VMI for malware detection and analysis

(Dinaburg et al. 2008; Jiang et al. 2007), process monitoring (Srinivasan et al. 2011), rootkit detection (Rhee et al. 2009), etc.

Several out-of-VM security approaches have also proposed to monitor and ensure the protection of the introspected system while addressing specific security problems of the guest OS. The Antfarm (Jones et al. 2006) approach tracks and implicitly obtains the execution of the guest processes' information while functioning at the VMM. It makes use of the CR3 register of the x86 architecture to store the page table directory base address that corresponds to currently running process. The content of the CR3 register allows to ascertain process creation, switching, and termination. This binding offers a view of all process dealing with events. The VMM can make use of Antfarm to address the part of the semantic gap issue. Maximum hypervisor-based process identification techniques depend on the CR3 register content so that it catches the fundamental OS-stage object life-cycle such as a process. Since Antfarm resembles passive monitoring approach, it is incapable to assure interposition on events before they occur.

Process Implanting is another VM introspection framework (Gu et al. 2011). It implants a process from the host system into the Monitored VM to tackle the semantic gap. The implanted process does its job stealthily on the guest OS and exits with negligible negative impact. It uses a number of coordination and protection mechanisms offered by the hypervisor to protect the implanted process as a tamper resistant against the tricks used by the malware. This prototype is implemented and tested in KVM hypervisor to demonstrate the Proof of Concept (PoC). The major drawback of Process Implanting is that behavior of the implanted process is limited.

Lycosid (Jones et al. 2008) extended the Antfarm approach aimed to detect and identify the hidden processes at the VMM based on the obtained trusted and untrusted views of the guest OS processes. However, the employed cross-view analysis technique (Wang et al. 2005) by the Lycosid depends on manual analysis that is incapable of detecting the disguised processes[1] that appear due to the malware.

The VMwatcher (Jiang et al. 2007) uses the guest view casting technique to externally reconstruct the internal semantics views of the guest OS while functioning

[1]Disguised processes (Shevchenko 2007): These processes may appear as legitimate (eg, `svchost.exe`) by attaching themselves to existing benign processes based on their injected malicious code or by originating from a wrong directory path on the guest OS.

at the VMM. It uses the view comparison-based method to detect elusive malware based on the discrepancy obtained between the internal and external views. However, the results still do not help in detecting particular variants of the stealthy malware, which use code obfuscation technique (Saleh et al. 2014).

Three models such as Out-of-Band delivery, In-Band delivery, and Derivation have been proposed (Pfoh et al. 2009) to bridge the semantic gap. In Out-of-Band delivery model, view extraction function knows the semantic knowledge prior to VMI begins. During view generation, there is no need to run the VMs. The main disadvantage is that it cannot be practically implemented as guest-portable. However, it facilitates the integration of tools like Volatility that is capable to extract data from the acquired memory dumps of the Monitored VM. The view-generating function of the In-Band delivery makes uses of guest OS's knowledge to extract the data internally. The main disadvantage of the In-Band delivery arises from the components that are vulnerable to malware based attacks which have compromised the guest OS. Further, it does not address the semantic gap rather avoids it. In case of derivation, the Monitored VM data is gathered from the hypervisor by knowing hardware architecture semantic knowledge. Some semantic knowledge can be gathered by understanding a specific hardware architecture with monitoring the CPU control registers. So, this approach is guest-portable.

VMI-PL (Westphal et al. 2014) is monitoring language developed to extract guest OS related data. It is an out-of-the-box approach. In terms of functionality, LibVMI is inferior as compared to VMI-PL. Even though VMI-PL offers extensive features, it still faces glitches related to performance. Moreover, it does not deliberate effective methods of creating the triggers by which forensically rigorous evidence cannot be collected. This drawback consequences in the investigation may not be foolproof.

TxIntro is able to offer timely, concurrent and consistent VM introspection by using the support provided by the commodity Hardware Transactional Memory (HTM) (Liu et al. 2014). TxIntro is utterly transparent and it can actively observe critical kernel data structures updates. This is due to the usage of resilient atomicity provided by HTM. TxIntro directly accesses guest VM data by leveraging a method called as core planting that dynamically inserts a stealthy core to the guest VM.

XenAccess (Payne et al. 2007) is a monitoring library used to monitor the memory state as well as disk activities of the Monitored VM. XenAccess makes use of its capabilities such as disk monitoring and virtual memory introspection to introspect safely and efficiently the execution state of the Monitored VM from out of the one being monitored. Generally XenAccess functions on secure VM and it does not require any alteration either to the hypervisor or Monitored VM to view or introspect the execution state of the Monitored VMs running on the same virtualized environment (VMM). XenAccess architecture has been designed by fulfilling the following requirements such as 1) No extra alterations to the VMM or to the Monitored VM or to the guest OS, 2) negligible performance impact, 3) Fast growth of new monitoring programs, 4) capable to observe any data on the guest OS and 6) monitoring code is completely inaccessible to guest OS or Monitored VM. Similarly, patagonix (Litty et al. 2008) ensure the security of the introspected VM at the hypervisor by performing manual analysis of the reconstructed semantic view.

From another perspective, a number of VMM-based in-guest monitor techniques (Gu et al. 2011; Fu et al. 2014) have also been explored to achieve better robustness while leveraging the security advantages from an out-of-VM security approach.

Lare (Payne et al. 2008) pioneered the active monitoring approach by placing a hook into an introspected system using the VMM-protected address space. These hooks trap and analyze the events inside the guest OS and trigger the security application of the trusted VM to take suitable action against the attacks. The main limitation of this approach is its high overhead. In order to intercept particular events whenever occur on the Monitored VM, it places a hook inside the guest OS that invokes the security tool residing in Monitoring VM (trusted VM). The major drawback of Lares is that communication cost is high during transfer of an event from one VM to another via the hypervisor. Thus, Lares is unsuitable for fine-grained active monitoring.

The processes-out-grafting (Srinivasan et al. 2011) inserts the kernel modules into the guest OS to relocate the suspect processes from the introspected VM on to a secure VM, whereby the parallel running VMI tools can access the data structure of the guest OS without an intervention of the untrusted introspected VM. However, the proposed approach is limited to out-graft a single process.

The SYRINGE (Carbone et al. 2012) uses a guest-assisted function-call injection technique to implant a function on to the introspected VM along with a localized shepherding technique to confirm the execution of the invoked guest code on to the guest OS. However, this approach does not ensure the integrity of the data delivered by the infected guest OS that was tampered by kernel-level attacks.

Secure In-VM (SIM) monitoring framework (Sharif et al. 2009) leverages hardware virtualization and memory protection features to offer same security benefits of out-of-the-box monitoring approach. In other words, it utilizes hardware assisted feature to place hook or In-VM monitor in a VMM-protected address space to ensure the security of the Monitored VM while warranting the same level of security at the VMM. It places a security monitor code inside the hypervisor protected virtual address space and this is named as SIM virtual address space. Since SIM is hypervisor protected, it can view the guest OS address space. However, applications running on the monitored VM cannot view security monitor address space. The main limitation of this technique is that it requires manual effort for detection of malicious processes from their reconstructed high-level view.

Malware analysis platform should be transparent and extensible to defeat malware. The transparent platform is essential so that malware cannot straightforwardly identify and evade it. The extensible platform is required to provide resilient support for analysis efficacy and heavyweight instrumentation. Malware analysis platform that provides transparency and extensibility has been proposed by combining hardware virtualization and software emulation (Yan et al. 2012). Hardware virtualization is used to record the malware execution and software emulation (dynamic binary translation) is employed to analyze the obtained records. The objective of combining hardware virtualization with software emulation is to simplify custom fine-grained malware analysis. A prototype implementation of this malware analysis platform is named as V2E and its effectiveness is measured by using 14 real-world emulation resistant malware as well as synthetic samples. However, a major limitation of V2E is that current implementation is restricted only to a single core of guest environment. Lightweight VMM (Nguyen et al. 2009) for malware analysis (MAVMM) eliminates unimportant hypervisor modules to make the VMM as lightweight. It retains only the modules which are necessary to monitor the behavior of malware. A major characteristic of MAVMM are 1) transparent malware analysis system, 2) fixed malware monitoring time and 3) rapidly restore to original status after completion of malware execution. Even though it offers many benefits, it is hard to deploy widely in reality and also it needs additional support from virtual technology.

Usually, shells are designed above an OS kernel. However, hypervisor layer shell (HYPERSHELL) (Fu et al. 2014) has been proposed to demonstrate that shell can also be designed below an OS. HYPERSHELL is a practical hypervisor layer guest OS with centralized management, better automation and uniformity characteristic Further, it possesses all of the functionality of the traditional shell. It automates the guest OS management with no user accounts from the guest OS. Reverse System Call (R-syscall) abstraction has been introduced in HPERSHELL to address the semantic gap problem. Hypervisor programmers can make use of R-syscall abstraction to improve guest OS management utilities without no semantic gap problem. Its transparency feature straightforwardly permits several legacy utilities to be executed in HYPERSHELL without any alteration. A major requirement of the HYPERSHELL is that both guest OS kernel and the `init` process must be trustworthy. However, it cannot be used as an effective security solution to monitor the critical applications, especially when the OS kernel gets compromised. Since HYPERSHELL is designed with a prime intention is to manage the guest OS from the hypervisor in the same means as In-VM, it cannot consider as stealthy introspection approach. Another demand of the HPERSHELL is that both Monitored VM running in the host OS to have compatible syscall interface. Syscall Dispatcher of the HYPERSHELL makes use of dynamic library interposition, and it neglects the syscall policy checking in the dynamic loader. This results in preventing static linked native utility execution in HYPERSHELL.

2.3 MFA Perspective

The possibility of incorporating MFA with VMI offers a number of advantages such as, examination of rich semantic views of critical kernel data structure of all VMs that are manipulated by operating system specific rootkits or malware. The use of MFA at VMM aims to extract and investigate digital artifacts of semantic kernel data structure of the introspected system. More importantly, it significantly reduces the development of introspection program of large kernel data structure by addressing the semantic gap problem of VMI (Dolan-Gavitt et al. 2011). However, The forensic investigation in a cloud environment is a more complex task because cloud forensic still is in the infancy stage. Cloud computing forensics related problems, challenges, issues are systematically examined including open challenges and future directions are summarized in (Zawoad and Hasan 2013). Analysis results suggest that most of the issues need to solve by the CSPs.

Mini VMs named as Forensic Virtual Machines (FVMs) (Shaw et al. 2014) are able to analyze the memory space of other guest VMs. These FVMs are named as lightweight because they consume nominal resources on the Xen virtualization platform and naturally computationally less expensive to run. The main reason for this is that they are in small size. FVMs are small VMs with VMI to discover the symptoms of malware execution occurring on other VMs in real-time. Each FVM is devoted to recognize only one symptom. Further, easy to inspect manually the vital part of the code within the FVM because they are trivial in size. To make the FVMs to communicate with other VM, Xen access control policy has been modified. However, it highlights the necessity of appropriate access control procedures that regulate VM to VM access.

Sophisticate malware or kernel-mode rootkits main purpose is to tamper the kernel data structure or kernel memory data. This questions the credibility of memory analysis tool. Hence, there is a need of memory analysis tool study, particularly, closed-source operating systems such as Window OSs. To address this question, an experiential study conducted thoroughly on memory analysis tools particularly for Windows OS (Prakash et al. 2015) and obtained empirical results demonstrated that both signature-based and traversal-based analysis tools are inaccurate in providing precise evidence even under truthful context. In addition, value equivalence directed field mutation technique has been devised to investigate the manipulation attack space. Experimental results show that these attacks are practically applicable and are able to change much semantic values without being detected and cause adverse consequence on the infected system.

Generally, sophisticated rootkit uses compromise techniques to hide its presence so that the standard forensic measures cannot detect the hidden rootkit. Such kind of hidden rootkits can only be identified by specially designed hardware or software. Rootkit or malware that uses various techniques to alter the x86 architecture at firmware level has been described and proposed methods to detect the firmware-level attacks in the context of memory forensic investigation (Stüttgen et al. 2015). Experiments were conducted on both physical and virtual environments to measure the effectiveness of the proposed approaches with open-source memory forensic tools like Rekall and Volatility that have been used to incorporate the proposed tactics.

Kernel data structure can be reconstructed by analyzing memory dump with the instructions correspond to kernel structure member (Case et al. 2010). The capability to dynamically reconstruct C structures used within the kernel permits for a great quantity of information to be acquired and processed. The tool such as RAMPARSER has this ability and it is able to simulate `ps` and `netstat` commands. In addition, it is also able to reconstruct `task_struct`, `mm_struct`, `File`, `Dentry`, `Qstr`, `inet_sock`, `Sock`, and socket kernel data structures. An forensic investigator can make use of RAMPARSER capabilities to gather significant live forensic evidence from the target system memory. The collected evidences ensure the state and activities of the target system at the time of memory acquisition.

A Stealthy and Practical Execution Monitoring System (SPEMS) (Shi et al. 2015) uses VMI technique to observe the execution state of the Monitored VMs by integrating various open-source tools. SPEMS is able to perform a number of tasks directly onto the target VM that includes memory forensics, malware analysis, out-of-box malware injections and process insertion, VM clone, and program monitoring. Execution tracing is achieved by using LibVMI and Rekall. Limited trapping of user-level functions is the drawback of SPEMS.

Digital evidence plays a vital role by providing great implication for governing cybercrime. However, many digital evidence gathering tools are ineffective in providing accurate digital evidence due to a number of reasons. Hence, the correctness of their acquired evidence cannot be assured. To tackle this problem, a virtualization-based monitoring system for mini intrusive live forensics (VAIL) has been proposed (Zhong et al. 2015). In order to collect target system digital evidence without interrupting the execution of the one being monitored, it utilizes hardware assisted virtualization method. VAIL maintains resistant to attacks from the target system. VAIL has been tested on Windows VM to demonstrate the PoC and is capable to collect various states of the target system such as CPU state, I/O devices activities and the memory content.

There have been efforts to perform live forensic analysis by leveraging virtualization extension on a potentially compromised system without modification or termination of the guest OS state. HyperSleuth framework (Martignoni et al. 2010) uses virtualization technology to enable secure live forensics analysis of the infected systems with three forensics applications operate above the HyperSleuth. Four essential properties of the HyperSleuth ensures that execution environment is trusted: 1) even if an attacker takes the control of the system with kernel-level privilege, analysis results cannot tamper. 2) Rebooting the system is not necessary to install HyperSleuth on the execution system with no volatile data lost. 3) The analysis operation is entirely transparent to both attacker and the OS. 4) regular analysis operation can be conducted and securely interrupted to restart the system so that system can bring back to a normal execution state. However, digital forensics involves extensive manual analysis to acquire empirical evidence of real malicious executables from the infected memory dump.

2.4 Machine Learning Technique Perspective

Over the past decades, several research works have witnessed the use of machine learning techniques to detect unknown malware executables. Schultz et al. (2001) pioneered machine learning for detection of malicious executables by understanding the features of the malware and benign executables. Generally, there are two kinds of methods that can be utilized to detect and classify malware, namely, static malware analysis and dynamic malware analysis.

In static malware analysis, the detection of malware is performed by examining the malicious executables without its execution. The traditional malware detection and classification approaches use static properties of malicious executables that include header details, embedded strings details, packer signature, checksum or MD5 hash, metadata etc. Shafiq et al. (2009) presented a malware detection framework with the name PE-Miner that has the potency to extract individual features from a Portable

Executable (PE) files to spot unknown or zero-day malware in real-time. The proposed approach is based on a three-fold research methodology that consists of identifying structural features of the PE file, reduction of feature by pre-processing and classifying based on the data mining algorithms. The authors experimented with large data samples and achieved detection rate greater than 99% with less than 0.5% FPR.

Hellal and Romdhane (2016) proposed a graph mining algorithm, called, minimal contrast frequent subgraph miner that extracts malicious behavioral patterns from malware executables. It is based on graph mining approach with static malware analysis. The proposed method is evaluated by considering 1083 malware and 1000 benign Windows executables and achieved highest detection rate with low FPR on detection of new variants malware and obfuscated malware.

Ahmadi et al. (2016) proposed a malware classification approach. The extraction of the features was based on the structure and content of the malware samples as multi-features. The authors used an ensemble based XGBoost learning algorithm to validate their classification methodologies and achieved 98.80% detection accuracy on a large malware dataset. A number of classic approaches use Bytecode N-gram (Kolter and Maloof 2006; Masud et al. 2008; Nissim et al. 2012) and Opcode N-gram (Santos et al. 2013; Shabtai et al. 2012; Bai and Wang 2016) based feature extraction techniques to detect and classify malware using the static analysis method. However, the main limitation of this analysis is that it is susceptible to inaccurate detection of the malware that uses strong evasion and obfuscation techniques (Moser et al. 2007). As a consequence, static malware analysis techniques are inadequate to detect unseen malware (Shan and Wang 2014).

To address the limitations of static malware analysis, dynamic or behavioral analysis based techniques are widely used, it detects and classifies the malware by observing the behavior of the malicious executable while it is actually running on the controlled and monitoring environment. It can use API calls (Willems et al. 2007) or system call (Rieck et al. 2011) or any other function-based (Menahem et al. 2009) features to observe the behavior of the executables during their run time on the OS. Therefore, these approaches are well suited for capturing new and unseen malware variants.

Moskovitch et al. (2008) have proposed an approach for the detection of worm activity on a monitored computer system. They managed to extract over 323 computer features using their developed agent on the monitored system. The obtained features were reduced by using four feature selection techniques, while four machine learning classifiers were used to evaluate the results. The evaluation results showed 99% accuracy for specific unknown computer worm activity.

Shahzad et al. (2013) proposed a novel genetic footprint concept by mining information from the process control block of a process. The footprint consists of the semantic behavior of each executing process that can be used to discover malicious processes at run-time. The proposed approach is capable of discovering a malicious process rapidly (less than 100 ms) and achieved an accuracy of 96% with 0% false alarm rate. Similarly, Miao et al. (2016) proposed bilayer abstraction method. It utilizes discriminant and stable behavior features from semantic analysis of dynamic API call sequences that are extracted during execution of samples in a controlled system. Finally, the authors have experimented using improved version of one-class Support Vector Machine (SVM) algorithm to detect a new variant of unknown malware with low FPR.

Rieck et al. (2008) proposed a machine learning-based system for automatic analysis and classification of unknown malware samples based on API function call monitoring from the behavior-based analysis reports generated by the CW sandbox. They used more than 10000 unique malware samples and split them into training, validation, and testing partition to validate their proposed methodologies using the SVM machine learning technique. The proposed system is proficient in classifying individual malware family.

Recently, (David and Netanyahu 2015) presented a DeepSign framework that uses a novel deep-learning technique for automatic malware signature generation and classification. The behavior feature execution of each sample was obtained based on the generated report of the Cuckoo sandbox. The proposed approach was able to achieve 96.4% accuracy on unseen tested malware samples. While the behavioral analysis based methods achieve promising results, they have shortcomings. For example, when a sophisticated stealthy malware, which uses rootkit functionality, is executed on the sandbox or VM, the majority of the execution path will not execute or it's hard to retrieve full execution path and behavior of such malware by the existing dynamic malware analysis approach (Lengyel et al. 2014; Zhang et al. 2016).

Islam et al. (2013) presented classification method that uses static and dynamic features to classify benign and malware samples. The static feature vector was formed by extracting a function length frequency and printable string information from each executable (benign and malware). Similarly, the dynamic features include API function names and parameters execution of each benign and malware. Finally, these three features are combined to construct integrated feature vector. Authors have used four machine learning classifiers namely, SVM, Random Forest, Decision Trees and Instance-Based (IB) classifier to evaluate the classification methodologies and achieved 97.05% of detection accuracy. Recently, (Kumar et al. 2017) also used an integrated feature vector that was comprised of each PE files of header fields raw value and derived values. Authors have used various machine learning classifiers and achieved 98.4% classification accuracy.

Subsequently, there were many excellent recent studies (Zhang et al. 2016; Ahmadi et al. 2016) use a multi-feature of malicious executables to achieve high detection accuracy of malware using efficient ensemble learning techniques on a large scale of malware datasets. Similarly, (Ozsoy et al. 2016) presented a hardware-based online malware detector that used low-level information such as architectural events, instruction, memory addresses, and branches as multi-features that are collected during the execution of the executables. The proposed approach used logistic regression and neural networks for classification purpose and achieved excellent performance towards the detection of malware with little hardware overhead.

In contrast, very few approaches were concentrated to protect CPS while detecting and performing classification of malware from a VMM perspective. In order to provide real-time protection of CPS, recently, (Huda et al. 2017) proposed a semi-supervised approach that protects the CPS against unknown malware attacks. It uses an automatic malware database update strategy that helpful to extract patterns of dynamic changes of malware attacks. The proposed approach has been evaluated against a real-world malware samples of both static and dynamic malware features by using four supervised machine learning classifiers such as Random Forest, SVM, J48, and IB and achieved higher malware detection rate. Recently, one such research effort (Watson et al. 2016) leveraged a one-class SVM technique at the hypervisor for detection of malware on a live VM. The utilization of the features was at both system and network level. In addition, custom Volatility plugin was used to extract features from every resident process that impacted on the generating training dataset. Overall, this approach achieved more than 90% detection accuracy for two types of malware used in the work.

While the behavioral analysis based method achieves promising results, it has shortcomings. For example, when a sophisticated stealthy malware, which uses rootkit functionality, is executed on the sandbox or VM, the majority of execution path will not execute or it is hard to retrieve full execution path and behavior of such malware by the existing dynamic malware analysis approach (Lengyel et al. 2014; Zhang et al. 2016).

However, a few research works have concentrated on the VMM perspective to detect and analyze malicious executables on a live introspected system using machine learning techniques. Recently, one such research effort (Watson et al. 2016) leveraged a one-class SVM technique at the hypervisor for detection of malware on a live VM. The utilization of the features was at both system and network level. In addition, the custom Volatility tool plug-in was used to extract features from every resident process that impacted on the generated training dataset. Overall, this approach achieved more than 90% detection accuracy for two types of malware used in the work.

2.5 Outcome of Literature Survey

1. It is observed in the literature that much of the works concentrated to protect hypervisor against attacks originating from malicious guest OSs. Some work concentrated to mitigate VM to VM attack. However, only a few works focused to protect guest OSs from the untrusted hypervisor. More work is needed in the direction of guest OSs protection from the hypervisor-based security system. This strengthens the security of entire virtualized environment in real-time.

2. The detection of hidden and malicious processes being executed on a VM using the VMI, in a virtualized environment, presents three problems. Firstly, the previous VMI solutions (Jiang et al. 2007) facilitated in reconstructing a semantic view of the processes on the live introspected guest OS. However, this

information was available in dubious forms.[2] For example, the proliferation of the Kelihos (Garnaeva 2012) malware on the guest OS spawned a number of malicious child processes before exiting from the main process. In such an instance, manually distinguishing, detecting, and preventing the running malicious processes from hundreds of benign processes is time-consuming for a security administrator, as it requires a wide knowledge of the malicious executables. To address this issue, machine learning techniques prove to be more promising in the detection of new variants of malware (Kolter and Maloof 2006). However, the adoption of these scientific techniques is difficult for researchers due to lack of benchmarked datasets (benign and malware) in a virtualized cloud environment.

3. The number of running processes on the monitored guest OS may differ significantly from time-to-time, even if there are no hidden processes[3] at the moment of introspection (while checking inside the VM and introspected from the VMM). This is due to the fact that the number of processes available in the system are not constant and change frequently due to the dynamic nature of process creation and destruction. In such cases, it is highly dubious to rely on the introspected data of the VMI for manual detection of the malware.

4. It is difficult to accurately estimate the number of hidden, dead, and dubious processes and to detect the spiteful processes and to identify and analyze the manipulated semantic information in a timely manner on the introspected guest OS.

[2]Dubious Process (DP) is a process that is currently running on a guest OS, and it may or may not be a malicious process (not hidden).

[3]The execution of stealthy malware may hide its presence by executing on the user mode (Ring-3) or kernel mode (Ring-0) of the guest OS, which may conceal other benign processes running on the guest OS.

Chapter 3

Hypervisor based Intrusion Detection and Prevention System

3.1 System Design

The hypervisor is a pillar for virtualization and allows to abstract bare hardware resources to VMs. The sharing of resources increases a greater security risk and challenges for CSP. Once the hypervisor is compromised from an intruder (attacker), the entire virtual environment is under the control of an attacker. In cloud computing environment, intrusion may originate from multiple sources such as VM (Bahram et al. 2010), virtual network (Chung et al. 2013), malicious hypervisor (Azab et al. 2010), etc. The VMs are a prime target for attackers as they are easily available for rent and VMs are directly accessible to the external world through CSP (Wesley Vollmar and Green 2014). An Attacker adopts various tricks to uncover VM vulnerabilities so that exploited vulnerabilities can use as a door to compromise the VM. Once the VM is under the control of an attacker they can inject malware (Rootkit or Spyware or Virus or Worms or Trojans etc.) to disorder the normal operation of the VM. Further, an attacker can trespass the virtualized environment through the compromised VM. Only anti-malware (malware detection software) is scanty to maintain a healthy state of the VM all the time. To protect the entire VM against various types of threats and attacks, IDPS is a prime requirement (Scarfone and Mell 2007). The IDPS should have the capability of malware detection (Rootkit detection, Spyware detection, Virus, Worms, Trojan etc.), log analysis, file integrity checking, incoming traffic analysis, active response etc. (Bray et al. 2008). In addition, it should have detection capacity of unknown attacks as well as known attacks. The only detection is inadequate to

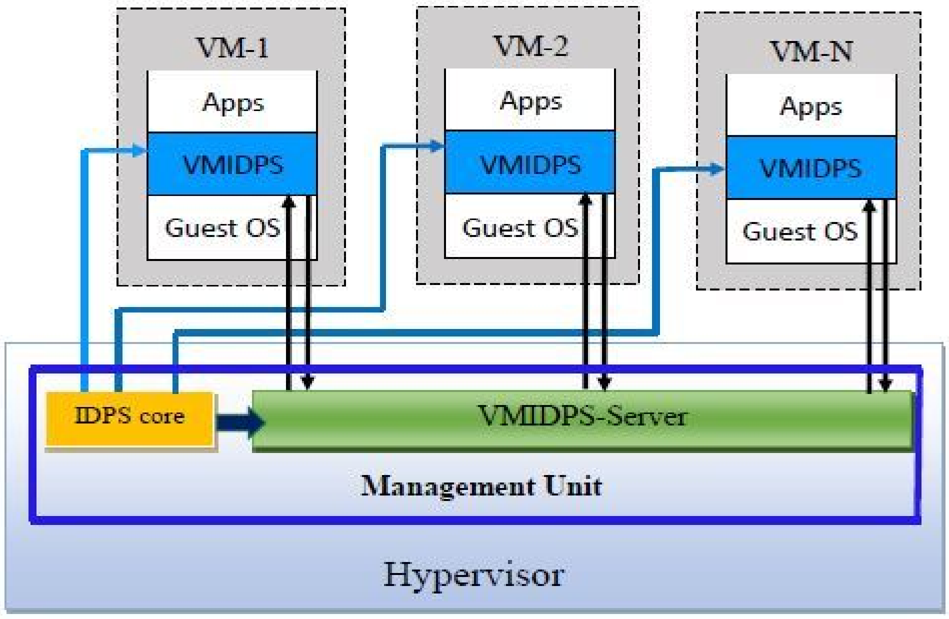


Figure 3.1: Architecture of the VMIDPS for virtualized environment

safeguard the virtual environment without prevention.

In this work, In-and-out-of-the-box Virtual machine and hypervisor dependent intrusion detection and prevention system is proposed for a virtualized environment. The architecture of the VMIDPS is as shown in Figure 3.1. The management unit is one of the significant components of the hypervisor and IDPS core resides on it. The VMIDPS-Server is an integral part of the components of the IDPS core and it runs on the hypervisor. The hypervisor informs the management unit to deploy IDPS-agent onto a new VM whenever the new VM is launched. The IDPS running on the VM is named as Virtual Machine based Intrusion Detection and Prevention System (VMIDPS-agent). The VMIDPS-agent scans the entire VM to certify that VM is in robust as well as the uninfected state. The VM allows for function only if it is certified as robust system else VMIDPS-agent triggers an alert to take an appropriate action to bring back the VM to a normal state.

The VMIDPS-agent on each VM continuously monitors and analyses occurring events to detect and avert malicious events in real-time. Multiple intrusion detection techniques such as file integrity verification, signature based intrusion detection and anomaly based intrusion detection are adopted in VMIDPS-agent to detect various types of intrusion (Rootkits, Virus, Worms, Ports scan, File alteration and so on). The VMIDPS-agent periodically sends the entire state of the VM to VMIDPS-Server

Algorithm 3.1: File Integrity verification algorithm

```
1 Notations used in this algorithm are as follows
      1. $F_i$ ← Integrity of the $i^{th}$ file to be checked
      2. h() ← One-way Hash function
      3. hD' ← Hash digest of file "$F_i$" stored in the hash digest database
         [$h(F_i) \leftarrow hD'$]
      4. hD" ← Newly computed hash digest for the file "$F_i$" by one-way hash function
         h()
      5. N ← Number of files integrity to be checked
  Input: File "$F_i$" integrity to be checked and Hash digest database for the file
         "$F_i$"
2 begin
3     for $i = 1; i \leq N; i++$ do
4         Compute hD" and compare it with hD' (hD"==hD')
5         If  the comparison is true goto 3
6         else
7         declare file "$F_i$" integrity is violated and display both hD" as well as
            hD' present in hash digest database ;
8         Send file "$F_i$" to signature based intrusion detection module to spot the
            malicious content present in it
9     end
10 end
```

to detect the intrusions that are bypassed at VM level. The VMIDPS-Server utilizes cross-view analysis based intrusion detection technique to spot the intrusions.

3.1.1 File Integrity Verification

The VMIDPS periodically monitors operating system files as well as other critical files to ensure their integrity. It is achieved by computing and comparing their cryptographic hash digest with the pre-computed cryptographic hash digest. Comparison mismatch demonstrates file content alteration. The VMIDPS triggers an alert as and when file integrity violation noticed. File integrity verification steps are shown in algorithm 3.1. However, file integrity verification does not provide accurate information regarding the type of intrusion (what content of the file modified). Therefore, files whose integrity is violated are further analyzed to uncover more details of the intrusion.

3.1.2 Signature based Intrusion Detection System

Signature based Intrusion Detection System (SIDS) is also known as misuse detection. It detects intrusion by comparing the observed signature with pre-defined signatures present in the database. The operation of the SIDS is as shown in algorithm 3.2. It detects any kind of known attacks effectively. However, it cannot detect new attacks without the corresponding signature in the signature database. Continuous updation of the signature database is crucial to detect new attacks. This limitation is overcome by Anomaly based Intrusion Detection System.

Algorithm 3.2: Signature Based Intrusion Detection

```
1 Notations used in this algorithm are as follows
      1. $F_i \rightarrow$ File to be checked
      2. $SD \rightarrow$ Signature Database of $F_i$
      3. $M \rightarrow$ Number of files to be checked
      4. $FSD_i \rightarrow$ Signature of the file "$F_i$"
  Input: File "$F_i$" to be checked and Signature database "$SD$"
2 begin
3     for $i = 1; i \leq M; i++$ do
4         Extract $FSD_i$ for the File $F_i$ and compare it with $SD$.
5         if the comparison is match with one or more entries of the $SD$ then
            declare the corresponding attack.
6     end
7 end
```

3.1.3 Anomaly based Intrusion Detection System

It detects intrusion by comparing observed activities with baseline profile without a signature database. $'Th_i'$ is a threshold for an activity "i" where $i = 1, 2, 3, ..., N$. Profile of the system $PS = \{Th_1, Th_2, ..., Th_N\}$. AS_i is an anomaly score of an activity $'i'$ observed for the period of time $'T'$ is

$AS_i = \sum_{i=1}^{N}, ac_i \quad i = 1, 2, 3, ..., N$. Where "$ac_i$" is i^{th} observed activity score at time 'T'.

$$f(AS_i, Th_i) = \begin{cases} Anomalous, & As_i > Th_i \\ Normal, & As_i \leq Th_i \end{cases} \tag{3.1}$$

Table 3.1: Experimental testbed using OSSEC HIDS

Hypervisor	Oracle Virtual Box 4.3.16
HIDPS	OSSEC 2.8.1
VMs	Ubuntu 10.04 and Windows-7
OSSEC 2.8.1	(server, local, agent)
OSSEC2.8.1-sever	out-of-the-box IDPS
OSSEC2.8.1-agent	In-the-box IDPS (Ubuntu10.04)
OSSEC2.8.1-agent	In-the-box IDPS (Windows-7)

Where $'f'$ is a comparison function. The outcome of the comparison is anomalous when observed activities exceed the threshold otherwise legitimate. The major advantage of Anomaly based Intrusion Detection System is that it can detect new attacks including zero day attack.

3.1.4 Cross-View Analysis

Each VM periodically sends its entire state information to VMIDPS-Server. Once the VMIDPS-Server receives the VM information then it requests the hypervisor to supply actual low-level information of that particular VM. As hypervisor has complete control on every VM, it supplies requested VM information to VMIDPS-Server. By comparing information supplied by the hypervisor with the information received from the VM, it identifies the intrusion if the comparison result is deviated.

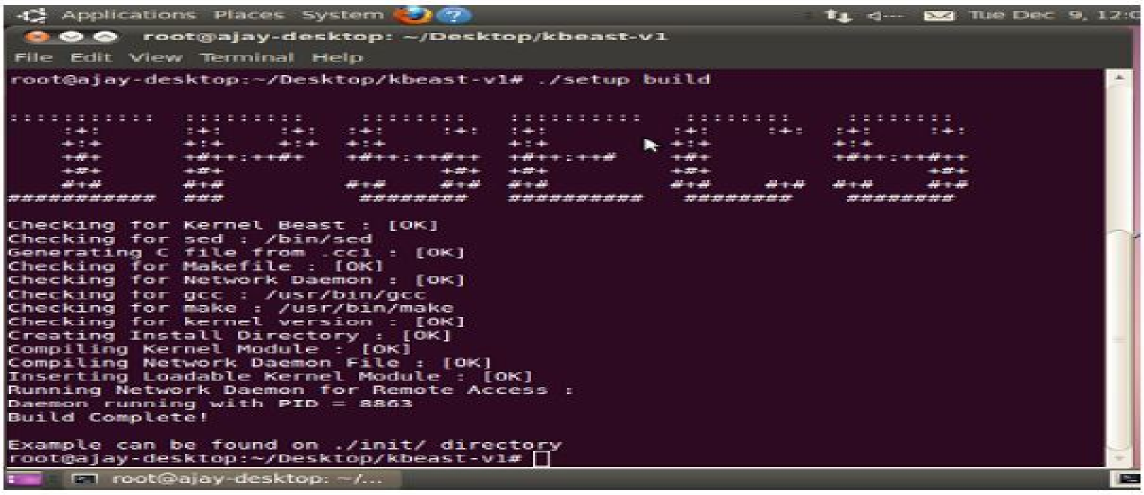
3.2 Experimental Setup and Results

In this work, Oracle Virtual Box (GmbH 2007) was used to setup the virtualized environment as shown in Table 3.1. The VMs were used to conduct the experiments. The HIDS such as OSSEC was used as VM based IDS in this experimental work. The OSSEC supports server, agent, local and hybrid type of installation. A local type of OSSEC installation was adopted to simulate In-the-box detection approach. In this method, each VM has its own dedicated IDS which monitors the entire VM events and triggers an alert as per the conditions set in the configuration file. The server type of installation was followed to simulate out-of-the-box intrusion detection technique. In this approach, the OSSEC-agent is needed on each VM. The OSSEC-agent captures the events occurring on the VM and sends them to the OSSEC-Server in order to ascertain the malicious event trace. The OSSEC-Server and OSSEC-agent communicate securely using the key shared between them. Prevention is achieved by removing the detected intrusions. In this experimental work, both Linux and Windows based rootkits were demonstrated. In addition, DoS attack, and File Integrity verification tests were conducted on Linux and Windows guest OS using Oracle Virtual Box hypervisor version 4.3.16.

3.2.1 Linux rootkits

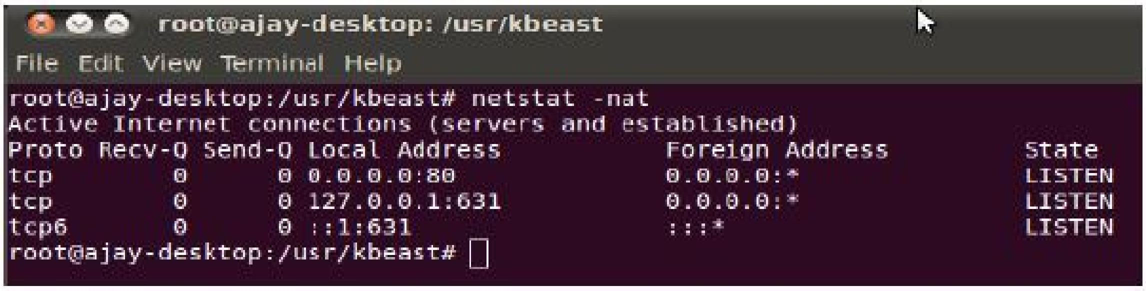
- **Kernal Beast (KBeast) Rootkit** (Team 1998): The KBeast is a kernel-mode rootkit, it has the ability to hide LKMs, Process, Ports, Socket, and connections (`netstat, lsof`) files and directory. Anti-kill process, Anti-remove files, Anti-delete loadable kernel modules, Local root escalation backdoor and remote binding backdoor hidden by the kernel-mode rootkit are also features of KBeast rootkit. In addition, it has the capacity to launch password protected backdoor. As a proof of experiment Figure 3.2a, the KBeast rootkit injected onto Ubuntu 10.04 32-bit operating system VM. In order to hide the port number that needs to be hidden, passwords for backdoor, prefix of the files or directories that are to be hidden are required to set in the configuration file prior to compilation of the rootkit (as shown in Figure 3.2b). Finally, our proposed approach successfully detect KBeast rootkit by generating alert messages as shown in Figure 3.2c.

- **Xingyiquan** is a simple Linux OS based kernel-mode rootkit. The major stealthy functions of this rootkit are escalate root privilege of legitimate, files/directories hide, connections hide, module hide, hook kill the process, hook open, etc. The Xingyiquan rootkit successfully injected onto Ubuntu 12.04 LTS VM where OSSEC-agent was running. The OSSEC-Server was running on another Ubuntu 12.04 LTS VM. The OSSEC-Server effectively detected the malicious activity based on the information reported by OSSEC-agent and triggered the alert message after the detection.



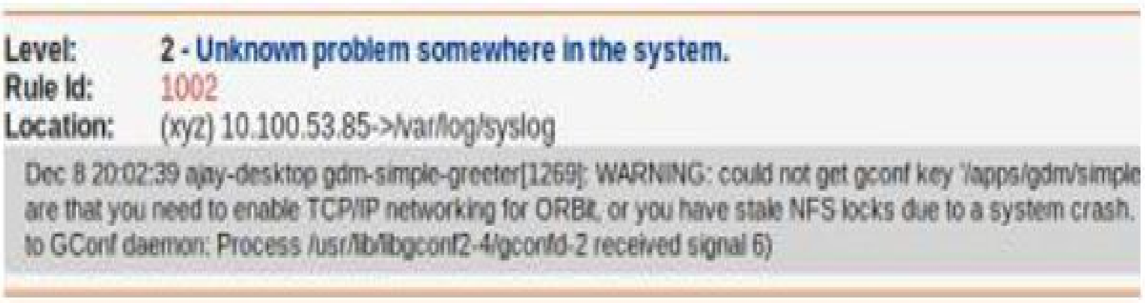
```
root@ajay-desktop:~/Desktop/kbeast-v1# ./setup build

:::::::::::  :::::::::    ::::::::   ::::::::::   ::::::::   ::::::::
    :+:      :+:    :+:  :+:    :+:  :+:         :+:    :+: :+:    :+:
    +:+      +:+    +:+  +:+         +:+         +:+        +:+
    +#+      +#++:++#+   +#++:++#++  +#++:++#    +#+        +#++:++#++
    +#+      +#+                +#+  +#+         +#+               +#+
    #+#      #+#         #+#    #+#  #+#         #+#    #+# #+#    #+#
###########  ###          ########   ##########   ########   ########

Checking for Kernel Beast : [OK]
Checking for sed : /bin/sed
Generating C file from .cc1 : [OK]
Checking for Makefile : [OK]
Checking for Network Daemon : [OK]
Checking for gcc : /usr/bin/gcc
Checking for make : /usr/bin/make
Checking for kernel version : [OK]
Creating Install Directory : [OK]
Compiling Kernel Module : [OK]
Compiling Network Daemon File : [OK]
Inserting Loadable Kernel Module : [OK]
Running Network Daemon for Remote Access :
Daemon running with PID = 8863
Build Complete!

Example can be found on ./init/ directory
root@ajay-desktop:~/Desktop/kbeast-v1#
```

(a)



```
root@ajay-desktop:/usr/kbeast# netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp6       0      0 ::1:631                 :::*                    LISTEN
root@ajay-desktop:/usr/kbeast#
```

(b)



Level: 2 - Unknown problem somewhere in the system.
Rule Id: 1002
Location: (xyz) 10.100.53.85->/var/log/syslog

```
Dec 8 20:02:39 ajay-desktop gdm-simple-greeter[1269]: WARNING: could not get gconf key '/apps/gdm/simple
are that you need to enable TCP/IP networking for ORBit, or you have stale NFS locks due to a system crash.
to GConf daemon: Process /usr/lib/libgconf2-4/gconfd-2 received signal 6)
```

(c)

Figure 3.2: KBeast rootkit file compilation (a), rootkit hides port details (b), OSSEC alert message for rootkit injection(c)

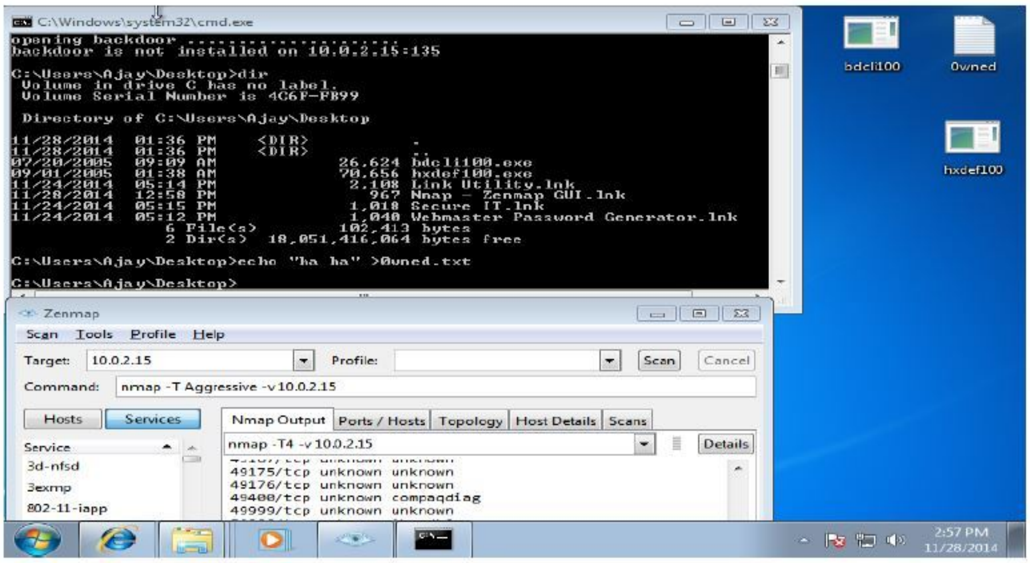
- **FK-0.4 Rootkit** (Team 1998): This rootkit injected onto Ubuntu 12.04 LTS VM. It has the capacity to hide listening port, remote ports, processes and backdoor password changing. Configuration file"/dev/proc/fuckit/conFig/rkconf" allows to set the ports to be hidden, the process to be hidden, and password for the backdoor. The file "lport" under "/dev/proc/fuckit/conFig/rkconf" is use to set listening port to be hidden. Similarly other files such as "rport" and "progs" are used to set the remote port to be hidden and process to hidden respectively. In addition, the file "password" is used to set the password for the backdoor. The rootcheck component of the OSSEC successfully detected the injected rootkit and the corresponding alert messages are recorded onto log file as possible hidden files including complete path of the file where the observation made.

- **Average Coder** (Team 1998): It is a Linux based kernel-mode rootkit. It uses the LKM to inject the malicious code into the kernel space during the run time of the OS. It has the capability to hide itself from `lsmod` and also hides the process, TCP connections, ports and logged-in users details. We have tested this rootkit by injecting onto a Linux distribution of Ubuntu 12.04 VM.

- **Jynx2** (Team 1998): It is a Linux based user-mode rootkit. It dynamically inserts malicious library function into system binaries during runtime. It does not replace the original binary files of the system to evade the detection from digest based detection method. Once Jynx2 rootkit program is executed, the system checks for shared libraries that are needed to be loaded at run time by referring /etc/ld.so.conf and /etc/ld.so.preload files. The `LD_PRELOAD` is an environment variable may be used to point to a shared library (added by Jynx2 rootkit). Using this feature, the Jynx2 rootkit creates a malicious shared library to hide the files, processes, and ports. Utility like `ps`, `netstat` and `ls` loads malicious shared library function onto running process. As a result, files, processes, network connections and ports used by malware are hidden. The malignant user or an attacker can view the processes, files and network connections details through a shared library created by the rootkit. Experiment for this rootkit successfully conducted on Ubuntu 12.04 LTS VM.

3.2.2 Windows Rootkit

- `Hacker Defender`: It is a persistent user-mode rootkit for Windows system. It modifies the native Application Program Interface (API) function of Windows system and its major goal is to allow a hacker to hide the process, files, registry keys, and system driver. Moreover, it checks open port of the

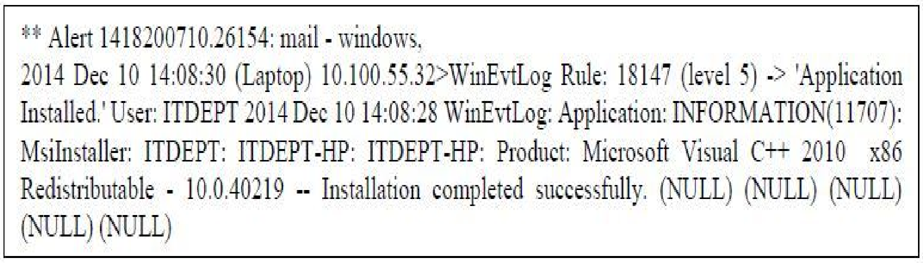


```
opening backdoor ......................
backdoor is not installed on 10.0.2.15:135

C:\Users\Ajay\Desktop>dir
 Volume in drive C has no label.
 Volume Serial Number is 4C6F-FB99

 Directory of C:\Users\Ajay\Desktop

11/28/2014  01:36 PM    <DIR>          .
11/28/2014  01:36 PM    <DIR>          ..
07/20/2005  09:09 AM            26,624 bdcli100.exe
09/01/2005  01:38 AM            70,656 hxdef100.exe
11/24/2014  05:14 PM             2,108 Link Utility.lnk
11/28/2014  12:58 PM               967 Nmap - Zenmap GUI.lnk
11/24/2014  05:15 PM             1,018 Secure IT.lnk
11/24/2014  05:12 PM             1,040 Webmaster Password Generator.lnk
               6 File(s)        102,413 bytes
               2 Dir(s)  18,051,416,064 bytes free

C:\Users\Ajay\Desktop>echo "ha ha" >0wned.txt

C:\Users\Ajay\Desktop>
```

Target: 10.0.2.15
Command: nmap -T Aggressive -v 10.0.2.15

```
nmap -T4 -v 10.0.2.15
49175/tcp unknown unknown
49176/tcp unknown unknown
49400/tcp unknown compaqdiag
49999/tcp unknown unknown
```

(a)



** Alert 1418200710.26154: mail - windows,
2014 Dec 10 14:08:30 (Laptop) 10.100.55.32>WinEvtLog Rule: 18147 (level 5) -> 'Application Installed.' User: ITDEPT 2014 Dec 10 14:08:28 WinEvtLog: Application: INFORMATION(11707): MsiInstaller: ITDEPT: ITDEPT-HP: ITDEPT-HP: Product: Microsoft Visual C++ 2010 x86 Redistributable - 10.0.40219 -- Installation completed successfully. (NULL) (NULL) (NULL) (NULL) (NULL)

(b)

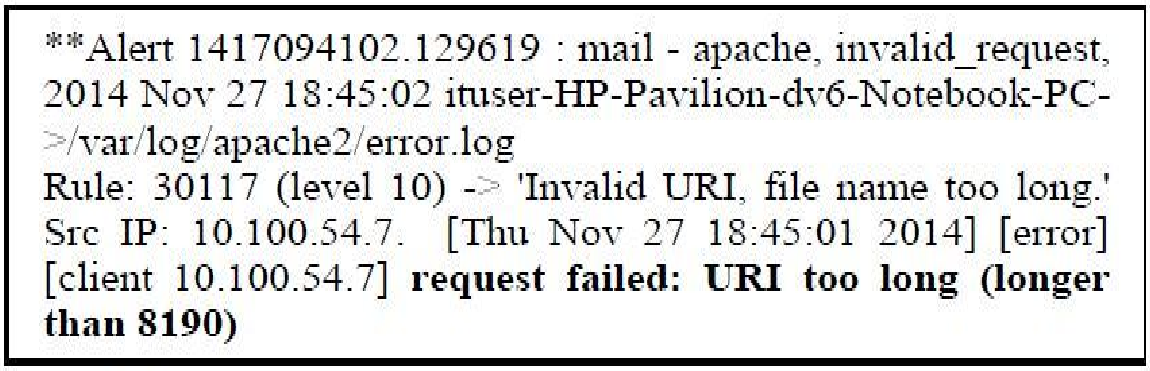
Figure 3.3: `Hacker Defender` rootkit injection (a) OSSEC alert message for `Hacker Defender` rootkit (b)

network connections. This rootkit consists of two files one is executable file `hxdef100.exe` and another one is configuration file `hxdef100.ini` which is used to set the attributes of the rootkit. This rootkit successfully injected onto Windows-7 operating system based VM. The injected executable file creates a new process that hides the process. Screenshot illustrated in Figure 3.3a provides the successful execution of `Hacker Defender` rootkit on Windows-7 VM. Alert message generated by the OSSEC after the rootkit detection is as shown in Figure 3.3b.

- `FU Rootkit` (Butler 2005): It is a kernel-mode rootkit and it is based on Direct Kernel Object Manipulation (DKOM) technique. This rootkit is available from public malware repository. The `FU-rootkit` allows an intruder to hide information of the user as well as a kernel module and it is also able to modify the kernel data structure. `FU-rootkit` successfully tested on Windows-7 32 bit OS based VM where OSSEC-agent was running. The OSSEC-Server detected the injected rootkit effect and notified the observed changes by recording an alert message onto the log file.

3.2.3 DoS attack

DoS attack intention is to make the computer or network resource unavailable to legitimate users by flooding enormous bogus messages. Detecting and preventing DoS attack is massive challenges for CSP. The attackers may use sophisticated powerful DoS attack tool to flood massive number fake requests onto victim machine/server so that service disruption arises to the legitimate one. LOIC[1] (Low Orbit Ion Cannon) is one such open source tool to launch a massive number of UDP, TCP and HTTP attack towards target machine. DoS attack successfully initiated from Windows-7 OS machine as a sending host towards target machine using LOIC. The OSSEC-Server successfully detected and alerted the identified event by recording onto the log file. Figure 3.4 shows the notice triggered by the OSSEC.



```
**Alert 1417094102.129619 : mail - apache, invalid_request,
2014 Nov 27 18:45:02 ituser-HP-Pavilion-dv6-Notebook-PC-
>/var/log/apache2/error.log
Rule: 30117 (level 10) -> 'Invalid URI, file name too long.'
Src IP: 10.100.54.7. [Thu Nov 27 18:45:01 2014] [error]
[client 10.100.54.7] request failed: URI too long (longer
than 8190)
```

Figure 3.4: OSSEC detected DoS attack as URI too long

3.2.4 Port Scanning Attack

Port scanning is the process of identifying exploitable ports or listening ports of the victim system. In this technique, an attacker tries to gather some specific information from the potentially exploitable target port of the targeted machine. Using this

[1]https://sourceforge.net/projects/loic/.

method an attacker creates a list of potential weaknesses and vulnerabilities in the open port leading to the exploitation and compromise the machine.

Similarly, an attacker identifies the opened ports of the VM so that he/she can compromise the VM through the identified open ports vulnerabilities. In a virtualized environment, researchers enlisted different types of known method to detect and prevent port scan attack. We have used Nmap[2] 6.47 tool as the launching pad for port scanning (intense scan) towards target machine. Port scan successfully detected by the OSSEC. Table 3.2 depicts Linux and Windows rootkits and the different stealthy malicious functionality tested in this experimental work and Table 3.3 shows the few different attack scenario conducted in the empirical tests. The proposed In-and-Out-of-the-Box VMIDPS is able to detect and prevent intrusions (with admin intervention) using an open source OSSEC tool.

3.3 Discussion

The proposed hypervisor based Virtual Machine based Intrusion Detection and Prevention System (VMIDS) assumes that the hypervisor (Oracle Virtual Box) is trustworthy and is capable to send low level information such as CPU, Memory and Disk information to the OSSEC-Server. The second assumption is that management unit of the hypervisor automatically deploys OSSEC IDS onto new VMs as and when they launched and OSSEC-Server runs on the hypervisor.

3.4 Limitation of the In-and-Out-of-the-Box Virtual Machine Based IDPS

The proposed in-and-out-of-the-box-hypervisor and Virtual Machine dependent Intrusion Detection and Prevention System is capable to detect stealthy rootkits and different types of the intrusions. For example, DoS attack, and port scanning attack etc. These kinds of intrusions are potentially harmful to virtualization environment which are originating via VM.

[2]https://nmap.org

Table 3.2: List of Windows and Linux rootkits used in this experiments and detected by our proposed approach

Rootkit Name	Virtual box 4.3.16	Monitored VM		Agent running	user-mode	kernel-mode	Hides Process	File Hide	Hides Port	Privilege Escalation	Detected
		Windows	Linux								
`KBeast`	√	×	√	√	×	√	√	√	√	√	√
`FK Rootkit`	√	×	√	√	×	√	√	×	×	√	√
`Xingyiquan`	√	×	√	√	×	√	√	√	√	√	√
`Average coder`	√	×	√	√	×	√	√	×	×	√	√
`Jynx2 Rootkit`	√	×	√	√	√	×	√	√	×	√	√
`Hacker Defender`	√	√	×	√	×	√	√	√	√	√	√
`Fu-Rootkit`	√	√	×	√	×	√	√	×	×	√	√

Table 3.3: Security attack scenarios experimented and detected by our proposed approach

Security Attack	Detection results by the proposed approach
DoS Attack	Massive number of bogus request reported by proposed approach based on the rule defined in URL tag
Port Scanning attack	OSSEC detected the port scanning attack by reporting the used tool (ex: Nmap)

The architecture of the VMIDPS leverages the Rule-based technique to detect the intrusions by performing file integrity verification and we believe that the proposed approach is capable to detect both known and unknown attack successfully. Further, it ensures the healthy state of the VM by detecting and eradicating the intrusions in real-time at VM level only. We have injected and demonstrated 7 both Linux and Windows real-world rootkits. The proposed VMIDPS with our modified rules applied in the OSSEC tools is capable to detect real-world rootkits and other attacks with active response or notification.

The main limitation of the in-the-box Intrusion Detection or agent based Intrusion Detection solutions is not much efficient to detect unknown malware or APT in the virtualized environment. The evolution of the robust malware (e.g. Agobot variant rootkit) (Jiang et al. 2007) much more sophisticated to bypass even to evade agent based solution in order to avoid this types of rootkit.

3.5 Summary of the Work

In this paper, In-and-out-of-the-box Virtual Machine and hypervisor dependent Intrusion Detection and Prevention System is proposed. The main goal of this work is to detect real-time intrusions including stealthy self-hiding rootkits and DoS attack. These kinds of intrusions are potentially harmful to virtualization environment those are originating via the VM. The architecture of the VMIDPS leverage cross-view based technique for real-time identification of the intrusions. File alteration identification is achieved by integrity checking algorithm. The proposed approach ensures the healthy state of the VM by detecting and eradicating the intrusions in real-time. The OSSEC IDS effectively utilized as VM based IDS to identify the abnormal activities of the VM. Experimental results show the detection capability of the VMIDPS. The same work continues as a future work with VMI technique as out-of-the-box intrusion detection technique. A number of experiments are planned to conduct for different attack scenarios (hypervisor and VM based) to measure the efficiency of the VMI based detection approach. Different malicious software such as rootkits, malware, spyware, and worms are planned to test in future work to verify the detection as well as prevention capability of our proposed approach.

Chapter 4

VMI-based Stealthy Malware and Rootkit Detection System

4.1 Introduction

The VMI is able to gather the run-state information of the Monitored VM without the consent or knowledge of the one being monitored while functioning at the hypervisor or the VMM. However, obtaining meaningful guest OS state information such as process list, module list, system calls, network connections, etc., from the viewable raw bytes of the guest OS memory is a challenging task for the VMI and referred to as the semantic gap (Dolan-Gavitt et al. 2011; Jain et al. 2014). To tackle this problem, several approaches have evolved over the last few years by considering different constraints of the guest OS (Fu and Lin 2013; Saberi et al. 2014). However, the current challenges of VMI are: 1) It must have higher scalability features to introspect the rich semantic views of the live state of the guest OS. To achieve this, it requires tremendous manual effort to build kernel data structure knowledge of large volumes of guest OS (Bauman et al. 2015b), 2) The VMI solution requires frequent rewriting of the introspection program due to the dynamic and frequent upgrading of the kernel version, and 3) The VMI must be built with a robust introspection technique that would help to reduce the performance overhead and make the VMI automated with little human effort.

On the other hand, many modern families of malware leverage stealthy rootkits functionality to conceal itself, and to evade detection system to tamper other critical kernel data structure such as files, directories, sockets, etc., of the guest OS (Goudey 2012a; Xuan et al. 2009). The best way of detecting it is by identifying the hidden running processes. The process details are a key source of information for any introspection program to spot the existence of malware. A prior attempt, the VMM-based Lycosid (Jones et al. 2006) is aimed at detecting and identifying only the Hidden Processes (HP) of the Monitored VM at the VMM. However, the current generation of evasive malware may create new malicious processes (not hidden) or attach itself to the existing legitimate running processes. In such cases, Lycosid is inadequate to detect such a malicious malware process. It does not identify the name or binary of the process and it is also inefficient in checking the detected hidden details are malicious or benign. Moreover, the process visible from the hypervisor may be noisy, most likely incorrect, and may lead to a false positive. Another approach, named the Linebacker (Richer et al. 2015), also uses the cross view analysis to investigate the rootkit running on the guest OS. The efficiency of the Linebacker has been demonstrated on the VMware vSphere-based guest OS. However, only the Windows guest OSs were considered for evaluation.

The significant challenges in detecting the malicious and dead processes, particularly in a virtualized environment are:

- The number of running processes on the monitored guest OS may differ significantly from time-to-time, even if there are no hidden processes[1] at the moment of introspection (while checking inside the VM and introspected from the VMM). This is due to the fact that the number of processes available in the system are not constant and change frequently due to the dynamic nature of process creation and destruction. In such cases, it is highly dubious to rely on the introspected data of the VMI for manual detection of malware.

- Accurately estimate the number of hidden, dead, and dubious processes and detect the spiteful processes in a timely manner on the introspected guest OS is challenging task.

To address above challenges, the VMI-based A-IntExt system for a virtualized environment is presented. It mainly detects the hidden, dead and malicious processes

[1]The execution of stealthy malware may hide its presence by executing on the user mode (Ring-3) or kernel mode (Ring-0) of the guest OS (Florio 2005), which may conceal other benign processes running on the guest OS.

that are invoked by the rootkits or malware by leveraging ICVA for process algorithm between the externally (VMM-level) captured run-state information and the internally (In-VM level) acquired execution-state information of the Monitored VM. The pertinent contributions of the present work are as follows:

- We have designed, implemented, and evaluated a consistent, real-time VMM-based guest-assisted A-IntExt system that periodically examines the state of the live introspected system to detect the running of malicious processes from the forensically reconstructed executables.
- We have implemented a mathematical model of the ICVA algorithm as PoC and implanted it into the A-IntExt system to intelligently cross-examine the internally (VM-level) and externally (VMM-level) gathered state information to detect hidden, dead, and dubious process, and also to predict early symptoms of malware execution using the novel TIT technique.
- The robustness of the A-IntExt system was evaluated using publicly available Windows rootkits. In addition, malware were also employed in the experimental work to make the evaluation comprehensive. The A-IntExt system correctly detected all of the hidden, dubious and dead processes.

4.2 Assumption and Threat Model

In this work, we first assume that both the VMM and the Monitoring VM are to be trusted and are operating under a trusted computing base (Intel 2016) that sufficiently enforces physical security control, while resisting hardware-based attacks on the virtualized cloud infrastructure (Wojtczuk and Rutkowska 2009). Secondly, the malware leverages guest OS vulnerabilities that cannot jeopardize the security of the A-IntExt system operating in the privileged domain (Dom0) of the Xen hypervisor. These assumptions are consistently shared by most of the VMM-based previous security research work (Garfinkel et al. 2003; Payne et al. 2008; Srinivasan et al. 2011; Jiang et al. 2007). Thirdly, the established communication channel between the State Information Requester (SI-Requester) of the A-IntExt system and the Guest Assisted Module (GAM) is secure during the lifetime of the live introspected VM. It cannot be modified by any kind of attack or security threat. Some previous research work (Wojtczuk 2008) have highlighted the successful compromising of the VMM and Dom0, but that is beyond the scope of this work.

4.3 Overview

The A-IntExt system is a VMM-based guest-assisted introspection system that advances the current out-of-VM approach in an automated, isolated, real-time manner, while functioning at the secure Monitoring Virtual Machine (Monitoring VM) or Dom0. Figure 4.1 shows an overview of the A-IntExt system. It is introduced to efficiently investigate and detect any hidden, dead, and dubious processes, while predicting early infection of malware symptoms by internally gathering and externally introspecting the volatile memory of the live untrusted Monitored Virtual Machine (Monitored VM). The A-IntExt system achieves this goal by using the ICVA, which is an integral component. The major components of the A-IntExt system are the GVM-Introspector, GAM, ICVA, OMS.

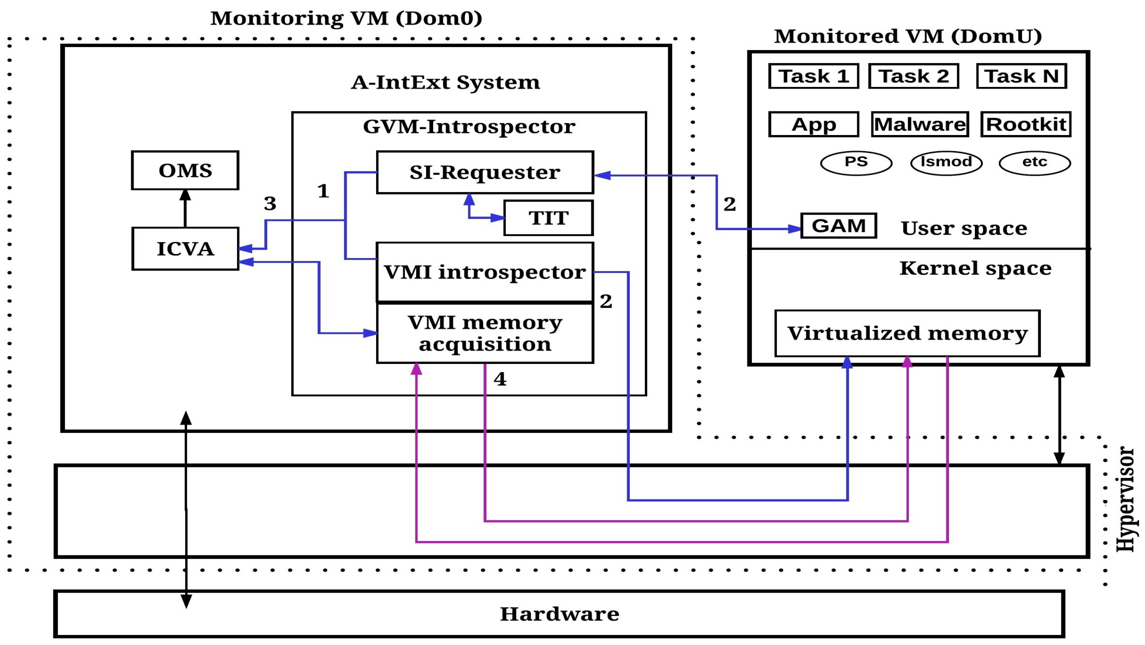


Figure 4.1: The proposed VMI based A-IntExt system

4.3.1 GVM-Introspector

The prime function of the GVM-Introspector is to introspect and extract the running processes information of the Monitored VM. Its sub-components are: SI-Requester, TIT, VMI introspector, and VMI memory acquisition. To start with, the GVM-Introspector initiates the procedure of investigation by signaling to both the SI-Requester and VMI introspector (step 1) to introspect and acquire the current execution of the process state information of the Monitored VM. The SI-Requester (step 2) triggers the GAM via a secure communication channel[2] to internally enumerate the executing process state of the Monitored VM. Upon receiving the internally acquired process state information, the SI-Requester verifies whether the reply arrived within TIT. The time interval between the state information request sent and the reply received by the SI-Requester from the GAM is known as TIT. If the time gap between the state information request and the reply lies within the TIT, it continues the operation. At the same time, the VMI introspector (step 2) introspects and reconstructs the memory state of the Monitored VM from the hypervisor (externally) to get currently running process details. The procedure followed by the VMI introspector to reconstruct and obtain the semantic view of the processes (including hidden processes) by traversing the _EPROCESS Windows kernel data structure of the introspected VMs (discussed below). In addition, the TIT, as shown in Figure 4.2, efficiently addresses the time synchronization problem, which impacts on the detection of the malicious processes under the dynamic creation and destroy of processes as discussed in the second challenge of Section 4.1.

For example: Let T_1 be the date and time at which the state information request is sent to the Monitored VM, and T_2 be the date and time at which the reply is received by the SI-Requester from the GAM. After receiving the state information, the SI-Requester checks the time interval between T_2 and T_1; T_2 - T_1 > ΔT (where ΔT denotes the predefined threshold time that ranges between 2 to 3 sec). Then, the SI-Requester instantly resends the request (maximum 3 times), if the response is *delayed*

[2]A secure communication channel is established between the SI-Requester and the GAM by the A-IntExt system as soon as the Monitored VM is launched by the VMM. The SI-Requester holds native information (IP address and Port number) of the Monitored VM and it triggers the GAM in the form of state information request to receive internally acquired process information of the Monitored VM.

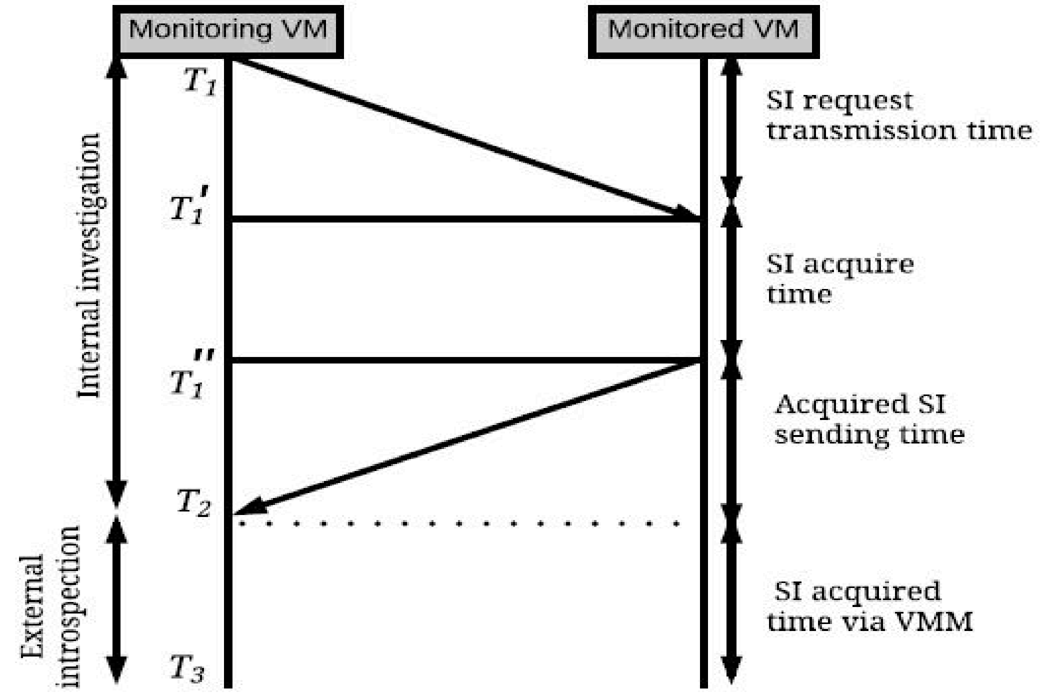
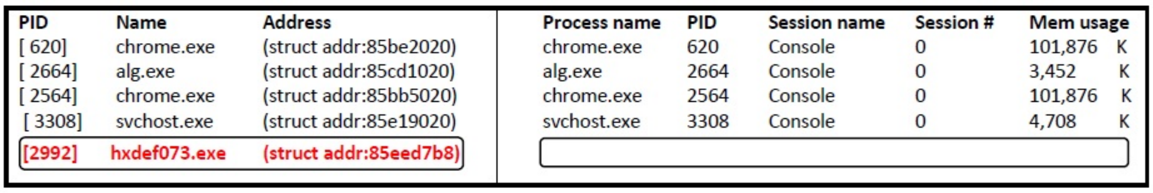


Figure 4.2: Time interval threshold used by A-IntExt system

or not received within the predefined ΔT. Then, the A-IntExt system confirms that the Monitored VM is in an infected state and immediately informs the VMI memory acquisition (step 3) to pause and perform memory acquisition (step 4), and then, to resume running the Monitored VM.

If T_2 - T_1 $\leq$ ΔT then, both the SI-Requester and VMI introspector continue periodic introspections of the Monitored VM by extracting and relocating the process run-state information to the ICVA (Step 2 and Step 3). Further, the ICVA (will be shortly introduced) cross-examines the acquired process state information. This process continues throughout the lifetime of the live Monitored VM. The novel time synchronization technique, the TIT helps the ICVA to detect hidden and malicious state information of the Monitored VM. For instance, the processes $P_1, P_2, P_3...., P_N$ currently being run at the Monitored VM and their details are extracted internally between the time intervals T_1 and T_1''. If any process expires or dies after T_1'' and before T_2, the process details will not appear in the state information caught externally by the VMI introspector, but can be found in the internally captured state information, such processes are treated as `dead processes`.

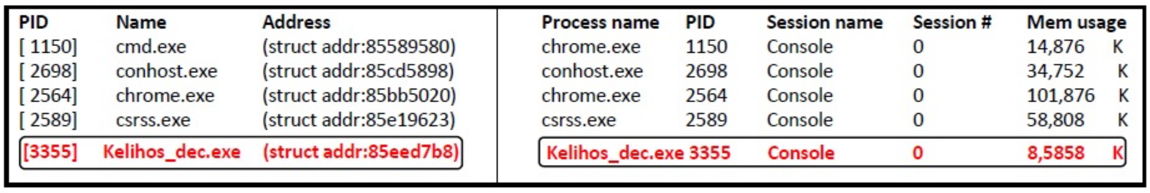
In contrast, if a new process, P_{N+1} is created between T_1'' and T_3, then the process details will appear only in the externally captured state information and not in the



PID	Name	Address
[620]	chrome.exe	(struct addr:85be2020)
[2664]	alg.exe	(struct addr:85cd1020)
[2564]	chrome.exe	(struct addr:85bb5020)
[3308]	svchost.exe	(struct addr:85e19020)
[2992]	hxdef073.exe	(struct addr:85eed7b8)

Process name	PID	Session name	Session #	Mem usage
chrome.exe	620	Console	0	101,876 K
alg.exe	2664	Console	0	3,452 K
chrome.exe	2564	Console	0	101,876 K
svchost.exe	3308	Console	0	4,708 K

(a)



PID	Name	Address
[1150]	cmd.exe	(struct addr:85589580)
[2698]	conhost.exe	(struct addr:85cd5898)
[2564]	chrome.exe	(struct addr:85bb5020)
[2589]	csrss.exe	(struct addr:85e19623)
[3355]	Kelihos_dec.exe	(struct addr:85eed7b8)

Process name	PID	Session name	Session #	Mem usage
chrome.exe	1150	Console	0	14,876 K
conhost.exe	2698	Console	0	34,752 K
chrome.exe	2564	Console	0	101,876 K
csrss.exe	2589	Console	0	58,808 K
Kelihos_dec.exe	3355	Console	0	8,5858 K

(b)

Figure 4.3: Hidden processes (a) dubious processes (b) details of Monitored VM externally introspected (left side) and internally acquired (right side) by the A-IntExt system after rootkit and stealthy malware injection on Windows guest OS

internally captured state information. As a result, process P_{N+1} is recognized as a `hidden process`, even though process P_{N+1} is not concealed. Thus, a disparity emerges between the internally and externally captured state information. Further, to check the maliciousness of the new process P_{N+1}, the ICVA immediately signals to the VMI memory acquisition (step 4) to pause and accomplish memory acquisition of the Monitored VM. The executable file extractor (shown in Figure 4.5) extracts the entire executable file of the corresponding process from the seized memory dump of the Monitored VM.

Figure 4.3 shows the processes details of the Monitored VM captured internally and externally after malware and rootkit injection, it includes hidden, dubious processes information. This is achieved by the well-built isolation property of the hypervisor which guarantees that the state information captured from the A-IntExt system is accurate.

Memory state reconstruction: Since the VMM view of the Monitored VM is available in raw memory state, the A-IntExt system performs memory state reconstruction before it collects and analyzes the process run state information of the Monitored VM at the VMM. This can be done with the VMI introspector by leveraging the address translation mechanism (Payne et al. 2007). The use of the *xc_map_foreign_range()* function, provided in the Xen Control Library (libxc) helps the VMI introspector of A-IntExt system to understand and reconstruct the volatile memory artifacts of the live Monitored VM without its consent. Later, the same function accesses the RAM artifacts, and finally, converts the page frame number to memory frame number (Russinovich et al. 2012).

In the Windows system, each process associated with a data structure is called as `_EPROCESS`. The Figure 4.4 shows the `_EPROCESS` structure of Windows system, each `_EPROCESS` has many data fields including one Forward Link (`FLINK`) pointer and one Backward Link (`BLINK`) pointer. The `FLINK` contains the address of the next `_EPROCESS`, while the `BLINK` stores the address of the previous `_EPROCESS`. The first field of the `_EPROCESS` is a process control block, which is a structure of the type Kernel Process (`KPROCESS`). The `KPROCESS` is used to provide data related to scheduling and time accounting. The other data fields of the `_EPROCESS` are `PID`, Parent `PID` (`PPID`), exit status, etc., (Russinovich et al. 2012). The field position of the `PID` and the `PPID` in the `_EPROCESS` structure may differ from one OS to another, and the series of `FLINK` and `BLINK` systematizes the `_EPROCESS` data structure in a doubly linked list. The Windows symbol, such as the `PsActiveProcessHead` (head of the doubly linked list) traverses the whole of the doubly linked list `_EPROCESS` from the beginning to the end providing all the running process details. In order to identify the hidden processes at user and kernel-mode of the Monitored VMs, our proposed VMI introspector scans the raw memory by looking at the `_EPROCESS` structure patterns of the Monitored VM, as similar to the previous approach (Jiang et al. 2007; Lamps et al. 2014).

4.3.2 Guest Assisted Module

The GAM is a lightweight component that is placed inside the Monitored VM (soon after the guest OS is created by the VMM). It is controlled and operated by the SI-Requester on-demand via an established secure communication channel. During introspection by the A-IntExt system, the GAM will not create its own processes but will make use of its built-in `tasklist` command of the native Windows to acquire the running processes of the live Monitored VM and forward it to the SI-Requester in the form of a text file. The GAM can be tampered by malware as it is placed in the untrustworthy Monitored VM. Under these circumstances, the A-IntExt system

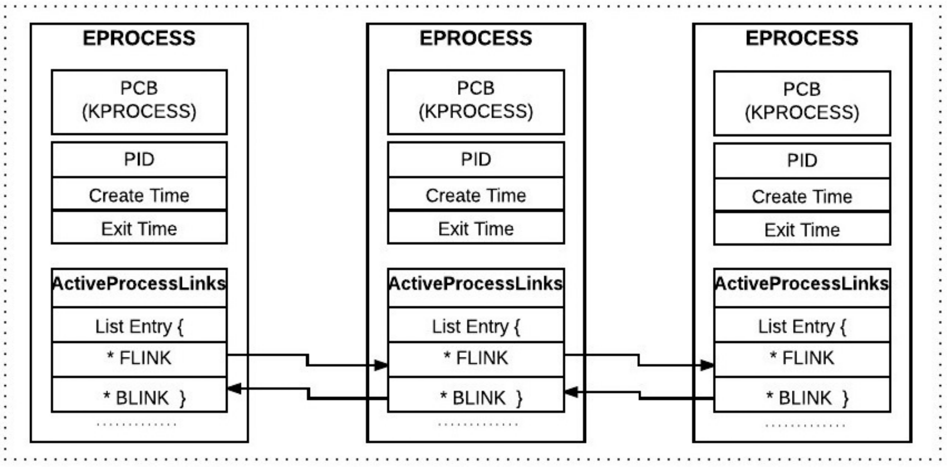


Figure 4.4: Simplified _EPROCESS structure of Windows system (Florio 2005)

estimates the symptoms of malware execution when *delay* or *modification* occurs, while forwarding the internally acquired state information by the GAM (see Section 4.3.1).

4.3.3 Intelligent Cross-View Analyser

The ICVA is an integral component of the A-IntExt system and its prime function is to perform an intelligent examination of the internally captured and externally introspected execution state information using ICVA algorithm to recognize hidden, dead, and dubious running processes of the Monitored VM.

The process details captured from the hypervisor (externally) undergo the preprocessing operation, and then stored as $EXT_{ps} = \{PID \parallel PN_1, PID \parallel PN_2, PID \parallel PN_3....., PID \parallel PN_m\}$ where m = 1, 2, 3,..., and $PID \parallel PN_m$ represent the m^{th} process. The internally captured process details after the preprocess operation are represented as $INT_{ps} = \{PID \parallel PN_1, PID \parallel PN_2, PID \parallel PN_3....., PID \parallel PN_n\}$ where n = 1, 2, 3,..., and $PID \parallel PN_n$ represent the n^{th} process. The ICVA performs the preprocessing operation to remove unimportant state information and sort the elements of both the EXT_{ps} and INT_{ps} in ascending order, based on the PID.

The total number of EXT_{ps} processes is symbolized as EXT_{psc}

$$\sum_{j=1}^{m} PID_j \parallel PN_j = \begin{cases} PID_j||PN_j = 1 \ \text{if}(PID||PN_j \in EXT_{ps}) \\ PID_j||PN_j = 0 \quad else \end{cases} \tag{4.1}$$

The total number of INT_{ps} processes is represented as INT_{psc}

$$\sum_{k=1}^{n} PID_k \parallel PN_k = \begin{cases} PID_k||PN_k = 1 & \text{if}(PID||PN_k \in INT_{ps}) \\ PID_k||PN_k = 0 & else \end{cases} \tag{4.2}$$

$$EXT_{psc} = \sum_{j=1}^{m} PID_j \parallel PN_j \tag{4.3}$$

$$INT_{psc} = \sum_{k=1}^{n} PID_k \parallel PN_k \tag{4.4}$$

Any inconsistency between EXT_{psc} and INT_{psc} i.e. $EXT_{psc} \neq INT_{psc}$ indicates an abnormal state of the Monitor VM. Algorithm 4.1 depicts the procedure followed by the ICVA to perform the cross-examination between the EXT_{ps} and INT_{ps}. At the end of the scrutiny, ICVA provides Hidden Process Count (HPC) and Dead Process Count (DPC), hidden and dead processes.

$$ICVA(EXT_{ps}, INT_{ps}) \rightarrow HPC, DPC, hidden, dead\ \ process \tag{4.5}$$

To ascertain the hidden and dead processes, the ICVA compares the $EXT_{ps}(PID \parallel PN_m)$ with $INT_{ps}(PID \parallel PN_m)$, where $(PID \parallel PN_m)$ is the m^{th} PID and PN. It treats the examined processes as dubious when they are equal. If they are unequal, it checks further to determine whether $EXT_{ps}(PID \parallel PN_m)$ is greater than $INT_{ps}(PID \parallel PN_m)$; if the condition is satisfied, then the ICVA declares the $INT_{ps}(PID \parallel PN_m)$ as a dead process. It continues the comparison operation $EXT_{ps}(PID \parallel PN_m)$ with $INT_{ps}(PID \parallel PN_j)$, where j = m + 1, m + 2,..., until it finds that $EXT_{ps}(PID \parallel PN_m)$ is equal to $INT_{ps}(PID \parallel PN_j)$, and then declares the processes from $INT_{ps} = \{PID \parallel PN_m, ..., PID \parallel PN_{j-1}\}$ as dead processes when the condition $EXT_{ps}(PID \parallel PN_m) = = INT_{ps}(PID \parallel PN_j)$ is satisfied. If $EXT_{ps}(PID \parallel PN_m)$ is less than $INT_{ps}(PID \parallel PN_m)$, then the ICVA declares the $EXT_{ps}(PID \parallel PN_m)$ as a hidden process. The comparison operation between the externally and internally captured state information is continued until all of the elements are examined.

Case 1: $HPC > 0$ indicates that some processes are hidden at the Monitored VM. Equation (4.6) is an indication of malware infection.

$$((EXT_{psc} \neq INT_{psc}) \wedge (HPC > 0)) \tag{4.6}$$

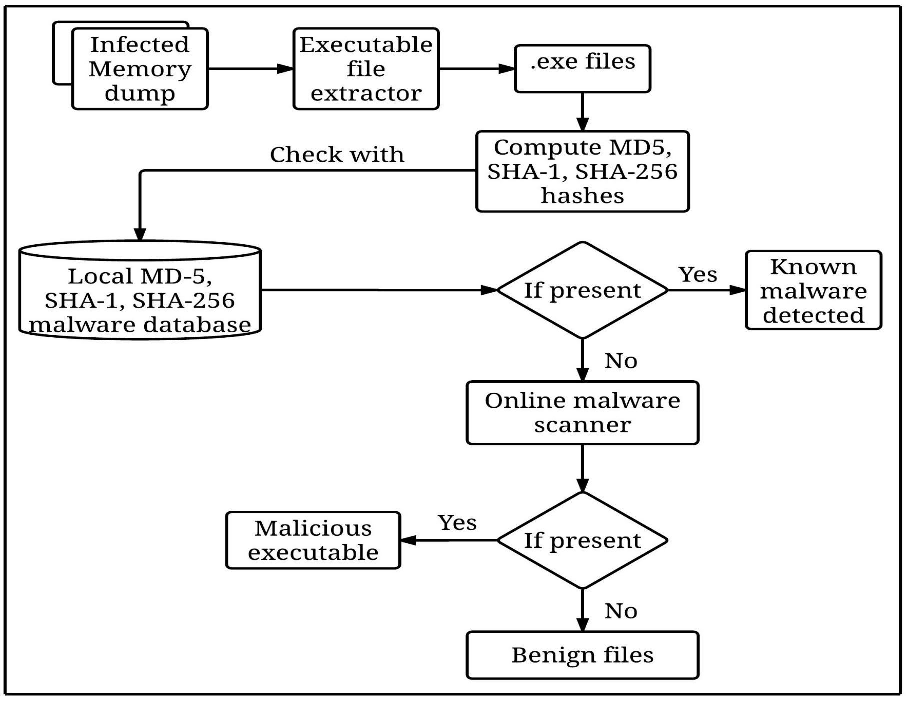


Figure 4.5: Online malware scanner

Case 2: $HPC = 0$ denotes that the processes viewed externally are the same as the processes viewed internally. The state of the Monitored VM is dubious state when equation (4.7) is satisfied.

$$((EXT_{psc} == Int_{psc}) \wedge (HPC == 0)) \tag{4.7}$$

Case 3: The dead process count indicates that the number of processes captured externally is smaller than the number of processes captured internally. This is due to the dynamic nature of the create and destroy of processes. To overcome this situation, A-IntExt system first captures the state information of the Monitored VM internally, followed by externally within the TIT.

$$(EXT_{psc} < INT_{psc}) \tag{4.8}$$

ICVA for processes algorithm has the capacity to recognize hidden, dead and dubious processes. Further, the A-IntExt system classifies the introspected processes as hidden and DPs to ascertain whether the detected hidden process and DPs of Monitored VM are benign or malicious by performing a cross-examination with the public OMS.

Algorithm 4.1: Intelligent Cross View Analyzer for Process

```
1 Input
  1: Processes details captured externally from hypervisor stored as $EXT_{ps}$.
  2: Process details captured and sent by the monitored VM (internally) stored as $INT_{ps}$.
  Output
  1: Hidden and dead processes details
  2: $HPC$ and $DPC$

 1: Pre-process the $EXT_{ps}$ and $INT_{ps}$ such that their elements are in sorted order based
    on PID
 2: Assign $HPC$=0, $DPC$=0, $p$=$EXT_{psc}$, $q$=$INT_{psc}$, $n$=1, m=1
 3: for all $m$ such that $1 \leq m \leq p$ do
 4:   if $n > q$ then
 5:     Break
 6:   else
 7:     compare $EXT_{ps}(PID \parallel PN_m)$ with $INT_{ps}(PID \parallel PN_m)$
 8:     if $EXT_{ps}(PID \parallel PN_m) = INT_{ps}(PID \parallel PN_m)$ then
 9:       $m$=$m$+1; $n$=$n$+1; goto step 4;
10:     else
11:       if $EXT_{ps}(PID \parallel PN_m) < INT_{ps}(PID \parallel PN_m)$ then
12:         store $EXT_{ps}(PID \parallel PN_m)$ as hidden process into HP.txt
13:         $m$=$m$+1; $HPC = HPC + 1$; goto step 4
14:       else
15:         if $EXT_{ps}(PID \parallel PN_m) > INT_{ps}(PID \parallel PN_m)$ then
16:           store $INT_{ps}(PID \parallel PN_m)$ as dead process into DP.txt
17:           $DPC = DPC + 1$; $n$=$n$+1; goto step 4
18:         end if
19:       end if
20:     end if
21:   end if
22: end for
23: if $m < p$ && $n > q$ then
24:   Store $EXT_{ps}(PID \parallel PN_m)$,...,$EXT_{ps}(PID \parallel PN_p)$ as hidden processes into HP.txt
25:   $HPC = HPC + (p\text{-}m)$.
26: end if
27: if $m > p$ && $n < q$ then
28:   Store $INT_{ps}(PID \parallel PN_n)$,..., $INT_{ps}(PID \parallel PN_q)$ as dead processes into DP.txt.
29:   $DPC$=$DPC + (q\text{-}n)$
30: end if
```

4.3.4 Online Malware Scanner

The OMS is another key component of the A-IntExt system and it performs two key functions. First, from the hypervisor, it extracts the complete binary of the hidden processes (executable file) reported by the ICVA. The Figure 4.5 shows the flowchart of OMS. The OMS accomplishes this by utilizing `procdump` plugins of an open source tool[3] on the acquired memory dump of Monitored VM. For each executable file, it computes three distinct hash digests, such as Message Digest (MD5), Secure Hash Algorithm-1 (SHA-1), and Secure Hash Algorithm-256 (SHA-256). Further, these computed hash digests were checked with LMSD[4] to identify any types of hash digests were matched with stored hash digests of known malware types, if not it sends the computed hash digests to powerful public free OMSs and gets an examination report to ascertain whether the extracted executable file is benign or malignant. Similarly, OMS also extracts other processes' executable files that are not classified as hidden processes by the ICVA that are currently being run in the Monitored VM. These processes are named as dubious processes. Like shrouded processes, the OMS additionally registers hash digest for non-concealed processes and sends them to OMS to identify whether the non-concealed process' executable file is malevolent or benign.

4.4 Experimental Results and Evaluation

4.4.1 Experimental Setup

Experiments were conducted on the host system, which possessed the following specifications: Intel(R) core(TM) i7-3770 CPU@3.40 GHz, 8 GB RAM, and Ubuntu 14.04 (Trusty Tahr) 64-bit operating system. The popular open-source Xen 4.4 bare metal hypervisor was utilized to establish a virtualized environment. To introspect the run state of the live Monitored VM, Windows XP-SP3 32 bit guest OS created as DOMU-1 under the Xen hypervisor. The guest OS was managed by the trusted VM (DOM-0 i.e. management unit) of the Xen hypervisor. The A-IntExt system was installed on the DOM-0 VM, and it leveraged the popular VMI tool, namely, the LibVMI version 0.10.1 to introspect low-level artifacts of the guest OSs. The LibVMI traps the

[3]http://www.volatilityfoundation.org/

[4]LMSD consists of 107520 MD5, SHA-1, and SHA-256 hash digest for all previously identified well-known families of malware which was obtained by using https://virusshare.com/ malware repository.

hardware events and accesses the vCPU registers, while functioning at the hypervisor.

4.4.2 Implementation

The implementation of A-IntExt system is at three levels: i) it acts as a VMI system by leveraging a prominent VMI tool to introspect and acquire the guest OS run state information without human intervention, ii) the ICVA algorithm is implemented as PoC and induced into the A-IntExt system, wherein the ICVA detects hidden, dead and dubious processes. In addition, GAM implemented and controlled by A-IntExt system facilitates the transfer of state information of the Monitored VM to the A-IntExt system. iii) The A-IntExt system comprises another major component named OMS (see Section 4.3.4). It is used to identify whether the detected hidden and detected DP or malicious process are benign or malicious by auxiliary verification with LMD and large online free public malware scanners[5].

4.4.3 Experiments and Results Analysis

To convert the benign Windows guest OS into a malicious one and to perform malicious activities on the Monitored guest OS, two stages of experiments were performed using a combination of both malware and publically available Windows rootkits. In the first stage, the evasive malware variant called `Kelihos` was directly collected from malware repository[6] to generate bulk malicious processes. In the second stage of the experiment, five publicly available real-world Windows rootkits that have the ability to hide the processes were used.

Experiment 1: `Kelihos` is a Windows malware also known as Hlux. Once it starts to execute, it generates a number of child processes and then exits from the main process to conceal its existence. It launches a set of processes in a span of a short interval, which influences the process count. The main function of the generated child process is to monitor user activities, and then report it to the Command and Control Server (C&C) to be joined into a botnet. The `Kelihos` malware was used to breed a number of processes, and at the same time, the `Hacker Defender` rootkit was used to hide the process.

[5]https://www.virustotal.com/
[6]http://openmalware.org/

Table 4.1: Detection of hidden, dead and DPs by the A-IntExt system for Windows guest OS

Exp	PS used	PS visible at Monitored VM	PS Introspected by A-IntExt system	No. of PS classified by A-IntExt system			Time in (seconds)
				HPC	DPC	DPs	
Test-1	25	20	25	5	0	20	0.22
Test-2	50	45	50	5	0	45	0.41
Test-3	75	70	74	5	1	69	0.63
Test-4	100	95	99	5	1	94	0.82
Test-5	125	120	123	5	2	118	1.03

This test was done to demonstrate the detection proficiency of the A-IntExt system under a dynamic process creation environment. The A-IntExt system extracts the manipulated semantic kernel data structure details related to the process by walking through the _EPROCESS data structure (see Section 4.3.1).

The ICVA is a subcomponent of the A-IntExt system and its task is to identify hidden, dead and dubious processes by performing a comparison operation on the internally and externally captured state information of the Monitored VM. The performance evaluation tests for both the ICVA and the A-IntExt system were conducted separately. To measure the execution speed of the ICVA in detecting the hidden and dead processes, experiments were performed with different numbers of processes, i.e., 25, 50, 75, 100, and 125. The execution speed denotes the amount of time the ICVA takes to derive a conclusion as to whether the process is hidden, dead or dubious processes. The last column of Table 4.1 depicts the average detection time of the ICVA for different numbers of processes on the Windows guest OS. One can observe that the detection time of the ICVA for 125 processes is 1.03 seconds.

Twenty-five processes were considered in the first test; each test was performed five times to derive the average detection time. Prior to the evaluation, five processes were hidden at the Monitored VM and all of them were correctly detected by the A-IntExt system, including the hidden, dead, and DPs, as shown in Table 4.1. Further, A-IntExt system precisely addresses the malicious process detection challenges by leveraging its OMS component (see Section 4.3.4).

As part of the experimental observations, Test-1 of Table 4.1 describes the 25 processes externally introspected by the A-IntExt system, which includes five hidden

Table 4.2: Identifying an actual malicious process from the detected hidden processes by the OMS of A-IntExt system on Windows guest OS

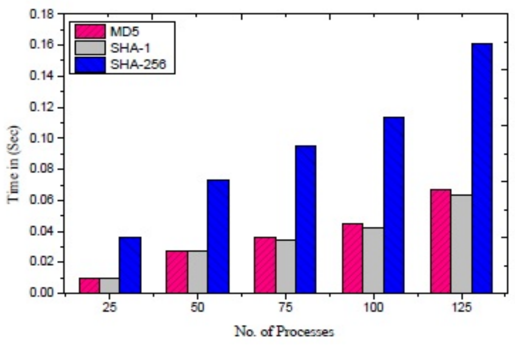
Exp	No.of HP	Computed MD5 hash for classified HP	Checked as	PS name	DR
1	5	55cc1769cef44910bd91b7b73dee1f6c	Malicious	`hxdef073.exe`	37/53
		be046bab4a23f8db568535aaea565f87	N-F	`procdump.exe`	0/53
		6cf0acd321c93eb978c4908deb79b7fb	N-F	`chrome.exe`	0/53
		bf4177e1ee0290c97dbc796e37d9dc75	N-F	`explorer.exe`	0/53
		d068da81e1ab27dc330af91bffd36e6b	N-F	`firefox.exe`	0/53

Table 4.3: Identifying an actual malicious processes from detected and classified DPs by OMS of A-IntExt system on Windows guest OS

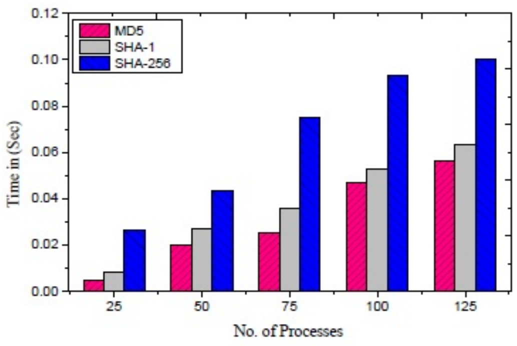
Exp	No.of DPs	Scanned result		Malicous PS reported with MD5 hash	Name	DR
		Benign	Malware			
1	20	18	2	0bf067750c7406cf3373525dd09c293c	`EFMTnkT7m.exe`	–
				5fcfe2ca8f6b8d93bda9b7933763002a	`kelihos_dec.exe`	37/55

processes and twenty DPs that are classified by the ICVA; these hidden and DPs are propagated by the malware. In our experiment-1, we used `kelihos` malware to generate malicious processes (not hidden) and perform spiteful activity on Monitored VM. At the same time, we used `Hacker Defender` rootkit to hide some processes. During introspection of the untrustworthy Monitored VM, A-IntExt system precisely classified the infection activity of the malware processes as hidden and DPs.

Table 4.2 describes that from the five detected hidden processes, one process `hxdef073.exe` is correctly identified as malicious with Detection Rate (DR) of 37/53 based on the computed hash, and the other four processes such as the `procdump.exe`, `chrome.exe`, `explorer.exe`, and `firefox.exe`, which were actually hidden by the `Hacker Defender` rootkit, are reported as benign by the OMS. Similarly, Table 4.3 represents the 20 DPs that were identified by the A-IntExt system. Further, those processes were checked with both LMD and OMS based on the computed hashes. The time taken to compute MD5, SHA-1, SHA-256 hashes and cross-check with LMD are depicted in Figure 4.6. As a result, one process (`EFMTnkT7m.exe`) is identified as malicious by locally checking with LMD (without forwarding to virustotal) and other advanced malware process (`kelihos_dec.exe`) identified as malicous checking with OMS, and the rest were recognized as benign or Nothing-Found (N-F).



(a)



(b)

Figure 4.6: The average time taken by the OMS to compute MD5, SHA-1, and SHA-256 hashes for different processes (5a). Time taken by OMS to detect malware by cross-checking with LMD based on it's computed hashes (5b)

Table 4.4: List and functionality of Windows rootkit

Rootkit name	User-mode/ Kernel-mode	Target object	Hide PS	Detected by A-IntExt system
`Fu Rootkit`	Kernel-mode	_EPROCESS	Yes	Yes
`HE4Hook`	Kernel-mode	_EPROCESS	Yes	Yes
`Vanquish(0.2.1)`	User-mode	IAT, DLL	Yes	Yes
`Hacker Defender`	User-mode	IAT, DLL	Yes	Yes
`AFX Rootkit`	User-mode	IAT, DLL	Yes	Yes

IAT: Interrupt Address Table, DLL: Dynamic Link Library

Experiment 2: In the second stage of the experiment, five publicly available Windows rootkits were used as shown in Table 4.4. The third and fourth columns of Table 4.4 represent target object and complete functionality of the rootkit, respectively. However, in this stage of the experiment, the detection capability of the A-IntExt system was limited to only the processes. For example, the `FU rootkit` leverages the direct kernel object manipulation technique to hide a list of active processes by directly unlinking the doubly linked list `_EPROCESS` data structure. It contains the `fu.exe` executable file and the `msdirectx.sys` system file. The function of hiding the kernel driver module files is achieved by the `msdirectx.sys`, whereas the `fu.exe` file is used to configure and command the driver. The `FU rootkit` is capable of achieving privilege escalation of the running processes and can also alter the DLL semantic object of the kernel data structure by rewriting the kernel memory. The `HE4Hook` is a kernel-mode rootkit and the user-mode rootkits

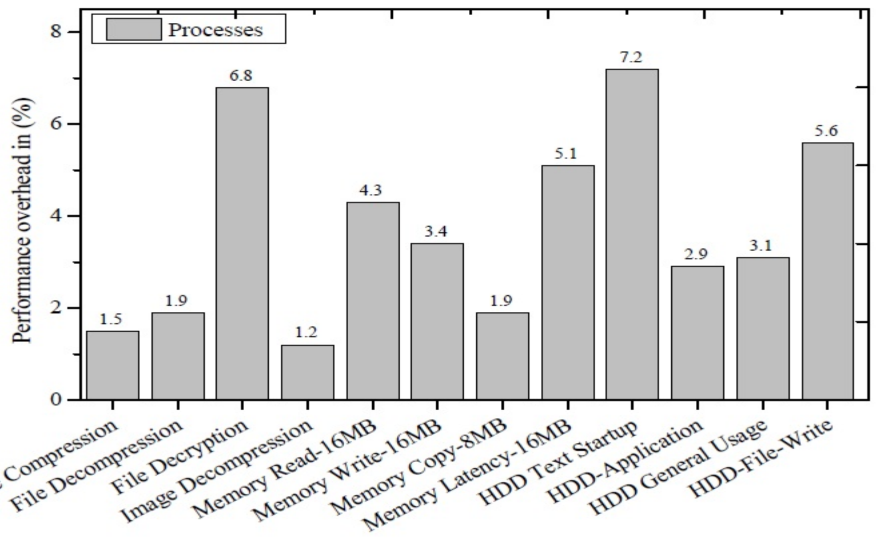


Figure 4.7: Performance impact of A-IntExt system on PCMark05 in detecting hidden and malicious state information of Monitored VM for Windows guest OS

are `Vanquish`, `Hacker defender`, and `AFX Rootkit`. These rootkits have the potential to hide the running processes on the Windows system. The fifth column of Table 4.4 represents the detection of hidden processes performed by the A-IntExt system.

4.5 Performance Overhead

A series of tests were conducted using Windows system benchmark tools to determine the performance impact of the A-IntExt system. The benchmark tests were executed on the Windows guest OS in two different scenarios to evaluate the performance impact of the A-IntExt system. In the first scenario, the A-IntExt system was disabled (not functioning), and in the second scenario, the A-IntExt system was enabled (running). PCMark05, an industry standard benchmark, was executed on the Windows guest OS to quantify the performance impact of the A-IntExt system. Tests such as the CPU, Memory, and HDD of the PCMark05 suite were considered. These tests were executed separately five times on the guest OS. Finally, the results were considered on an average five-time execution of each test.

During hidden, dead and DP detection, tests such as File Decryption, HDD-Text Startup, and HDD-File-Write induced maximum performance overheads of 6.8%, 7.2%, and 5.6%, respectively. Other tests performance overheads observed is less than 5.5 %. These were noticed while the A-IntExt system performed process introspection traces on the executed malware and rootkits. Figure 4.7 represents the overall performance of the A-IntExt system in detecting hidden, dead and dubious processes detection. The main reason for the performance loss is due to direct introspection and the semantic view reconstruction operation performed by the A-IntExt system. As the ICVA achieves the job offline, there is no performance overhead.

4.6 Discussion

The existing VMI techniques facilitate reconstructing a few semantic views of the Monitored VM by directly intercepting the RAM content of the live Monitored VM by overcoming the semantic gap problem. However, these techniques are yet to be intelligent and automated to introspect and accurately detect hidden or malicious semantic state information on their reconstructed high-level semantic view. The design, implementation, and evolution of the proposed A-IntExt system are signified as an intelligent solution to precisely detect the malignant processes running on the Monitored VM. It acts as a perfect VMI-based malware symptoms detector by logically analyzing the malicious infection of the operating systems key source information (processes). The ICVA of the A-IntExt system judiciously performs a cross-examination to detect the hidden-state information of the guest OS that is manipulated by different types of evasive malware or stealthy rootkits. Malicious processes (not-hidden) are identified by the OMS. We believe that the current development of A-IntExt system is proficient in detecting hidden, dead, and malicious processes of any kind of malware or rootkit. However, detecting and identifying both known and unknown malware processes by performing cross-examination with both LMD and powerful online malicious content scanners (virustotal) using its computed hashes (MD5, SHA-1, and SHA-256). The major limitation in identifying malicious processes by the VirusTotal is that it accepts only four requests per minute.

4.7 Summary of the Work

In this work, we designed, implemented, and evaluated the A-IntExt system, which detects hidden, dead and dubious processes by performing an intelligent cross-view analysis on the internally and externally captured run-state information of the Monitored VM. The A-IntExt system abstracts the semantic view (processes) of the live Windows guest OS externally (VMM-level). It uses an established communication channel between the Monitoring VM and Monitored VM to receive internally captured run-state information (at-VM-level). Further, proficiently detecting hidden and malignant state information of the Monitored VM that could be manipulated by sophisticated malware or real-world rootkits. The A-IntExt system is intelligent enough to address the challenges that lie in detecting malicious (not-hidden) processes of the run state of the Monitored VM using its OMS component. Publicly available evasive malware, real-world Windows rootkits were used to perform a series of experiments to accurately measure the hidden-state and malicious detection capability of the A-IntExt system. The experimental results showed the accuracy of the A-IntExt system in detecting stealthy processes with a maximum performance overhead of 7.2%.

Chapter 5

VMM-based Automated Multi-level Malware Detection System

5.1 Introduction

In order to fulfill the requirements like stringent timing restraints and demand on resources, CPS must deploy on the virtualized environment such as cloud computing. The CPS has evolved as a most crucial infrastructure that integrates devices like computation, networking, and physical elements together to facilitate and communicate various applications (Chen et al. 2016). Primary goals of the CPS is to enable intelligent monitoring as well as controlling of the various applications running on the computing devices. In other words, CPS main objective is to assist quick extraction of information, analysis of the data, decision making and data transmission in real-time. The CPS has been widely adopted in various fields like robotics, health care, military, industrial control, power systems, avionics systems, intelligent building, smart electrical power grids, smart medical systems, etc., (Ma et al. 2016).

The central processing core of the CPS receives a massive amount of data from various cyber-enabled devices as well as other physical devices through communication networks. For example, in a smart and reliable transportation system, a vehicle equipped with the sensor device sends data to the roadside unit deployed on the roads which in turn transmits the received data to central processing core of the CPS over a high-speed network. Since a large number of devices send the data, inbound traffic at the central processing core of the CPS is massive. Generally, CPSs are real-time systems, thus, the significance of computational latency of their critical components are as same as their correctness of their functional modules. In order to mitigate the loss caused due to the violation of the real-time property of a perilous function, the best solution is to deploy central processing core of the CPS in the cloud computing (virtualized) environment. The main aim of development of the cloud computing is to offer fast computational speed. Moreover, the cloud computing can offer a vast pool of resources to store, process, and analyze the data, which creates precise data information. Elasticity property of the cloud computing offers adequate amount of resources to central processing core of the CPS as and when the demand arises (Chaâri et al. 2016). The virtualization enables a number of benefits such as reduced operation and maintenance cost as well as setup cost, easy the procurement process, more importantly, dependability and availability.

Much of the works have been done in the context of hardware virtualization and fault-tolerance on virtualization (Nagarajan et al. 2007; Jablkowski and Spinczyk 2015). Recent work (Jablkowski et al. 2017) provides the aspects of integration and consolidation of CPS with virtualization. Since central processing core of the CPS frequently communicates with the other systems in the physical world through communication media, malware can be used as a weapon by an attacker or malignant user who intentionally desire to create havoc (Reddy 2015).

Malware is a malicious program developed with an intention to launch malignant tasks. Generally, malware uses the stealthy techniques to exploit the system and network vulnerabilities in order to gain control of the user system to achieve unauthorized activities (Reddy 2015). Its prime target is not only restricted to destroy the single system or group of systems, it also targets to disrupt the normal functions of the computer networks (Moskovitch et al. 2008). This results in increasing threat to the information systems that are used in day to day activities. Malware not only makes use of zero-day exploits to acquire the control of vulnerable machine but also stealthily achieve its intended job by hiding in an infected system and cause contaminations over time. The proliferation of new variants or a particular class of malware constantly uses code obfuscation technique (Lin and Stamp 2011) or rootkit functionality (Goudey 2012a) to subvert most of the existing in-host security solutions to gain access to the targeted machine.

Since VMs are easily available through the CSP, the virtualized cloud environment is the prime target by an attacker (Pearce et al. 2013). To protect VMs in which CPSs are functioning against malware-based attacks, malware detection and mitigation technique is emerging as a highly crucial concern. The traditional VM-based anti-malware software themselves a potential target for malware-based attack since they are easily subverted by sophisticated malware. Thus, a reliable and robust malware monitoring and detection systems are needed to detect and mitigate rapidly the malware-based cyber-attacks in real-time particularly for a virtualized environment.

To tackle this issue, VMI (Garfinkel et al. 2003) has emerged as a promising out-of-VM security solution that operates at the VMM and facilitates in constructing a semantic view of the live guest OS in real-time by introspecting low-level details of the volatile memory state of the introspected guest OS without the consent of one being monitored.

Other significant challenges in precisely detecting the malicious executables, mainly in a virtualized environment are:

- In order to provide real-time protection for CPS which is operating within the guest OS in a virtualized cloud environment, the traditional VM-based security solutions are inadequate to protect guest OS resources against the sophisticated malware. Therefore, VMI techniques provide out-of-VM security solution for the introspected guest OS while operating at VMM. However, introspected information (e.g., processes) was available in dubious forms[1]. For example, the proliferation of the `Kelihos` malware on the guest OS spawned a number of malicious child processes before exiting from the main process (Garnaeva 2012). In such an instance, manually distinguishing, detecting, and preventing the running malicious processes from hundreds of benign processes was time-consuming for a security administrator, as it required a wide knowledge of the malicious executables.

- The CPS functioning on the virtualized environment (e.g., guest OS) is targeted by malicious executables use the code obfuscation techniques (Sharif et al. 2009; Bayer et al. 2009) and other stealthy malware attacks (Jiang et al. 2007). Hence,

[1]Dubious process is a process that is currently running on a guest OS, and it may or may not be a malicious process (not hidden).

performing early symptoms of malware execution and accurately estimating the stealthy hidden, dead and dubious malicious processes under the dynamic nature of the process creation and expire on introspected live guest OS is challenging task.

To address the aforementioned challenges, we propose an intelligent and guest-assisted AMMDS that leverages both the VMI and MFA techniques to perform three levels of the investigation to secure the critical infrastructure of the virtualized cloud environment. As a first level of investigation, it performs introspection and precisely detects the semantic view of the hidden and malicious processes to estimate the perfect infection state of the live introspected guest OS. It seizes the execution state of the introspected guest OS by capturing the memory dump of the monitored guest OS soon after it identifies the unusual behavior and then instantly reconstructs and extracts executables from the acquired memory dump to carry out next level of investigations. As a second level of investigation, the OMD component of the AMMDS examines the extracted executables to ascertain the malicious one. The OFMC component of AMMDS analyses the extracted executables in order to identify unknown or zero-day malware using machine learning techniques as a third level of investigation. The AMMDS is evaluated by using real malware datasets on the virtualized environment established using Xen hypervisor. Our empirical results show that AMMDS is robust in detecting and classifying unknown malware that can evade VM-based security solution, and it only incurs acceptable moderate run-time overhead.

The key contributions of the present work are as follows:

1. We have designed, implemented, and evaluated a consistent, real-time VMM-based guest-assisted AMMDS that periodically examines the state of the live guest OS system while defending the CPS to detect the running malicious processes from the forensically reconstructed executables.

2. OMD and OFMC are two prime sub-components of the AMMDS. These are practically implemented and implanted into AMMDS to distinguish an actual malware from semantic view processes that are reconstructed as dubious executables at VMM. The OMD performs a malicious check on hidden and dubious processes by cross-verifying with its LMSD and OMD. On the other hand,

OFMC uses the extracted features recommended by feature selection techniques in the form of Final Feature Vector (FFV) to perform malware analysis in offline.

3. To the best of our knowledge, our proposed AMMDS is the first to adopt machine learning techniques from a VMI perspective at the VMM, and to perform runtime detection of unknown malware from the introspected and forensically extracted executables of the introspected guest OS. This idea opens the door for researchers to leverage other scientific techniques at the VMM to perform automatic detection of malware.

4. The robustness of the AMMDS was practically evaluated by injecting large samples of real-world Windows malware and rootkits on a live Windows guest OS. The OMD of the AMMDS is powerful enough to distinguish between malicious and benign executables. Similarly, the OFMC achieved malware detection rate of 100%, and 0% FPR with maximum performance overhead of 5.8%.

5.2 Overview of AMMDS

The AMMDS is a VMM-based guest-assisted introspection system that advances the current out-of-VM security approach in an automated, isolated, real-time, scientific manner, while functioning at the secure Monitoring VM or Dom0. Figure 4.1 shows an overview of the AMMDS, its major components are: Guest Virtual Machine Introspector (GVM-Introspector), ICVA, and Malware detector. It is introduced to efficiently investigate and detect any hidden, dead, and dubious processes, while predicting early infection of the malware symptoms by internally gathering and externally introspecting the volatile memory of the live untrusted Monitored VM. The AMMDS achieves this goal by using its integral component called ICVA. Furthermore, Malware detector of the AMMDS consisting of two subcomponents that are OMD and OFMC. The OMD identifies whether the detected hidden and dubious process is malicious or benign by cross-comparing with its LMSD and OMS based on the computed hash digest for each of the extracted dubious executables. The OFMC leverages a practical machine learning techniques to classify the execution of the unknown malware from the reconstructed dubious semantic view of the processes (i.e., noticed from a VMI

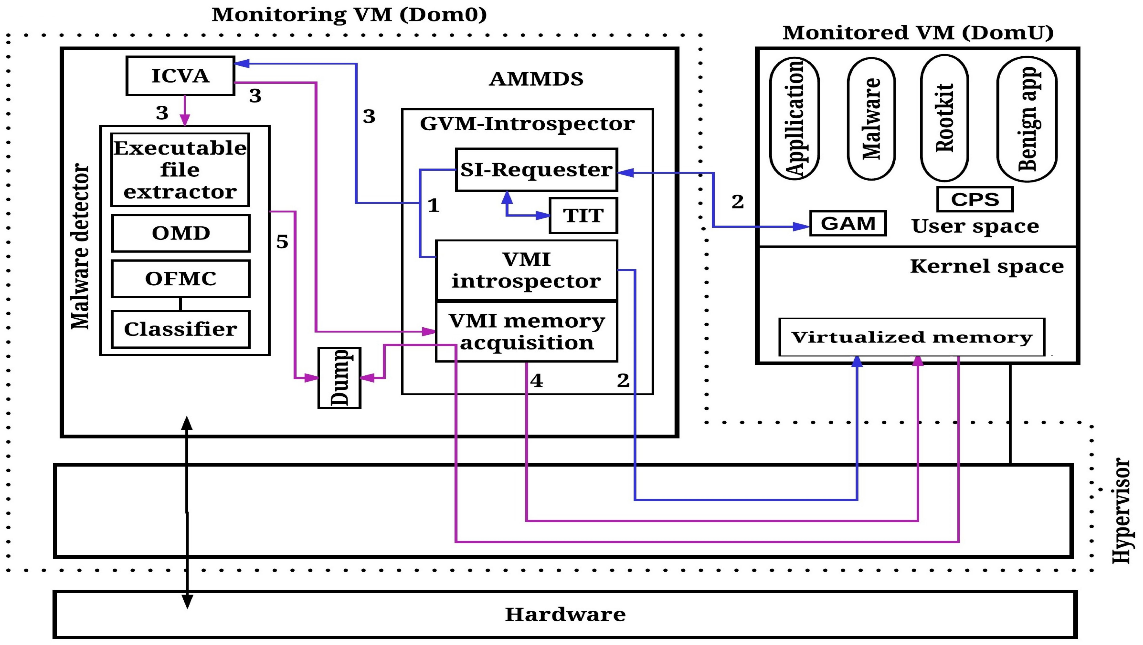


Figure 5.1: The proposed VMI based A-IntExt system

perspective) forensically extracted as executables from the seized live memory dump of the introspected guest OS.

The ICVA conducts the preprocessing operation while removing unimportant state information and sorts the elements of both the EXT_{ps} and INT_{ps} in ascending order based on the PID. The total number of processes gathered from the EXT_{ps} are symbolized as EXT_{psc}

$$EXT_{psc} = \mid EXT_{ps} \mid \tag{5.1}$$

Similarly, the total number of processes gathered from INT_{ps} are symbolized as INT_{psc}

$$INT_{psc} = \mid INT_{ps} \mid \tag{5.2}$$

where, $\mid INT_{ps} \mid$ and $\mid EXT_{ps} \mid$ represent the cardinality of INT_{ps} and EXT_{ps}, respectively. The cardinality of INT_{ps}, EXT_{ps} is the total number of elements (processes) in the INT_{ps} and EXT_{ps}.

The ICVA proficiently detects the hidden, dead, and dubious running processes and predicts the symptoms of the malware execution on the Monitored VM based on the decision function as follows:

$$ICVA(EXT_{ps}, INT_{ps}) = \begin{cases} Hidden & \text{if } (PID/PN \in EXT_{ps} \; and \; PID/PN \notin INT_{ps}) \\ Dead & \text{if } (PID/PN \notin EXT_{ps} \; and \; PID/PN \in INT_{ps}) \\ Dubious & \text{if } (PID/PN \in EXT_{ps} \; and \; PID/PN \in INT_{ps}) \end{cases} \tag{5.3}$$

However, at the end of the scrutiny, it is not feasible for the ICVA to check whether these processes are malicious or not. To resolve this ambiguity, it commands (on confirmation of Equation 5.3) the VMI memory acquisition (step 4) to pause and perform memory acquisition of the Monitored VM. At the same time, the executable file extractor (step 5) component extracts all the hidden and active dubious executables (.exe).

5.2.1 Malware Detector

The Malware detector of the AMMDS consists of three sub-components that are Executable file extractor, OMD, and OFMC.

5.2.2 Executable File Extractor

The major function of the executable file extractor is to extract complete executables that correspond to detected hidden and dubious processes as indicated by the ICVA. The executable file extractor accomplishes this task by utilizing the `procdump` plugin of an open source Volatility tool[2] based on the consistent state of seized memory dump of the Monitored VM. The executable file reconstruction is achieved by parsing the PE header data structure (Pietrek 1994; Ligh et al. 2014) from the obtained VM memory dump. Once the hidden and dubious executables are extracted, the OMD and OFMC investigate them individually to ascertain any malicious substance is present in the reconstructed executables.

5.2.3 Online Malware Detector

The OMD computes a hash digest for the extracted executables soon after receiving the confirmation from the executable file extractor. The OMD computes three distinct hash digests such as SHA-256, SHA-1, and MD5, for each of the extracted executables.

[2]http://www.volatilityfoundation.org/

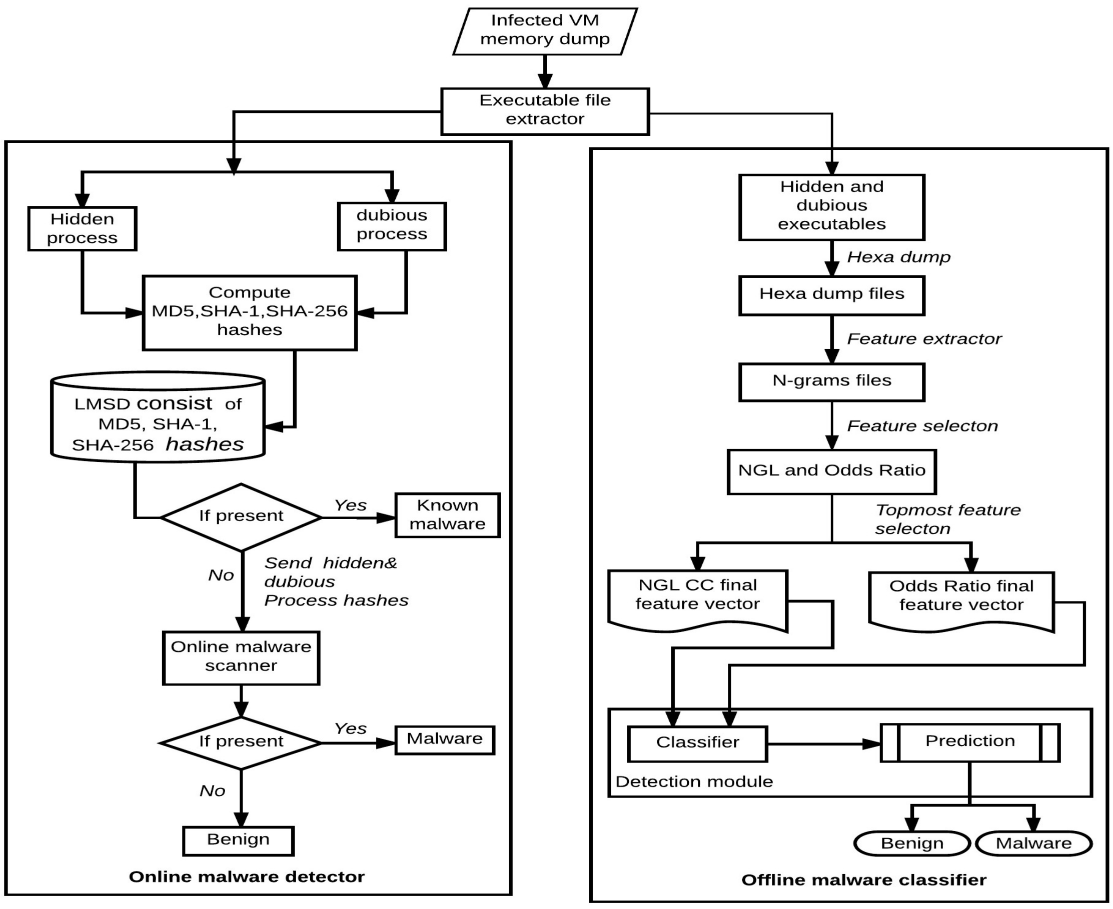


Figure 5.2: Flow chart of the AMMDS for detection of malware using OMD and OFMC components

Further, these computed hash digests checked with LMSD[3] to identify the malware which are known in the digital world. If the OMD does not find any match, then it sends the generated hash digests to powerful publicly available free online malware scanner[4] to obtain an analysis report of the examination that provides whether the tested executable file is malware or benign. The left side of the Figure 5.2 depicts the sequence of operations followed by the OMD to check maliciousness of the given input executables. However, the main limitation of the OMS is that it is unable to detect the new variants of malware due to unavailability of a new malware signature in its database. Meanwhile, malware detector uses the OFMC to detect and classify unknown malware using machine learning techniques.

[3]LMSD consists of 107520 MD5, SHA1, and SHA256 hash digests for known malware which were downloaded from https://virusshare.com/ malware repository.

[4]https://www.virustotal.com/

5.2.4 Offline Malware Classifier

The OFMC is another important subcomponent of the malware detector and it addresses the limitation of the OMD, while accurately classifying any kind of executables as benign or malware by employing machine learning techniques. For any machine learning based malware detection approach, first, the classifier model should be trained with a sufficient number of benign and malware executables so that the classifier model can easily and quickly distinguish malware executables from benign executables. The training phase of OFMC comprises of feature extraction and feature selection techniques (will be shortly introduced) and FFV generation which in turn needed to perform malware classification. In the evaluation phase, we perform detection of unknown malware which is forensically reconstructed as detected hidden and dubious executables based on the trained classifier. In this evaluation phase, the OFMC functions on the reconstructed executables by extracting N-grams as features and then prepares the testing file by using FFV with extracted N-grams of an executable file to be verified. The right side of the Figure 5.2 depicts the steps followed by the OFMC to identify the given test input executables as benign and malicious.

Feature extraction technique: The executables are used as input files in the first step of feature extraction to extract the hexadecimal dump, and then pre-process the hexadecimal dump to remove any irrelevant information. After the pre-processing operation, only the byte sequences that represent a snippet of the machine code of the executable received. The extracted byte sequences are grouped in the form of N-gram (Reddy and Pujari 2006), which represents contiguous bytes sequences, where N represents the number of bytes. In this work, we have chosen N-gram of size 4 byte for the OFMC experimental analysis to achieve best malware detection rate. The steps involved to obtain the N-grams from the executables is shown in Algorithm 5.1. Each individual N-gram is considered as a feature and all N-grams of all executables in the training dataset are treated as an original feature vector.

The N-gram based technique produces a large number of N-grams that include duplicate N-grams. All the N-grams cannot be used as final features to generate a training file as well as a testing file as needed by the classifier because it may result in memory consumption and performance overhead. In addition, it may also reduce

```
Algorithm 5.1: Feature extraction
  Input  : Binary files (.exe) F = {$f_1$, $f_2$, $f_3$, ... ,$f_M$}
  Output: N-gram files $f_N$ = {$f_{N_1}$, $f_{N_2}$, ... ,$f_{N_M}$}
1 foreach file $f_i \in F$ do
2     Extract hexadump, N-grams
3     $hd_i \longleftarrow$ hexadump($f_i$)    // hd - hexadecimal dump
4     $g_i \longleftarrow$ preprocess($hd_i$)    // g - temporary file
5     Create N-gram file $f_{N_i}$
6     while not EOF($g_i$) do
7         N-gram $\longleftarrow$ N-gram ($g_i$)
8         $f_{N_i}$.append(N-gram)
9     end
10 end
11 $f_N = \{f_{N_1}, f_{N_2}, ... ,f_{N_M}\}$
```

the predictive performance of the classifier with more FPR. In order to address these issues, the feature selection technique is employed.

Feature selection technique: The approach for selecting the best features from the original feature vector plays an imperative role in classifying the input files accurately. The feature selection step identifies which features are highly crucial and which are noisy from the original feature vector by generating a score for each feature. The noisy features are ignored because they degrade the predictive accuracy of the classifier. In this work, OFMC comprising of two statistical techniques such as NGL Correlation Coefficient (CC) and Odds Ratio to rank each feature individually and recommend the prominent features based the procedure described in Algorithm 5.2.

a) NGL Correlation Coefficient (NGL CC): it is a variant of the chi-square test (Dave 2011). It selects the features (N-grams) that are correlate with class C_i and does not select features that are correlated with other classes. The NGL CC score for a given N-gram of class C_i is computed as follows:

$$NGL(\text{N-gram}, C_i) = \frac{\sqrt{N}(PS - RQ)}{\sqrt{(P+R)(Q+S)(P+Q)(R+S)}} \tag{5.4}$$

Where, N represents the total number of N-gram files in the dataset, P indicates the number of N-gram files in class C_i that contains N-gram, Q is the number of N-gram files other than class C_i that contains N-gram, R is the number of N-gram files in class C_i that does not contain the N-gram, and S is the number of N-gram files that does not contain the N-gram, other than class C_i. The class C_i = {benign, malware}.

b) Odds ratio: It is one of the popular feature selection techniques. A positive score of Odds Ratio indicates that the given N-gram often appears in a given class as compared to other class. A negative score represents that the given N-gram presence is more in the other class. Odds Ratio for binary classification is defined as follows (Mladenic and Grobelnik 1999):

$$OddsRatio(\text{N-gram}, C_i) = \log \frac{P(\text{N-gram} \mid benign)(1 - P(\text{N-gram} \mid malware))}{P(\text{N-gram} \mid malware)(1 - P(\text{N-gram} \mid benign))} \tag{5.5}$$

P(N-gram | *benign*) is the probability of occurrence of N-gram in benign class. Similarly, P(N-gram| *malware*) is the probability of occurrence N-gram in malware class. The class C_i = {benign, malware}.

5.3 Implementation and Evaluation

5.3.1 Experimental Setup

To evaluate the efficiency of our proposed AMMDS, the host system that had the following specifications: Ubuntu 14.04 (Trusty Tahr) 64-bit operating system, 8 GB RAM, Intel(R) core(TM) i7-3770 CPU@3.40 GHz, was utilized to conduct experiments. The popular open-source bare metal hypervisor such as Xen 4.4.1 was used to set-up a virtualized environment. The Windows XP SP3 32 bit Monitored VM was created as DomU to act as CPS and it was controlled by the Monitoring VM (Dom0) of the Xen hypervisor. The AMMDS was installed on the Monitoring VM and the popular open source VMI introspection toll, such as the LibVMI version 0.10.1 to acquire the RAM dump of the Monitored VM. An open source memory forensics analyzer such as Volatility was applied to construct executables from the acquired infected memory dump. The machine learning algorithms' suite such as WEKA (Hall et al. 2009) was employed to achieve offline malware detection and classification at VMM level.

5.3.2 Implementation

Our proposed AMMDS is implemented using the Python programming language and its implementation was structured at three levels: i) the AMMDS functions as an

Algorithm 5.2: Feature selection

```
Input  : $\triangle$ = {B = {$f_{N_{B_1}}$, $f_{N_{B_2}}$, ... ,$f_{N_{B_M}}$} ∪ M = {$f_{N_{M_1}}$, $f_{N_{M_2}}$, ... ,$f_{N_{M_M}}$} }
Output: Selected features (N-grams)

1  Create files BOR, MOR, BNGL and MNGL
2     // B(M)OR - benign (malware) Odds Ratio
3     // B(M)NGL - benign (malware) NGL
4  foreach file $f_{N_i} \in \triangle$ do
5      foreach N-gram $\in f_{N_i}$ do
6          Compute Odds-Ratio as per equation(5)
7          OR_score ⟵ Odds-Ratio(N-gram)
8          if ($f_{N_i} \in B$) then
9              Store N-gram and OR_score into a file
10             BOR.append(N-gram, OR_score)
11             else
12             MOR.append(N-gram, OR_score)
13         end
14         Compute NGL score as per equation(4)
15         NGL_score ⟵ NGL(N-gram)
16         if ($f_{N_i} \in B$) then
17             Store N-gram and NGL_score into a file
18             BNGL.append(N-gram, NGL_score)
19             else
20             MNGL.append(N-gram, NGL_score)
21         end
22     end
23 end
24 Sort BOR and MOR in descending order based on OR_score
25 Sort BNGL and MNGL in descending order based on NGL_score
26 Select top 'L' number of N-grams from BOR and MOR separately
27 Select top 'L' number of N-grams from BNGL and MNGL separately
```

advanced automated VMI-based security solution by using open source VMI tool to introspect semantic view of run state of the Monitored VM. Further, it is also acquiring the memory dump of the Monitored VM when symptoms of malware are detected at the introspected guest OS without the knowledge of one being monitored. ii) ICVA is implemented as PoC and induced into the AMMDS, which detects hidden, dead, and dubious processes. iii) Implementation of the malware detection component includes both OMD and OFMC. The OMD checks the detected hidden and dubious process as benign and malware by cross-comparing with both LMSD and online malware scanner. At the same time, implementation of OFMC uses feature extraction and feature selection techniques (see Algorithm 5.1 and 5.2) facilitate to construct

FFV in order to detect actual unknown malware from introspection-cum-forensically extracted hidden and dubious executables using trained machine learning classifiers at VMM.

5.3.3 Dataset Creation and Use

About 3375 Windows malware samples (executables) were collected from the VX Heaven[5] malware repository by directly connecting the Windows Monitored VM to an external network. In addition, 675 Windows benign executables were also collected from a freshly installed Windows Monitored VM along with other data source[6] that included both Windows native utilities and application executables. These executables were invoked by our developed program on the Windows Monitored VM with different experimental scenarios, including the installation of 2 rootkits such as `Hacker Defender` and `FU Rootkit`. These rootkits used to explicitly hide the running benign and malware processes on the guest OS. The experimental results depicted in Table 5.1 illustrate the execution of different categories of malware samples on a live Monitored VM.

5.3.4 Evaluation and Results Discussion

In this section, we discussed the evaluation and results analysis of our proposed approach based on VMM-based generated dataset that was generated by executing benign and malicious executables on live Monitored VM. In order to prepare the training and testing files that are required to evaluate the performance of the OFMC, the collected malware, and benign executables were divided into two parts: Set-A and Set-B. The Set-A consisted of 60% of total samples and these were used to train the classifier and the remaining 40% of samples were grouped into Set-B. In the testing phase, malware and benign samples from Set-B were executed on live Windows XP Monitored VM in different experimental cases (as shown in Table 5.1) while measuring the detection proficiency of the AMMDS to detect hidden, dead and dubious processes. Finally, the semantic views of these detected processes were forensically reconstructed as executables from the infected memory dump of the guest OS. These injected malware

[5]http://vxheaven.org/

[6]http://download.cnet.com/windows/, accessed on July 2016

and benign samples were forensically extracted in the form of dubious executables to detect actual malicious executables by following the procedure discussed in Section 4.3.1.

The experiments were performed at different stages to evaluate the malware detection proficiency of the AMMDS. The procedure employed by the OMD to detect whether the extracted executables is malware or benign is discussed in Section 5.2.3. As part of the experimental observations, we have performed six different tests using different types of malware and benign executables on a freshly installed Windows Monitored VM for each test as shown in Table 5.1. More precisely, in test I, we executed 160 Trojan and 45 benign files, totaling 205 executables on a clean live Windows guest OS. Once all the executables were injected, some Trojan executables hid or disappeared on the Windows guest OS. At the same time, we also injected a `Hacker Defender` user-mode rootkit to explicitly hide two benign (`explore.exe` and `chrome.exe`) and one malware (`Trojan.Win32.exe`) process running on the Monitored VM. These experiments lasted 4 minutes. A periodic introspection of the AMMDS system helped in identifying the symptoms of malware execution by recognizing the disparity of the processes that emerged between the internal and external views of the processes state information of the Monitored VM (see Section 4.3.3).

As seen in test I of Table 5.1, the AMMDS system ascertained and counted the introspected processes, namely, internal, external, hidden, dead, and dubious processes as 221, 225, 5, 1, and 220, respectively. However, there were variations in the detected processes counted on the internally captured and externally introspected process state information including 21 (additional) running default processes of the clean Windows guest OS.

More specifically, the internal view of the total processes visible is 221, i.e., from the 205 total injected malicious and benign executables, 5 processes (3 were self-hidden by the Trojan and rootkit and 2 were benign processes that are `explore.exe` and `chrome.exe` explicitly hidden by the `Hacker Defender` rootkit) are hidden at

Table 5.1: Execution of malware and benign executables on live Monitored VM in different experimental test cases

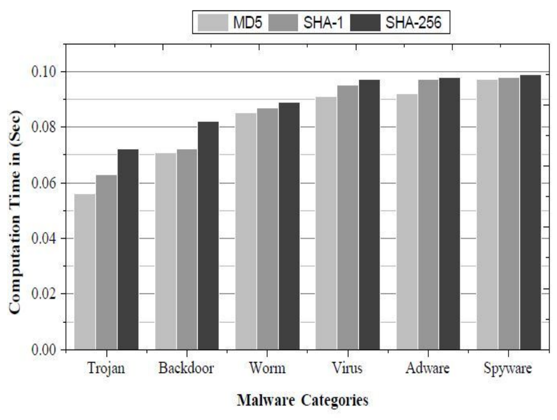
Test #.	Malware types	# Executables used			# PS visible by internal view	# PS introspected from external view	# PS detected by AMMDS			
		Malware	Benign	Total			Hidden	Dead	Dubious	Time (in seconds)
I	Trojan	160	45	205	221	225	5	1	220	2.45
II	Backdoor	185	45	230	249	250	2	1	248	2.25
III	Worm	210	45	255	273	274	3	2	271	2.81
IV	Virus	230	45	275	292	294	4	2	290	2.55
V	Adware	270	45	315	333	345	3	1	342	2.62
VI	Spyware	295	45	340	357	358	4	3	354	2.75
Total # of executables		**1350**	**270**				**21**		**1725**	

Table 5.2: Identifying an actual malicious process from test-I of detected hidden processes by the OMD of AMMDS

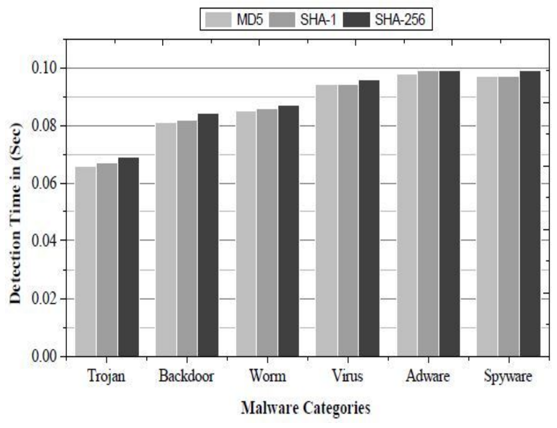
Malware type	No. of hidden process to be checked	MD5 hash digest for classified hidden process	Predicted as	PS name	Detection Rate
Trojan	5	55cc1769cef44910bd91b7b73dee1f6c	Malicious	`hxdef073.exe`	51/53
		6cf0acd321c93eb978c4908deb79b7fb	Benign	`chrome.exe`	0/53
		bf4177e1ee0290c97dbc796e37d9dc75	Benign	`explore.exe`	0/53
		1338dfc088a24a477dd3c6d65fe71b9b	Malicious	`stubd.exe`	33/43
		5fcfe2ca8f6b8d93bda9b7933763002a	Malicious	`kelihos dec.exe`	36/54

Table 5.3: Identifying an actual malicious process from test-I of detected dubious executables by the OMD of AMMDS

Malware type	No. of dubious process to be checked	MD5 hash digest for classified dubious process	Predicted as	PS name
Trojan	220	0bf067750c7406cf3373525dd09c293c	Known malware	`EFMTnkTm-216.exe`
		376121485bee9e8885d879d5407388c3	Known malware	`Win32.Hidedoor.exe`



(a)



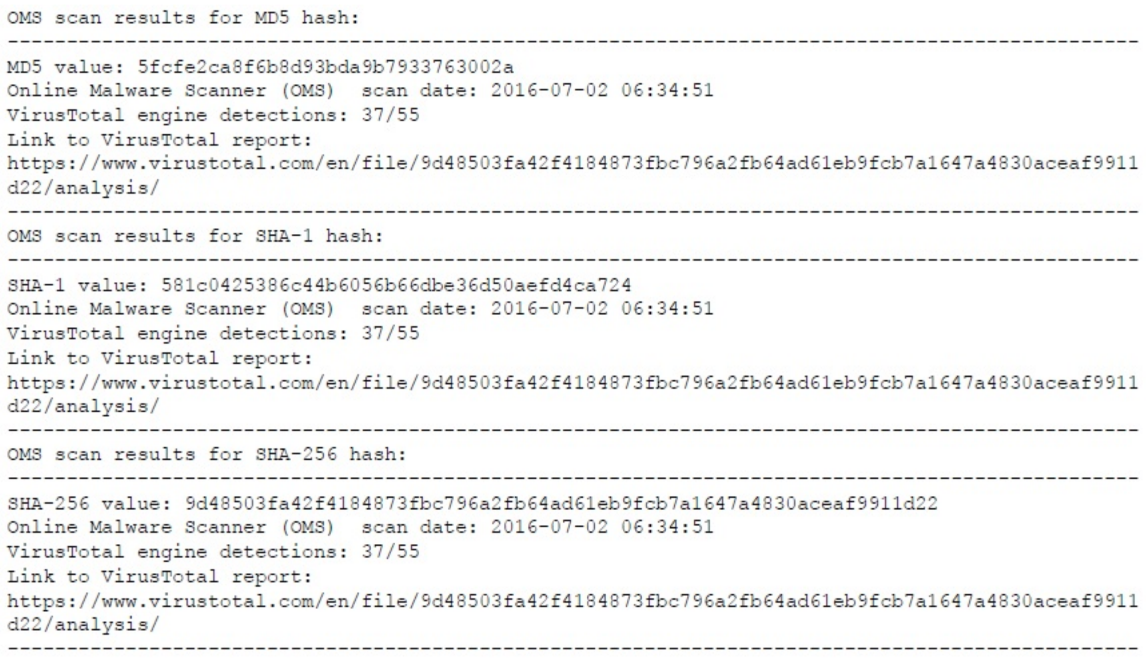
(b)

Figure 5.3: The average time consumed by the OMD to generate SHA-256, SHA-1, and MD5 hash digest for execution of different types of malware (4a). Time taken by the OMD to identify the known malware by cross-checking with LMSD based on computed hash digest (4b)

the guest OS and 21 are native running processes of the guest OS. Finally, 221 are internally gathered processes. Note that these 5 hidden processes are not gathered by the GAM (i.e., internal view of guest OS) as it does not appear in the `tasklist` command of the GAM (see Section 4.3.2). Finally, the VMI introspector of the AMMDS externally (i.e., VMM level) introspect and ascertain these 5 hidden processes.

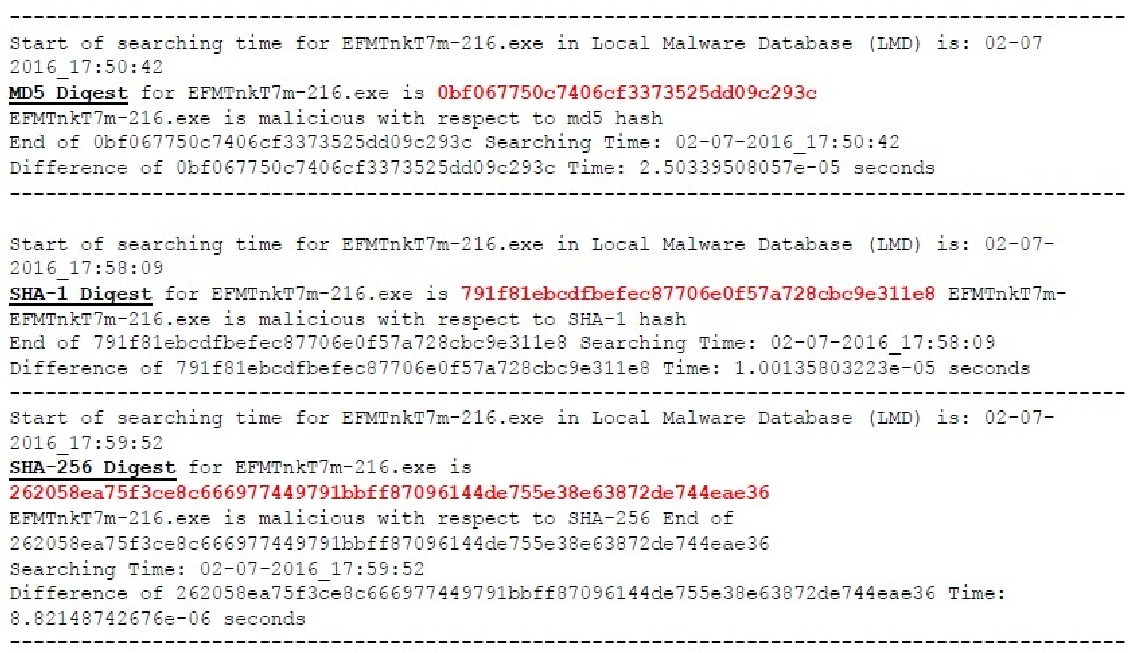
Similarly, in the II experiment, we injected 185 Backdoor and 45 benign files, totaling 230 executables on the clean Windows guest OS. Meanwhile, in this test we have injected DKOM based kernel-mode, `FU Rootkit` to explicitly hide a few benign and malware running processes by directly removing or unlinking it from the `_EPROCESS` data structure of the process list at the kernel-mode. Finally, the disparity of the processes is identified by the ICVA of the AMMDS similar to test I. Likewise, experiment III, IV, V, and, VI were conducted by executing other types of malware and benign executables on the guest OS. Finally, six different experimental test cases, the AMMDS reconstructed over 21 hidden and 1725 dubious executable as VMM-based generated malware dataset from six infected memory dumps of the guest OS at VMM as shown in Table 5.1.

In each test cases, the OMD computes hashes digest for all the reconstructed executables from the captured memory dump and then performed cross-examination with LMSD and OMS based on the computed hash digest. The time elapsed to generate



```
OMS scan results for MD5 hash:
----------------------------------------------------------------------------------------------
MD5 value: 5fcfe2ca8f6b8d93bda9b7933763002a
Online Malware Scanner (OMS)  scan date: 2016-07-02 06:34:51
VirusTotal engine detections: 37/55
Link to VirusTotal report:
https://www.virustotal.com/en/file/9d48503fa42f4184873fbc796a2fb64ad61eb9fcb7a1647a4830aceaf9911
d22/analysis/
----------------------------------------------------------------------------------------------
OMS scan results for SHA-1 hash:
----------------------------------------------------------------------------------------------
SHA-1 value: 581c0425386c44b6056b66dbe36d50aefd4ca724
Online Malware Scanner (OMS)  scan date: 2016-07-02 06:34:51
VirusTotal engine detections: 37/55
Link to VirusTotal report:
https://www.virustotal.com/en/file/9d48503fa42f4184873fbc796a2fb64ad61eb9fcb7a1647a4830aceaf9911
d22/analysis/
----------------------------------------------------------------------------------------------
OMS scan results for SHA-256 hash:
----------------------------------------------------------------------------------------------
SHA-256 value: 9d48503fa42f4184873fbc796a2fb64ad61eb9fcb7a1647a4830aceaf9911d22
Online Malware Scanner (OMS)  scan date: 2016-07-02 06:34:51
VirusTotal engine detections: 37/55
Link to VirusTotal report:
https://www.virustotal.com/en/file/9d48503fa42f4184873fbc796a2fb64ad61eb9fcb7a1647a4830aceaf9911
d22/analysis/
----------------------------------------------------------------------------------------------
```

Figure 5.4: Snapshot of OMD for identification of malicious (not hidden) process `kelihos_dec.exe` from OMS



```
----------------------------------------------------------------------------------------------
Start of searching time for EFMTnkT7m-216.exe in Local Malware Database (LMD) is: 02-07
2016_17:50:42
MD5 Digest for EFMTnkT7m-216.exe is 0bf067750c7406cf3373525dd09c293c
EFMTnkT7m-216.exe is malicious with respect to md5 hash
End of 0bf067750c7406cf3373525dd09c293c Searching Time: 02-07-2016_17:50:42
Difference of 0bf067750c7406cf3373525dd09c293c Time: 2.50339508057e-05 seconds
----------------------------------------------------------------------------------------------

Start of searching time for EFMTnkT7m-216.exe in Local Malware Database (LMD) is: 02-07-
2016_17:58:09
SHA-1 Digest for EFMTnkT7m-216.exe is 791f81ebcdfbefec87706e0f57a728cbc9e311e8 EFMTnkT7m-
EFMTnkT7m-216.exe is malicious with respect to SHA-1 hash
End of 791f81ebcdfbefec87706e0f57a728cbc9e311e8 Searching Time: 02-07-2016_17:58:09
Difference of 791f81ebcdfbefec87706e0f57a728cbc9e311e8 Time: 1.00135803223e-05 seconds
----------------------------------------------------------------------------------------------
Start of searching time for EFMTnkT7m-216.exe in Local Malware Database (LMD) is: 02-07-
2016_17:59:52
SHA-256 Digest for EFMTnkT7m-216.exe is
262058ea75f3ce8c666977449791bbff87096144de755e38e63872de744eae36
EFMTnkT7m-216.exe is malicious with respect to SHA-256 End of
262058ea75f3ce8c666977449791bbff87096144de755e38e63872de744eae36
Searching Time: 02-07-2016_17:59:52
Difference of 262058ea75f3ce8c666977449791bbff87096144de755e38e63872de744eae36 Time:
8.82148742676e-06 seconds
----------------------------------------------------------------------------------------------
```

Figure 5.5: Snapshot of OMD for the detection of known malware by cross-checking with LMSD

the hash digest for malware of different types malware and benign (i.e., reconstructed executables of all six test cases) is shown in Figure 5.3a. Similarly, the time taken by the OMD to detect the known malware by cross-comparing with LMSD (based on computed hashes) is shown in Figure 5.3b. Table 5.2 represents the results of

OMS for the detected 5 hidden processes that included 1 rootkit self-hidden process (`hxdef073.exe`), 2 explicitly hidden benign processes by the `Hacker Defender` rootkit (`chrome.exe` and `explore.exe`), and other 2 are self-hidden by the Trojan malware on the monitored VM. The Figure 5.4 represents the snapshot of execution of the `Kelihos dec.exe` malware (not hidden) detected by the OMS with a detection rate of 37/55. Table 5.3 represents the AMMDS classified 220 as dubious executables and cross-compared with both LMSD and OMS to detect any known malware. Finally, Figure 5.5 represents a snapshot of OMD detected the known malware named `EFMTnkT7m-216.exe` by cross-checking with LMSD.

5.3.5 Experimental Methods

The prime aim of the OFMC is to explore the accurate detection and classification of malicious executables that are semantically reconstructed as hidden and dubious executables on live Monitored VMs using machine learning techniques. It has been seen in many researches (Kolter and Maloof 2006; Sharif et al. 2009; Shabtai et al. 2012; Bai and Wang 2016; Watson et al. 2016; Valipour 2016) that the overall process of classifying unknown executables as benign or malware using machine learning techniques consists of two phases, training, and testing phase. In training phase, 60% of the benign and malware samples of the training set (i.e., Set-A) are used to prepare a training file. The first step in the training phase is to pre-process the training samples to derive the N-gram features using the approaches explained in Section 5.2.4. It has been seen in previous researches that N-gram feature of size 4 byte exhibits promising results (Kolter and Maloof 2006; Masud et al. 2008). Therefore, we have decided to perform feature construction using N-gram of size 4 bytes during the evaluation of our proposed approach. Since the constructed features size is quite large, it is impractical to use all the extracted features to prepare a training file required to train the classifier to attain the real-time detection of the malware. Therefore, only the crucial topmost features were selected on the basis of the rank assigned by the feature selection techniques (as discussed in Algorithm 5.2). For each of the features, two separate feature score (rank) are computed using two different feature selection techniques namely NGL CC and Odds Ratio. Based on the highest feature score, the length (L) of the topmost features 250, 500, and 750 are selected to verify which feature length achieves more accuracy. The symbol L represents the number of top-most features selected based on the highest feature score from each class separately used to generate FFV. Finally, a training file is built using the FFV with the N-gram files corresponding to the training samples. Lastly, the classifier is trained using the constructed training file.

Next, during the testing phase, executables excluded from the training set are used to construct a testing file. To evaluate the testing phase, binary files belong to Set-B are executed on a live Windows Monitored VM in different experimental scenarios for each type of malware and benign samples, and finally, those executables are semantically detected (VMI perspective) and forensically extracted at the VMM level are in dubious form (as discussed in Section 4.3.1). The reconstructed hidden and dubious executables are first parsed and then a representative feature vector is extracted as a training instance. Based on this feature vector, the classifier categorizes the testing file instances as either benign or malware in real-time at the hypervisor.

The overall malware detection rate of the OFMC is measured in the testing phase by following the K/N-fold cross-validation approach. Here, the independent dataset is randomly divided into N equal-sized subparts (samples). Out of these N subparts, a single subpart is retained as validation data, and the remaining N-1 subparts are used as training data. The cross-validation process is reiterated N times (i.e., N-folds) and the final results are presented as an average of all the folds. This approach helps in systematically evaluating the robustness of our OFMC to detect and classify unknown malware from extracted executables.

5.3.6 Evaluation Metrics

The classifier detection performance can be measured by computing the difference between the predicted class for a given input and the actual class that the input belongs to. For instance, if the test input data is of the benign class and the classifier predicts it as benign, then, it is a correct classification. To quantify the detection performance of the classifier, the 2×2 confusion matrix is used (shown in Table 5.4) as it provides all the possible outcomes of a prediction and has the form TP, TN, FP, and FN of the classifier. The detection of the classifier is considered as TP when a malware file is properly identified as a malware otherwise, it is treated as FN. Any

Table 5.4: Confusion matrix

Class	Predicted malware	Predicted benign
Malware	True Positive(TP)	False Negative(FN)
Benign	False Positive(FP)	True Negative(TN)

benign file classified as malware is treated as FP otherwise, it is measured as TN. Six performance metrics such as True Positive Rate (TPR), FPR, Precision, Recall, Accuracy, and F-measure were used in this work to measure the performance of the classifier as shown in equations below. Finally, the weighted average result of the performance metrics was considered.

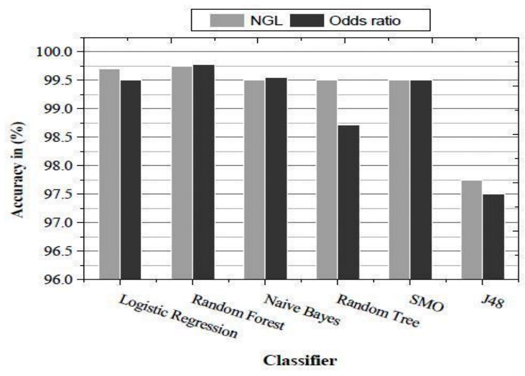
$$TPR = \frac{TP}{(TP + FN)} \qquad FPR = \frac{FP}{(FP + TN)}$$

$$Precision = \frac{TP}{(TP + FP)} \qquad Recall = \frac{TP}{(TP + FN)}$$

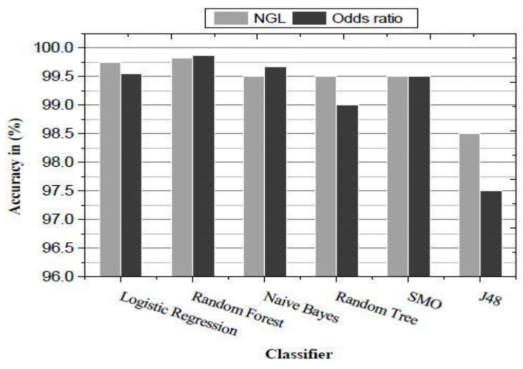
$$Accuracy = \frac{TP + TN}{(TP + TN + FP + FN)} \qquad F - Measure = 2 * \left(\frac{Precision * Recall}{Precision + Recall} \right)$$

5.3.7 Results Analysis

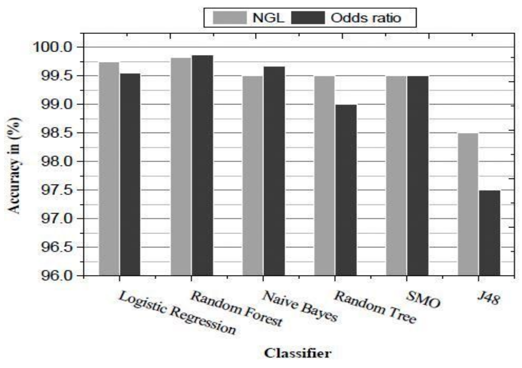
Six popular machine learning techniques such as such as Logistic Regression, Random Forest, Naives Bayes, Random Tree, and Sequential Minimal Optimization (SMO), and, J48 were used to measure the effectiveness of the proposed approach individually. All the chosen classifiers were initially trained with the default parameters available in the WEKA. Next, during the evaluation of each classifier, we selected three different FFV of length L=250, L=500, and L=750 from the training set to train the classifier. The evaluation performed by following the 10-fold cross-validation. It randomly splits the original input file into 10 equal subparts, where 9 subparts are used as the training dataset and the remaining 1 subpart is used as the validation data to measure the detection efficiency of the classifier. The cross-validation process is reiterated 10 times (the folds) for every combination with the condition that each subpart is used once as testing data. Finally, the outcome of each fold is averaged to estimate the overall efficiency of the classifier. The same steps are repeated separately for different FFVs of different sizes from two different feature selection techniques. This approach helps in systematically evaluate the feasibility of our proposed OFMC of the AMMDS



(a) L=250



(b) L=500



(c) L=750

Figure 5.6: Malware detection accuracy achieved by different classifiers based on NGL CC and Odds Ratio feature selection techniques for three different feature length

to measure detection and the classification accuracy of malware from semantically reconstructed dubious executables at the VMM.

In the initial experiments, we observed that the Random Forest classifier achieved the highest accuracy on all three different L for two feature selection techniques, as compared to the other classifiers. Figure 5.6 and Table 5.5 provides details of the malware detection accuracy and TPR and FPR achieved by different classifiers for 10-fold cross-validation evaluation. In particular, the Random Forest classifier achieved an accuracy of 99.75% with 0.003 FPR, 99.83% with 0.002 FPR, and 100% with 0 FPR for L=250, L=500 and L=750 features, respectively for suggested features of NGL feature selection technique. Similarly, the same classifier performs pretty well on the features that are suggested by Odds Ratio feature selection technique while yielding an accuracy of 99.78% with 0.002 FPR, 99.87% with 0.001 FPR and, 100% with 0 FPR for L= 250, L=500, and L=750, respectively.

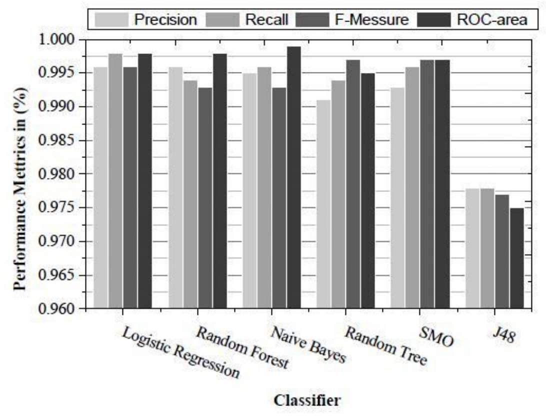
Table 5.5: TPR and FPR of different classifiers on different feature length

Feature Length		L= 250		L= 500		L= 750	
Classifier	Metrics	NGL CC	Odds Ratio	NGL CC	Odds Ratio	NGL CC	Odds Ratio
Logistic Regression	TPR	0.998	0.995	0.998	0.998	0.998	0.998
	FPR	0.002	0.005	0.002	0.002	0.002	0.002
Random Forest	TPR	0.998	0.997	0.998	0.998	1	1
	FPR	0.003	0.002	0.002	0.001	0	0
Naives Bayes	TPR	0.995	0.995	0.995	0.995	0.995	0.998
	FPR	0.005	0.005	0.005	0.005	0.005	0.002
Random Tree	TPR	0.995	0.988	0.995	0.99	0.995	0.993
	FPR	0.005	0.012	0.005	0.01	0.005	0.007
SMO	TPR	0.995	0.995	0.995	0.995	0.995	0.998
	FPR	0.005	0.005	0.005	0.005	0.005	0.002
J48	TPR	0.978	0.975	0.985	0.975	0.985	0.975
	FPR	0.022	0.025	0.015	0.025	0.015	0.025

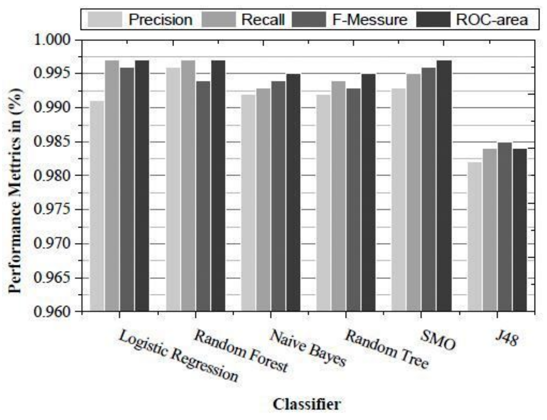
The main reason to achieve better accuracy by the Random Forest classifier is that, it uses multiple decision trees that are randomly chosen to vote for overall classification of the given input file, where each decision tree classifies the new instance of features with a majority of the vote (Oshiro et al. 2012). In this work, the Random Forest classifier achieved the highest accuracy under the default parameters (tree size, T=10), where, T represents the number of decision trees in the ensemble.

It can also be seen from Figure 5.6 is that the second highest accuracy yielded is by Logistic regression classifier ranging from 99.71% to 99.82% followed by Navies Bayes, Random Tree, SMO and J48 classifiers for L= 250, L= 500 and L=750 features of NGL based feature selection techniques. Similarly, for Odds Ratio based feature selection technique, Navies Bayes achieved second highest accuracy ranging from 99.55% to 99.75% followed by Logistic Regression, SMO, Random Tree, and J48 classifiers for L=250, L=500, and L=750, respectively.

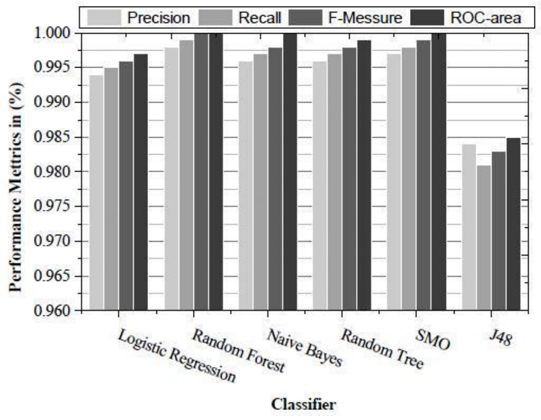
The performance of the each classifier was also evaluated using other performance metrics such as Precision, Recall, F-Measure, and Receiver Operating Curve (ROC) area separately for the both the feature section techniques. Figure 5.7 (a), (b), and (c) represents different performance metric results for three different feature lengths 250, 500, and 750, which are selected based on the highest feature score recommended by NGL CC selection technique. We notice that the maximum detection proficiency of the malware was achieved by Random Forest classifier for L=750 with 0.998, 0.999, 1, 1 of Precision, Recall, F-Measure and ROC respectively. Similarly, Figure 5.7



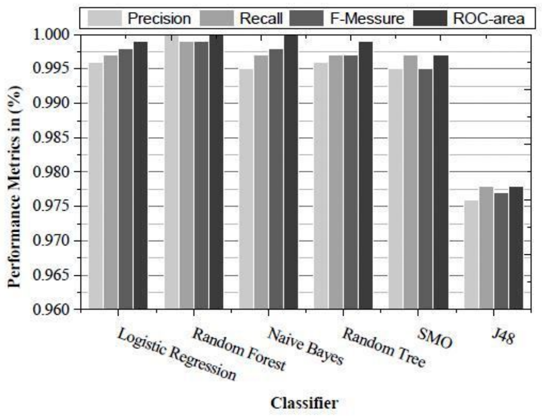
(a) L=250



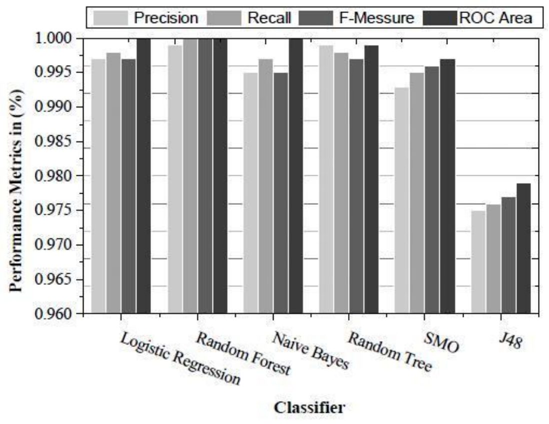
(b) L=500



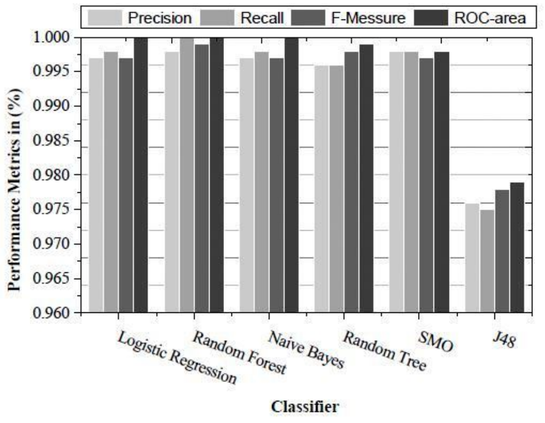
(c) L=750



(d) L=250



(e) L=500



(f) L=750

Figure 5.7: Comparison of performance of the classifier under different performance metrics for the different feature length recommended by NGL CC and Odds Ratio feature selection techniques

(d), (e), and (f) represents performance metric results of all classifier for L= 250, L=500, and L=750 as recommended by the Odds Ratio feature selection technique.

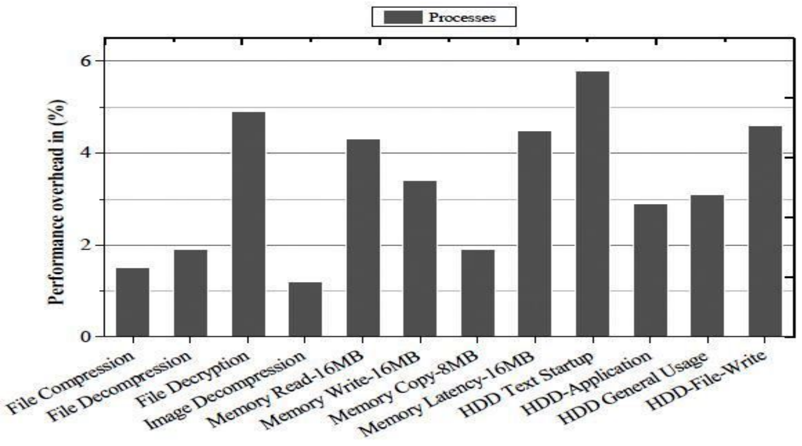


Figure 5.8: Performance overhead of the AMMDS on PCMark05 in detecting hidden and dubious state information of Monitored VM

It can also be notice that, for L=750 of Odds Ratio feature selection techniques the same Random Forest classifier yielded 0.998, 1, 0.999, 1 of Precision, Recall, F-Measure, and ROC, respectively. Furthermore, the malware detection performance of all other classifiers are depicted in Figure 5.7 with appropriate feature lengths of feature selection techniques. The J48 classifier shows lower accuracy for different feature length of both NGL CC and Odds Ratio feature selection techniques.

5.3.8 Performance Overhead

The performance overhead of the AMMDS is evaluated by following the series of test using PCMark Benchmark suite as discussed in the Section 4.5. Figure 5.8 represents the overall performance overhead caused by the AMMDS. Tests such as File Decryption, HDD-Text-startup, and HDD-File-Write induced maximum performance overheads of 4.9%, 5.8%, and 4.6%, while other test performance overheads were less than 4.5% on the Windows XP SP3 Monitored VM. These were noticed when the AMMDS abstracted the process semantic view during explicit detection of the hidden, dead, and dubious processes and pause and perform acquisition of the memory dump of the live Monitored VM.

5.4 Discussion

The current development of the proposed AMMDS included extended functionalities for detection and estimation of the symptoms of malware execution on the live Moni-

Table 5.6: Comparison of AMMDS with previous VMI-based malware detection approaches

Functionality	Lycoside (Jones et al. 2008)	Lare (Payne et al. 2008)	VMwatcher (Jiang et al. 2007)	Process-out-grafting (Srinivasan et al. 2011)	SYRINGE (Carbone et al. 2012)	AMMDS
Hidden process detection	√	□	√	□	□	√
Time synchronization	×	×	×	×	×	√
Incorporation of MFA	×	×	×	×	×	√
Malicious check on hidden and dubious process	×	×	×	×	×	√
Guest-assisted	×	√	×	√	√	√
Manual analysis	√	□	√	□	□	×
Fully automated	×	×	×	×	□	√
Machine learning techniques	×	×	×	×	×	√
Performance overhead	6%	Varied	□	Varied	Varied	5.8%

□: Information are not explicitly mentioned in the previous work

tored VM. In addition, the incorporation of machine learning techniques emphasized as the first scientific in-guest assisted VMI introspection technique to precisely detect and classify the running processes on the Monitored VM as benign or malicious at the VMM. The AMMDS performed this task from the semantically reconstructed executables that were introspected and forensically extracted at the VMM. The current demonstration of this approach is specific to the Windows guest OS to automatically detect the execution of large malware on the live Monitored VM by eliminating manual analysis.

The different categories of real world malware executables used in this work include self-hidden behavior malware, which hides on execution on the guest OS. In addition, we also used both user-mode and kernel-mode rootkits to explicitly hide some benign and malicious running processes to test the detection feasibility of our proposed AMMDS. We practically confirmed that VMI introspector of the AMMDS as being proficient in detecting hidden and malicious processes caused by stealthy malware and rootkits by traversing the semantic view of the `_EPROCESS` data structure of the live Monitored VMs.

5.4.1 Comparison with Existing Work

In order to highlight the significance of our proposed AMMDS, a comparison was undertaken in two phases. In the first phase, the extended functionality of the AMMDS was compared with other previous in-guest assisted and out-of-VM VMM-based malware detection techniques. Table 5.6 we can see that the AMMDS is able to detect and estimate the symptoms of malware with enhanced functionality, which was not addressed in previous approaches. Furthermore, functionalities such as the incorporation of MFA with VMI, malicious check on detected hidden and dubious process at the VMM, and fully automated and leveraging machine learning techniques for detection of known and unknown malware were not presented in any of the previous VMI-based relevant approaches.

In the second phase, the systematic evaluation of the proposed AMMDS was compared with other VMM-based introspection and non-introspection malware detection and classification approaches that used machine learning techniques. Table 5.7 summarizes the comparison of the AMMDS with other related works. To the best of our knowledge, except (Watson et al. 2016), none of the VMM-based malware detection approaches used machine learning techniques to detect malware. In particular, none of the VMI-based malware detection approaches used the machine learning techniques to detect and classify malware on semantically reconstructed high-level state information of the live introspected system from VMI perspective.

Authors (Watson et al. 2016) proposed VMM-based malware detection and classification system using one-class SVM machine learning technique. The vectorial (feature) representation of this approach not only considered the features related to the processes, but also the network activity of the introspected VM as statistical meta-features. In addition, this was not compared with any of the public benchmarked datasets to quantify the classification accuracy of their system. Maximum malware detection accuracy of this work was highlighted as more than 90%. Our proposed AMMDS achieved upto an accuracy of 100% with 0 FPR on the generated dataset at VMM-level.

Table 5.7: Comparison of results with other VMM-based and non-introspection based malware detection approaches that used machine learning techniques

Related work	Feature types	Approaches	Accuracy	FPR	VMM
Masud et al. 2008	N-gram	SVM, Boosted J48	96.88%	□	×
Shabtai et al. 2012	opcode	SVM, LR, RF, ANN, DT, BDT, NB, BNB	>96%	0.02	×
Bai and Wang 2016	Byte N-gram + opcode N-gram + Format feature	J48, RF, AdboostM1(J48), Bagging(J48)	99%	0%	×
Zhang et al. 2016	Multi-feature	XGBoost, ExtraTreeClassifier, GradientBoost	99.72%	□	×
Bai and Wang 2016	N-gram, opcode	J48, RF, AdboostM1(J48), Bagging(J48)	99%	0%	×
Huda et al. 2017	String feature	RF, SVM, J48, NB and IB	100%	0	×
Watson et al. 2016	Statistical meta-features	one-class SVM	>90%	□	√
Our proposed work	N-gram	LR, RF, NB, RT, SMO, J48	100%	0	√
Logistic Regression (LR), Random Forest (RF), Artificial Neural Networks (ANN), Decision Trees (DT), Boosted Decision Trees (BDT) Naive Bayes (NB), Boosted Naive Bayes (BNB), Random Tree (RT), □: indicates information not available in the previous work.					

To substantiate the effectiveness of the results, the proposed AMMDS was also compared with non-VMM or non-introspection based malware detection approaches. As a number of static and dynamic (non-introspection) based works are presented in the literature, we have chosen a relevant research with principal targets as feature extraction type and 96% or above accuracy and FPR. Masud et al. (2008) proposed hybrid feature selection technique to detect malicious executables. The construction of the hybrid feature set was based on the N-grams of the executables features, assembly instructions, and a dynamic link library. Overall, this approach achieved 96.88% accuracy.

Additionally, (Shabtai et al. 2012) used an opcode based feature extraction methods to detect unknown malware using several machine learning classifiers. Their approach was evaluated by considering a large number of benign and malware datasets and achieved greater than 96% of detection accuracy. Recently, in (Bai and Wang 2016) authors used Bytecode N-gram, opcode N-gram, and format features as multi-view features to detect unseen malware using three ensemble learning methods. They used two static malware datasets to validate the proposed classification methodologies with 0% false alarm rate.

Similarly, (Zhang et al. 2016) have proposed a lightweight malware classification system using ensemble tree-based XGBoost, ExtraTree, and GradientBoost as machine learning classification algorithms to detect new and real-world malware. They combined multiple categories of features that were extracted from malicious executables whereby the proposed approach would work even on obfuscated and packed malware of different families. The authors experimented their approach with Microsoft provided "Kaggle" malware challenge dataset and achieved 99.72% detection accuracy of malware.

In order to protect CPS against new malware variants recently, (Huda et al. 2017) proposed semi-supervised approach. They have used four supervised machine learning classifiers based on the static and dynamic (run-time) malware executables feature to evaluate classification methodologies. Finally, this approach yielded up to 100% accuracy with zero FPR as similar to our proposed approach. Some of the non-introspection based related work mentioned in Table 5.7 achieved equivalent accuracy than our proposed AMMDS. However, our approach is unique for the detection of unknown malware from a VMI perspective at the VMM with 100% detection accuracy.

5.5 Summary of the Work

In this work, we have presented the design, implementation, and evaluation of the AMMDS as an advanced VMI-based guest assisted out-of-VM security solution that leverages both VMI and MFA techniques to estimate symptoms of malware execution and also able to accurately detect unknown malware (malicious executables) running on the CPS-based Monitored VM. The OMD of the AMMDS is able to recognize known malware whereas the OFMC is capable of detecting and classifying unknown malware by using machine learning techniques. The proposed AMMDS extensively reduces the manual effort required to accurately identify the malware from the semantically reconstructed and forensically extracted executables as compared to other existing VMI and MFA based out-of-VM approaches. Finally, the AMMDS was evaluated against a large number of real-world Windows malware as well as benign executables to measure the malware detection rate. Our empirical results demonstrate that AMMDS is capable of recognizing malware with an accuracy of 100%. Further, the observed experimental results showed that the maximum performance overhead induced by the AMMDS is 5.8% under evaluation of the Windows benchmark suite.

Chapter 6

Leveraging Machine Learning Techniques to Detect and Characterize Unknown Malware at VMM

6.1 Introduction

In this approach, we have extended both A-IntExt system and AMMDS while addressing its limitations that are discussed in Section 1.9, research contribution 4. The prime aim of this work is to explore the accurate detection of both known and unknown (signature-less) malicious executables that are semantically reconstructed as hidden and dubious executables on live monitored VMs using machine learning techniques. It has been seen in many research works (Kolter and Maloof 2006) that the overall process of detecting unknown executables as benign or malware using machine learning techniques consists of two phases, the training phase and the testing phase. Our early proposed AMMDS performs the detection accuracy of malware based on the train-and-test splitting of the dataset at VMM. However, the approach of following the train-and-test evaluation has two limitations. First, the split of each class dataset samples used for the creation of the training model is not the same as the creation used for testing or prediction of the model leading to the deceptive error rate. Second, a model that is chosen for its accuracy on the preparation of training dataset as not same as its accuracy, or likely have lower accuracy on the independent test set (i.e., predicting unknown malware). These two problems are often referred to as `over-fitting` in machine learning (Domingos and Pazzani 1997). `Over-fitting` is a general problem that plagues many machine learning techniques. It occurs when a machine learning techniques report a low error on the training set and high error on the testing set. Also, when a model starts to "remember" the training data, instead of "learning" to generalize from the pattern. In order to combat over-fitting, cross-validation, and regularization, Bayesian priors techniques were widely used in previous research (Kearns et al. 1997).

In this work, the `over-fitting` issue was addressed by dividing the original dataset into three separate datasets such as the training, testing, and validation sets on both the generated and benchmarked datasets, respectively. In order to validate the malware detection proficiency of the A-IntExt system, we used two datasets: generated dataset (generated at VMM) and benchmarked datasets. Six popular machine learning techniques were used to measure the effectiveness of the proposed approach individually. To measure the generalization capacity of each classifier, we performed a comprehensive set of evaluations by following a common experimental procedure on both the generated and benchmarked datasets. The A-IntExt system was evaluated using the testing set and the validation set, and the obtained results were compared with the standard 10-fold cross-validation test results of the validation set.

Other additional and significant contribution of proposed A-IntExt are:

- We have implemented HF selection technique that uses representative instances of other individual feature selection techniques of the corresponding feature set that were extracted from the detected hidden and dubious executables of memory dumps of the introspected guest OSs.

- Our proposed A-IntExt system uses machine learning techniques from a VMI perspective at the VMM and performs runtime analysis on the introspected and forensically extracted executables of the Monitored VM to detect unknown malware. Further, it gathers the evidence related to identified malware. The A-IntExt system is not tailored to spot a particular malware rather than it is able to detect different types of malware.

- The robustness of the A-IntExt system was empirically evaluated by considering over 3750 different types of real-world malware and 4500 benign executables. Our proposed A-IntExt system achieved 99.55% accuracy and 0.004 FPR on the

10-fold cross-validation for the generated dataset at the VMM. In addition, the obtained results were compared with other benchmarked malware datasets that consisted of 3800 malware and 3930 benign executables. Further, the proposed A-IntExt system was compared with the existing malware detection approaches that use machine learning techniques.

6.2 System Design and Implementation

The working functionality and the overview of the A-IntExt system are described in Section 4.3 of chapter-3. In addition, the feature vector generator is on another extended component that A-IntExt (as part of this chapter contribution). Once Executable file extractor 5.2.2 of the A-IntExt system forensically reconstruct the executable files from infected memory dumps, the feature vector generator function as discussed below.

6.2.1 Feature Vector Generator

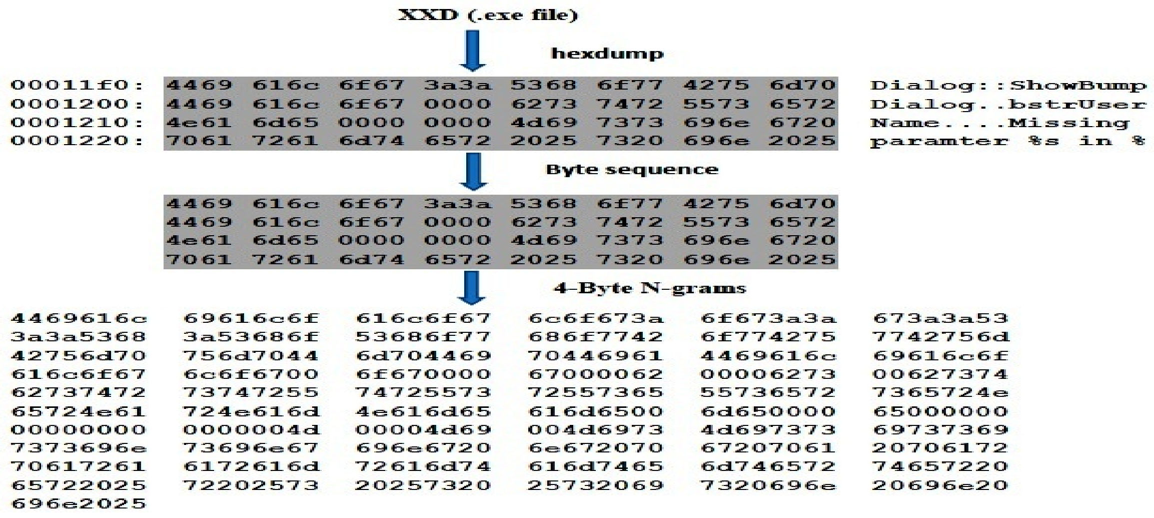
The prime function of the Feature Vector Generator (FVG) is to generate an attribute-relation file format during the training phase of the classifier. The FVG prepare a testing file by utilizing the feature extraction and feature selection techniques. First, it extracts the features (number of N-grams) from the reconstructed executables by using the feature extraction technique, and then uses the FFV and the extracted unique N-gram features of all the test cases to prepare a testing file.

In order to demonstrate which feature selection technique facilitates in improving classifier accuracy, the FVG generates four distinct testing files of predefined top features' length. Each testing file corresponds to different feature selection techniques such as IG, NF, CS, and HF.

Feature extraction method: In this work, the most popular feature extraction technique called N-gram (Abou-Assaleh et al. 2004) is used to extract the features from each of the executables (binary files) that have been forensically extracted from the infected memory dump of the Monitored VM. The extracted executables include semantically detected hidden and dubious processes by the A-IntExt system on the live monitored guest OS. Once all the executables are extracted at the VMM, these are provided as input to the UNIX hexdump utility of the Dom0 OS to acquire the 'hexdump' file for each of the extracted executables. The obtained hexdump is further processed to attain byte sequences that represent a snippet of the machine code of the executable. From each hexdump file, a byte sequence is extracted in terms of predefined N-gram size using the N-gram feature extraction technique. The procedure followed to obtain N-gram features of size 4 bytes (N-gram technique, where N = 4) from the executable is shown in Figure 6.1. Finally, these extracted byte sequences are grouped in the form of N-grams, which represent contiguous bytes sequences, where N represents the number of fixed bytes. Each individual N-gram is considered as a feature and all N-grams are treated as an original feature vector.

Many research works have demonstrated that N-gram based methods show great promise in detecting unknown malware (Abou-Assaleh et al. 2004; Kolter and Maloof 2006; Reddy et al. 2006; Reddy and Pujari 2006; Perdisci et al. 2008; Bai and Wang 2016; Raff et al. 2016; Ahmadi et al. 2016). However, the N-gram-based technique produces many N-grams that include duplicates, and all of them cannot be used as final features to generate a training file as well as a testing file. This results in substantial memory consumption during the training phase and also leads to performance degradation at runtime detection of malware. Furthermore, it also encourages more FPR, while decreasing the accuracy of the classifiers. This issue can be addressed by using feature selection techniques.

Feature selection techniques : In order to detect unknown malware using



```
00011f0:  4469 616c 6f67 3a3a 5368 6f77 4275 6d70   Dialog::ShowBump
0001200:  4469 616c 6f67 0000 6273 7472 5573 6572   Dialog..bstrUser
0001210:  4e61 6d65 0000 0000 4d69 7373 696e 6720   Name....Missing
0001220:  7061 7261 6d74 6572 2025 7320 696e 2025   paramter %s in %
```

```
4469 616c 6f67 3a3a 5368 6f77 4275 6d70
4469 616c 6f67 0000 6273 7472 5573 6572
4e61 6d65 0000 0000 4d69 7373 696e 6720
7061 7261 6d74 6572 2025 7320 696e 2025
```

```
4469616c   69616c6f   616c6f67   6c6f673a   6f673a3a   673a3a53
3a3a5368   3a53686f   53686f77   686f7742   6f774275   7742756d
42756d70   756d7044   6d704469   70446961   4469616c   69616c6f
616c6f67   6c6f6700   6f670000   67000062   00006273   00627374
62737472   73747255   74725573   72557365   55736572   7365724e
65724e61   724e616d   4e616d65   616d6500   6d650000   65000000
00000000   0000004d   00004d69   004d6973   4d697373   69737369
7373696e   73696e67   696e6720   6e672070   67207061   20706172
70617261   6172616d   72616d74   616d7465   6d746572   74657220
65722025   72202573   20257320   25732069   7320696e   20696e20
696e2025
```

Figure 6.1: 4 byte N-gram sequence extraction from an executable

machine learning techniques, the feature selection technique plays a crucial step in detecting unknown malware from a number of other benign programs (Blum and Langley 1997; Dash and Liu 1997; Shabtai et al. 2009). Feature selection techniques operate according to filter-based approach (Mitchellet al. 1997), in which a measure is calculated to quantify the correlation of each feature (i.e., presence) with the class (benign and malware) to predict its anticipated results of the classification techniques. In our work, the A-IntExt system extracted a number of N-gram features from the executables (i.e., forensically extracted from infected memory dump of the Monitored VMs) at VMM. Typically, many of the N-gram features generated for each executable are not important for machine learning techniques to perform detection of malware. To address these issues, the feature selection step helps to identify an optimal set of features while eliminating other irrelevant and redundant features from the original feature vector, because redundant features reduce the predictive performance of the classifier (Kwak and Choi 2002). Since feature selection methods facilitate to reduce the dimensionality of the original feature vector, it allows the machine learning techniques to function more efficiently and rapidly on the given input set of features in the training phase. Additionally, selecting the most relevant features from the massive number of generated N-gram features on large-scale datasets increases accurate results from the classification techniques (Langleyet al. 1994).

In this work, the A-IntExt system is implemented by considering popular statistical-based feature selection techniques such as IG, NF, CS, and HF. Each feature obtains a score from individual and HF selection techniques and selects the topmost (sorted in descending order based on the score) and recommends the best feature to the classification techniques. Finally, the performance of various machine learning techniques is compared based on the predefined top feature length of the individual feature selection technique. The following sections provide the details of each feature selection technique.

- **Information gain**: The IG (Yang and Pedersen 1997; Quinlan 1986) is a commonly used feature selection approach in machine learning for the detection of malware. For a given i^{th} N-gram (Ng_i), the IG score is calculated using equation below:

$$IG(Ng_i, C_k) = \sum_{Ng_i=0}^{1} \sum_{C_k \in (B,M)} P(Ng_i, C_k) \log \frac{P(Ng_i, C_k)}{P(Ng_i)P(C_k)} \tag{6.1}$$

The symbol C_k takes one of the two classes, benign or malware, i.e., K $\in$ {Benign (B), Malware (M)} and Ng_i takes a value either 1 or 0 based on its presence or absence in the N-gram file; P(Ng_i, C_K) is a proportion of N-gram files in which Ng_i takes a value to the total number of N-gram files in the C_k; P(Ng_i) is the ratio of the number of N-gram files in the dataset in which Ng_i takes a value to the total number of N-gram files in the dataset, and C_k is the ratio of the class to the total number of classes.

- **N-gram frequency:** The NF (Yang and Pedersen 1997; Reddy and Pujari 2006) provides an integer count that demonstrates the given Ng_i is present in how many N-gram files in a certain class C_K. Class-wise NF score for Ng_i is computed using Equation (6.2).

$$FNg_i, C_K = \sum_{j=1}^{N} Ng_{i,j}, C_K \tag{6.2}$$

$$Ng_{i,j}, C_K = \begin{cases} 1, & \text{if } Ng_i \in f_j \\ 0, & \text{otherwise} \end{cases}$$

$$Ng_i, C_k = \begin{cases} Select, & \text{if } FNg_i, C_k \geq \delta_k \\ \text{Ignore}, & \text{otherwise} \end{cases}$$

The term C_K represents the K^{th} class and K $\in$ {benign, malware}, $Ng_{i,j}$ denotes i^{th} Ng_i of j^{th} N-gram file (f_j), and N designates the number of N-gram files in class C_K. If the value of FNg_i, C_k is greater than or equal to the predefined threshold δ_k, then it considers Ng_i as the best feature.

- **Chi-Square ($\tilde{\chi}^2$):** The CS test (Yang and Pedersen 1997) is one of the popular feature selection techniques, used to measure the lack of dependence of two variables such as feature Ng_i and class C_k. Higher CS score represents that the variables have a close relation. The Chi-Square formula is as follows:

$$\tilde{\chi}^2(Ng_i, C_k) = \frac{N[PS - QR]^2}{(P+R)(Q+S)(P+Q)(R+S)} \tag{6.3}$$

Where, N is the total number of N-gram files in the training set, P is the number of N-gram files present in the C_K containing Ng_i, R is the number of N-gram files not in the C_K containing Ng_i, Q is the number of N-grams files in the C_k not containing Ng_i, and S is the number of N-gram files not in the C_k not containing Ng_i.

- **Hybrid feature selection:** In order to select the best N-gram features, our proposed HF selection technique follows the ensemble strategy (Rokach et al. 2007), which combines features that are recommended by other feature selection techniques. The HF selection technique computes the score for the i^{th} Ng_i using the equation (6.4) (Moskovitch et al. 2008):

$$HF(Ng_i) = \sum_{j=1}^{3} score_j(Ng_i) \tag{6.4}$$

The $score_j(Ng_i)$ denotes the Ng_i score given by the j^{th} feature selection technique. The procedure followed by the HF selection technique to generate FFV is described in Algorithm 6.1. It computes the score for each N-gram as per Equation (6.4) and selects the topmost N-grams on the basis of the highest score. This computation $(BFV_T - MFV_T)$ helps to derive unique features that represent the benign class. Similarly, $(MFV_T - BFV_T)$ provides unique features for the malware class. The union of $(BFV_T - MFV_T) \cup (MFV_T - BFV_T)$ provides a unique FFV that represents unique benign class features and unique malware class features.

Algorithm 6.1: Hybrid feature selection

```
Input  : Dataset ($\Delta$) contains malware and benign N-gram files ($f_i$)
Output: FFV
1  begin
2      Create empty files BFV, $BFV_T$, BNF, MNF, MFV, $MFV_T$
3      foreach $f_i \in \Delta$ do
4          if ($f_i \in$ Benign class) then
5              BNF $\longleftarrow BNF \cup f_i$
6          else
7              MNF $\longleftarrow MNF \cup f_i$
8          end
9      end
10     Remove duplicate N-gram from BNF and MNF separately
11     foreach N-gram $Ng_i \in BNF$ do
12         compute HF score as per Equation (6.4)
13         store N-gram and its HF score into a file.
14           BFV $\longleftarrow BFV \cup$ {N-gram, HF score}
15     end
16     sort BFV in DO based on the HF score
17     select topmost 'L' number of N-grams
18     for $j$=1 to $L$ do
19         $BFV_T \longleftarrow BFV_T \cup BFV\ (N - gram_j)$
20     end
21     foreach N-gram $Ng_i \in MNF$ do
22         compute HF score as per Equation (6.4)
23         store N-gram and its HF score into a file
24           MFV $\longleftarrow MFV \cup$ {N-gram, HF score}
25     end
26     sort MFV in DO based on the HF score
27     select topmost 'L' number of N-grams from MFV
28     for $j$=1 to $L$ do
29         $MFV_T \longleftarrow MFV_T \cup MFV\ (N - gram_j)$
30     end
31     Compute FFV = $\{[BFV_T - MFV_T] \cup [MFV_T - BFV_T]\}$
32 end
```

6.3 Experiments and Datasets

6.3.1 Datasets and Dataset Collection

In order to validate the malware detection proficiency of the A-IntExt system, we used two datasets: Generated dataset and Benchmarked datasets. The creation of generated dataset consists of a collection and in-execution of real-world known and unknown malware and benign executables on two Monitored VMs semantically reconstructed in the form of executables at the VMM. The benchmarked dataset consists of both static benign and malware executables. Both the generated and benchmarked datasets are divided into training, testing, and validation sets in 60%: 20%: 20% ratio similar to previous approaches (Rieck et al. 2008; Ozsoy et al. 2016). We have utilized an imbalanced training set that comprises of more numbers of benign and fewer numbers of malware executables.

Table 6.1: Types of malware used in the experiments

#	Types of malware	Quantity	%	Min. size	Max. size
1	Backdoor	700	19%	10KB	0795KB
2	Zeus botnet	690	19%	15KB	0855KB
3	Ransomware	715	20%	08KB	1038KB
4	Trojan horse	700	19%	12KB	0565KB
5	Spyware	695	19%	12KB	1125KB
6	Polymorphic & Metamorphic	250	4%	08KB	0856KB
Total # malware used		**3750**	100%		

Generated dataset: In this section, we explain the collection of real-world malware and benign executables that were used and dynamically executed onto the Monitored VMs in different experimental test cases in order create a generated dataset at the VMM. Over 3625 different Windows malware executables (samples) were collected from the VX Heaven[1] malware repository by directly connecting the Dom0 OS to an external network. During the collection of these samples, we disabled the security mechanisms of the Dom0 OS when it was connected to the network of the malware repository. Additionally, we used two families (Win32 Mydoom and Beagle) of 125 viruses (obtained from http://vxheaven.org) in the next generation virus creation kit (version 0.45 Beta obtained from http://vxheaven.org) to generate some non-signature based unknown malware. We generated 125 additional unknown malware. For each generating virus, we applied junk code insertion, encryption, and obfuscation techniques by following the procedure described in a previous work (Sung et al. 2004). Once all the 125 unknown (signature-less) malicious executables were generated, we examined these executables (.exe) on the VirusTotal[2] website, where each generated

[1]http://vxheaven.org/, last accessed in March 2017.
[2]https://www.virustotal.com/, last accessed in March 2017.

Table 6.2: The composition of generated dataset

Generated dataset	Split %	# Malware	# Benign
Training set	60%	2250	2700
Testing set	20%	750	900
Validation set	20%	750	900
Total # samples	100%	**3750**	**4500**

Table 6.3: The composition of benchmarked datasets

Benchmarked dataset	Split %	# Malware	# Benign
Training set	60%	2280	2358
Testing set	20%	760	786
Validation set	20%	760	786
Total # samples	100%	**3800**	**3930**

malware sample was checked with 59 anti-virus engines (up-to-date). From the VirusTotal generated report of each sample, we confirmed that none of these samples were malicious. But in reality, these were unknown malware with no signature that was generated by us. Finally, total 250 (i.e., 125 obtained and 125 generated) polymorphic and metamorphic malware were used. Table 6.1 illustrates the constitution of 3750 different categories of obtained and generated real-world malware used in this work.

In addition, 4500 benign executables were collected from various sources, including the freshly installed various versions of the Windows VMs, popular software downloader website,[3] and other data sources.[4][5] These benign executables included the Windows native software (notepad++, paint, etc.), system software (Adobe, OpenOffice, etc.), browser extension (Newz tools, WebReader, etc.), multimedia tools (video player, music, etc.), games, popular browsers, file utilities, management software, file archiver, image and video editors, etc. To ensure that the downloaded benign executables did not contain any malicious code, we checked with VirusTotal. Table 6.2 illustrates the composition of the generated dataset.

Benchmarked datasets: The main reason for including benchmarked datasets was to confirm and validate how well our proposed A-IntExt system worked on other tested benchmarked malware datasets not generated by us. This strengthens the

[3]http://software.informer.com/software/, last accessed in March 2017.
[4]http://download.cnet.com/windows/, last accessed in March 2017.
[5]https://sourceforge.net/, last accessed in March 2017.

trustworthiness of our proposed approach, which was tested on a public benchmarked malware datasets. We collected 3800 recent malware samples from another public malware repository[6] that included some common and different families of malware (Backdoors, Worms, Exploit, Torjan.Zbot-1433, Torjan-Zbot-1023, Torjan.Kelihos-5, and Trojan.Kelihos-4, etc.). For the benign executables, we gathered 3930 legitimate benign executables from a clean installation of the various versions of Windows guest OSs and another software downloader website (http://sourceforge.net). The obtained benign executables included system software, games, application software, multimedia utilities, downloaders, etc. Further, these were confirmed via VirusTotal malware classification interface, to identify the different types of malware and benign executables. The composition of the benign and malware samples of the benchmarked datasets is shown in Table 6.3. About 60% of the samples were grouped into the training set, which consisted of 2280 malware and 2358 benign samples. The remaining 40% of the samples were equally partitioned into testing set (20%) and validation set (20%).

6.4 Evaluation

In this section, we discuss the procedure for generating a realistic dataset by using a large number of benign and malicious executables by considering the real-world scenario. We perform in-execution of real-world known and unknown malware on both the Windows XP SP3 and Windows 7 guest OSs in different experimental scenarios, while measuring the detection proficiency of the A-IntExt system. Finally, the semantic view of these detected processes is forensically reconstructed as executables from the infected memory dump of both the guest OSs. In order to prepare the training, testing, and validation files for the classifier, the total collection of malware and benign executables were divided into three parts: training, testing, and validation sets (as shown in Table 6.2). The training set consisted of 60% of the samples and these were used in the training phase (before being executed on the Monitored VMs) to train the classifier. The remaining 40% of the samples were equally divided into the testing and validation sets as 20% of each set. This grouping of executables was done to ensure that there was no overlapping of samples in the sets. In the testing set, 20%

[6]http://www.offensivecomputing.net/last accessed in March 2017.

of these malware and benign samples excluded from the training set were divided into three groups (i.e., I-III) that consisted of 250 (malware) and 300 (benign) samples in each group. Similarly, in the validation set, remaining 20% of the benign and malware executables excluded from the training and testing sets were also equally divided into three groups (i.e., IV-VI) and each group comprised of 250 malware and 300 benign samples.

Later, the first three groups (i.e., I-III) of the testing set malware and benign executables were invoked by a program on both the Windows guest OSs with different experimental scenarios as shown in Table 6.4. Meanwhile, simultaneously installed the user-mode and kernel-mode rootkits on the Windows XP SP3 32-bit Monitored VM. Through the injected rootkits, we explicitly performed hidden activity of benign and malicious running process on the guest OS.

Later, these executables with different test cases were introspected and forensically extracted in the form of dubious executables (as discussed in 4.3.1), including detected hidden processes at the VMM as shown in the prediction phase of Figure 6.3. Finally, for all the three different experimental tests, the A-IntExt system was successfully detected and semantically reconstructed over 1705 hidden and dubious executables (.exe) as a testing generated dataset from three infected memory dumps of the two guest OSs at VMM as shown in Table 6.4.

Similar to the testing set, the next three groups (i.e., IV-VI) of the validation set malware and benign executables was invoked by a program on freshly installed different Windows guest OSs as shown in Table 6.5. For example, in the IV test, we executed 250 Trojan horse and 300 benign executables, i.e., a total of 550 executables on a clean live Windows XP SP3 32-bit guest OS. Once all the executables were injected, some Trojan horse executables hid or disappeared on the Windows XP SP3 guest OS. At the same time, we also injected a `Hacker Defender` user-mode rootkit to explicitly hide one benign (`explore.exe`) and one malware (`Trojan.Win32.exe`) process running on the Monitored VM. These experiments lasted 4 minutes. A periodic introspection of the A-IntExt system helped in identifying the symptoms of malware execution by recognizing the disparity of the processes that emerged between the internal and external views of the processes state information of the Monitored VM (see Equation 5.3).

Table 6.4: Execution of benign and malicious executables on live Monitored VMs in different experimental test cases for generating testing dataset at VMM

Test #.	Malware types	Guest OS	# Executables file used			# PS visible by internal view	# PS introspected from external view	# PS detected by the A-IntExt system			
			Malware	Benign	Total			Hidden	Dead	Dubious	Time (in seconds)
I	Backdoor	Windows XP SP3	250	300	550	564	568	7	3	561	2.45
II	Zeus botnet	Windows XP SP3	250	300	550	560	568	11	3	557	2.25
III	Ransomware	Windows 7	250	300	550	563	569	8	2	561	2.55
Total # of executables			**750**	**900**				**26**		**1679**	

Table 6.5: Execution of benign and malicious executables on live Monitored VMs in different experimental test cases for generating validation dataset at VMM

Test #.	Malware types	Guest OS	# Executables file used			# PS visible by internal view	# PS introspected from external view	# PS detected by the A-IntExt system			
			Malware	Benign	Total			Hidden	Dead	Dubious	Time (in seconds)
IV	Trojan horse	Windows XP SP3	250	300	550	562	568	9	3	559	2.81
V	Spyware	Windows 7	250	300	550	561	568	10	3	558	2.62
VI	Polymorphic & Metamorphic	Windows 7	250	300	550	560	567	11	4	556	2.75
Total # of executables			**750**	**900**				**30**		**1673**	

As seen in test IV of Table 6.5, the A-IntExt system ascertained and counted these introspected processes, namely, internal, external, hidden, dead, and dubious processes as 562, 568, 9, 3, and 559, respectively. However, there were variations in the detected processes count on the internally captured and externally introspected process state information including 21 (additional) running default processes of the clean Windows XP SP3 32-bit guest OS. More specifically, the internal view of the total process visible is 562, i.e., from the 550 total injected malicious and benign executables, 9 (7 self-hidden by the Trojan horse and 2 were explicitly hidden by the `Hacker Defender` rootkit) processes are hidden at the guest OS, and 21 are native running processes of the guest OS. Finally, 562 are internally gathered processes. Note that these 9 hidden processes were not gathered by the GAM (i.e., internal-view of guest OS) as it does not appear in the `tasklist` command of the GAM (see Section 4.3.2). Finally, the VMI introspector of the A-IntExt system externally (i.e., VMM level) introspected and ascertained these 9 hidden processes.

Similarly, in the experiment V, we injected 250 spyware and 300 benign files, totaling 550 executables on the clean Windows XP SP3 32-bit guest OS. Meanwhile, in this test we injected DKOM based kernel-mode, `FU Rootkit` to explicitly hide a few benign and malware running processes by directly removing or unlinking it from the `_EPROCESS` data structure of the process list at the kernel-mode. Finally, the disparity of the processes was identified by the ICVA of the A-IntExt system similar to test IV. Likewise, experiment VI was conducted by executing other types of malware and benign executables on both the guests' OS. Finally, from three different experimental test cases, the A-IntExt system reconstructed over 1703 hidden and dubious executables as validation generated dataset from another three infected memory dumps of two guests' OS at the VMM as shown in Table 6.5.

From the experimental analysis results of both testing and validation sets, we observed that the A-IntExt system was proficient in estimating the abnormality of the introspected system by detecting the hidden and disguised processes' activity on the execution of malware. The time taken by the A-IntExt system to detect and categorize the hidden, dead, and dubious processes for each experimental test is shown in the last column of both Table 6.4 and Table 6.5. However, at this stage, it is not feasible for the A-IntExt system to check the detected processes benign, known

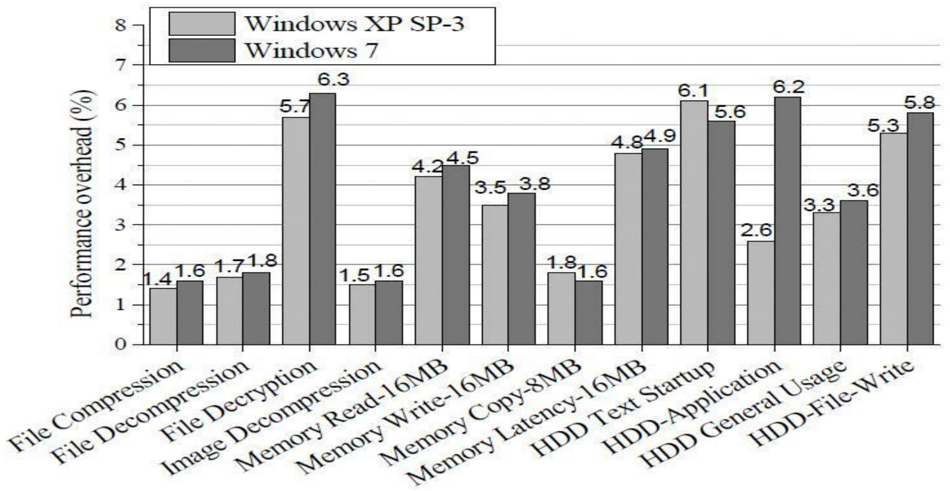


Figure 6.2: Performance overhead of the A-IntExt system using PCMark05 benchmark

or unknown malware. This issue was tackled by using machine learning techniques based on the features vector.

6.4.1 Performance Overhead

In order to measure the performance overhead induced by the A-IntExt system, a series of tests were performed by running the PCMark05 industry standard benchmark suite on both, Windows XP SP3 and Windows 7 Monitored VMs. Figure 6.2 demonstrates the overall performance overhead incurred by the A-IntExt system for both the Monitored VMs during the experimental analysis of this work.

Tests such as the HDD-Text Startup, File Decryption, and HDD-File-Write showed maximum performance overheads of 6.1%, 5.7%, and 5.3%, respectively, while other tests performance overheads observed was less than or equal to 4.8% on the Windows XP SP3 guest OS. These were noticed when the A-IntExt system abstracted the process semantic view during explicit detection of the hidden, dead, and dubious processes, and the pause and perform acquisition of the memory dump of the live Monitored VM. The same tests were executed while performing a similar operation for the Windows 7 guest OS. The HDD-File-Write, HDD-Application, and File decryption reported maximum performance overheads of 5.8%, 6.2%, and 6.3% respectively, and the other tests overhead were less than or equal to 5.6%. Finally, the A-IntExt system introduced maximum performance overhead of 6.3%, for introspection of the Windows 7 guest OS.

6.4.2 Experimental Methods

The prime aim of this work is to explore the accurate detection of both known and unknown (signature-less) malicious executables which are semantically reconstructed as hidden and dubious executables from infected memory dump of live Monitored VMs. It has been seen in many research works (Kolter and Maloof 2006; Sharif et al. 2009; Shabtai et al. 2012; Bai and Wang 2016; Watson et al. 2016) that the overall process of detecting unknown executables as benign or malware using machine learning techniques consists of two phases, the training and the testing phase. However, the approach of following the train-and-test evaluation has two limitations. First, the split of each class dataset samples used for the creation of the training model is not the same as the creation used for testing or prediction of the model leading to the deceptive error rate. Second, a model that is chosen for its accuracy on the preparation of training dataset as not same as its accuracy, or likely have lower accuracy on the independent test set (i.e., predicting unknown malware). These two problems are often referred to as over-fitting in machine learning (Domingos and Pazzani 1997). `Over-fitting` is a general problem that plagues many machine learning techniques. It occurs when a machine learning technique reports a low error on the training set and high error on the testing set. Also, when a model starts to "remember" the training data, instead of "learning" to generalize from the pattern. In order to combat `over-fitting`, cross-validation, and regularization, Bayesian priors techniques were widely used in previous research (Kearns et al. 1997).

In this work, the over-fitting issue was addressed by dividing the original dataset into three separate datasets such as the training, testing, and validation sets on both the generated and benchmarked datasets as discussed in Sections 6.3.1, and 6.4 respectively. In the training phase, a sufficient number of imbalanced benign and malware executables of the training set are used. To prepare a training file, which is used to train the classifier, executables of the training set are parsed to extract the relevant features.

Next, during the testing phase, executables excluded from the training set are used to construct a testing file. These are first parsed and a representative vector is extracted as a training instance. Based on this vector, the classifier categorizes

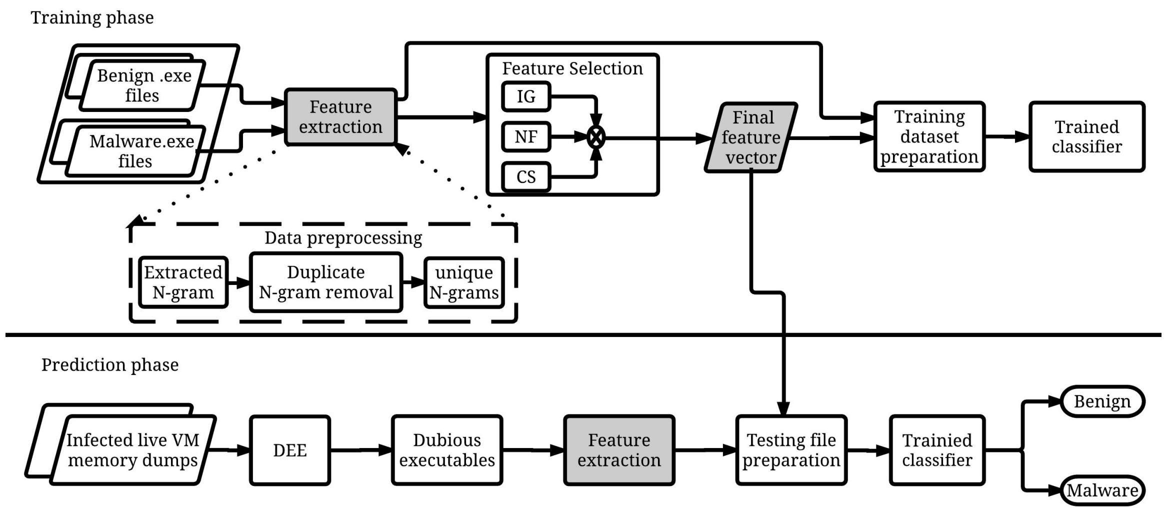


Figure 6.3: Training phase and prediction phase of the A-IntExt system

the testing file as either benign or malware. The purpose of the testing phase is to measure the performance of the chosen machine learning classifier in terms of standard accuracy on unknown malware.

Finally, in the validation phase, the performance of the trained classifier is validated using a new collection of files. This is because the training process guarantees that the accuracy of the classifier for the training data is generally high and the classifier is specifically well-matched to the files of the training set. In order to measure a more accurate estimation of how the trained classifier would categorize the unseen data, a validation set is used. The classifier measures the discrepancy between the authentic observed class and the predicted class of the observation to figure out the error in prediction and this is used to quantify the overall accuracy of the trained classifier.

In this work, in order to have a fair comparison and evaluation of both the testing and validation datasets, the evaluation results of an independent testing set were presented along with the accuracy and FPR metrics. The validation set included complete results presented based on the 10-fold cross-validation (Ng 1997) evaluation by considering different performance metrics. This facilitated in showing the actual performance predicted by the chosen machine learning classifiers for our proposed feature selection methodologies of the A-IntExt system.

The overall evaluation of our proposed approach on both the generated and benchmarked datasets are as follows:

Training model: The first step in the training phase is to pre-process the training samples to derive the N-gram features using the approaches explained in Sections 6.2.1. It has been seen in previous research work that N-gram features of size 4 byte exhibits promising results (Kolter and Maloof 2006; Masud et al. 2008). Therefore, we have decided to perform feature construction using N-gram of size 4 byte during the evaluation of our proposed approach. The total number of distinct 4-byte N-gram features constructed were 203,485,680 and 198,691,840 for the generated and benchmarked datasets, respectively, for both benign and malware class. Since the constructed features size is quite large, it is impractical to use all the extracted features to prepare a training file. Therefore, only the crucial topmost features were selected on the basis of the score assigned by the feature selection techniques (as discussed in Section 6.2.1). The FVG component of the A-IntExt system constructs the FFV that consists of the uppermost features recommended by the HF selection technique (Algorithm 6.1) described in Section 6.2.1(Feature Vector Generator). In other words, the top features endorsed by three different feature selection techniques, namely, the IG, NF, and CS are used to generate an FFV of HF selection technique. To demonstrate the efficiency of the HF selection technique, the FVG also constructs three other feature sets and each one consists of features suggested by individual feature selection technique such as the IG, NF, and CS. Finally, a training file is built using the FFV with the N-gram files corresponding to the training samples. Lastly, the classifier is trained using the constructed training file.

The experiments are repeated for different feature lengths (L) such as 100, 200, and 300, where, L represents the number of topmost features selected based on the highest feature score from each class separately. The obtained results corresponding to the HF selection technique is compared with individual feature selection techniques such as the IG, NF, and CS. Finally, the results are shown separately for each feature selection technique and HF selection technique for the recommended feature length of 300 features (which achieved the highest accuracy) on both the generated and benchmarked datasets.

Testing model: In the testing phase, the partitioned executables of the testing set as shown in Table 6.2 and discussed in Section 6.3.1 and Section 6.2.1 are executed one at a time in real-time onto the Monitored VMs in different experimental test scenarios as shown in Table 6.4. The A-IntExt system performs the initial screening using the ICVA, and then, reconstructs the executables corresponding to the detected hidden and dubious processes from the infected guest OSs. Table 6.4 depicts the total number of the semantically reconstructed hidden and dubious executables. Further, each of these reconstructed executables of the generated testing set undergoes pre-processing steps similar to the training phase as shown in the prediction phases of Figure 6.3. The steps used to generate a testing file from the reconstructed executables of the generated dataset is discussed in Section 6.2.1.

Similarly, on the benchmarked datasets, the testing file was prepared by considering the testing set (see Section 6.3.1). Finally, the generated testing file was sent to the trained classifier to verify whether the reconstructed executable file is benign or malware. The classifier examines the testing file and declares it as malware when it ascertains the presence of maliciousness, otherwise, it declares it as benign.

Validation model: In the validation phase, similar to the testing model, the partitioned executables of the validation set as shown in Table 6.2 were used to perform in-execution on the Monitored VMs, and later semantically reconstructed as validation generated set executables as shown in Table 6.5. Similarly, for benchmarked datasets, the partitioned executables of the testing set (shown in Table 6.3) were considered to generate validation test file. The evaluation of validation partitioned sets of both the generated and benchmarked datasets generally follows the K/N-fold cross-validation approach. Here, the independent dataset is randomly divided into N equal-sized subparts. Out of these N subparts, a single subpart is retained as validation data, and the remaining N-1 subparts are used as training data. The cross-validation process is reiterated N times (i.e., N-folds) and the final results are presented as an average of all the folds. The main aim of this N-fold cross-validation approach is to tackle the `over-fitting` issue during the evaluation of our proposed approach on both the generated and benchmarked datasets.

6.4.3 Evaluation Metrics

The list of evaluation matrices used in this approach were discussed in section 5.3.6. the In addition, the Matthew Correlation Coefficient (MCC) (Matthews 1975) Equation 6.5, which is a combination of Accuracy and FPR, takes the data imbalance into account of the generated and benchmarked datasets by using the amount of classified and misclassified files.

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{6.5}$$

6.4.4 Machine Learning Techniques

In this work, six different widely used supervised machine learning techniques (implemented in WEKA) were considered. These included Naive Bayes, SVM or Sequential Minimal Optimization (SMO), Simple Logistic, Random Forest, Random Tree, and J48 as they belong to the most popular categories, Bayes, Function, and Trees. The response of the classifier represents the detection accuracy of malware on both the generated and benchmarked datasets. The working procedure of each classifier is described below in brief:

Navie Bayes (Domingos and Pazzani 1997) is a well-known probabilistic method. Its classification is based on the Bayes' theorem with the assumption that all features in the training set are independent of each other. It classifies by calculating the maximum probability of attributes, which belong to a specific class.

SVM (SMO) (Platt 1999) technique is implemented (Witten and Frank 2005) as a support vector machine in the WEKA as a fast and efficient classifier to solve huge quadratic programming problems. It achieves high detection accuracy, while minimizing true error rate. During classification, it computes the probability of each class by maximizing the width of the margin. This maximization is calculated by dividing the problem into a series of sub-problems. Each possible sub-problem consists of two multipliers that are used to maximize and minimize the solution. This classifier uses the normalized polynomial kernel to map the input data. The SMO is a proficient classifier in the function category used to classify malware and benign executables based on the input FFV.

Simple Logistic (Han et al. 2011) is an ensemble-based classifier, which uses the LogitBoost (Witten and Frank 2005) algorithm to perform additive logistic regression. The main advantage of Simple Logistic is that it has a built-in feature selection function, when cross-validated, which leads to automatic attribute selection. It takes more time to build for fast classifications.

Random Forest (Ho 1998; Breiman 2001) is an ensemble-based technique for classification. It is computationally efficient in achieving the predictive performance of the detection of real-world malware prediction task. The Random Forest classifier generates many individual learners (decision trees) and makes predictions by aggregating the results of individual learners. Each decision tree in the Random Forest is a classifier that outputs the class by an individual tree, based on the value that occurs most frequently in the class of datasets. The Random Forest classifier combines such decision trees with a 'bagging' approach. In bagging, each decision tree is built separately by working with 'bootstrap' sample messages of the input training set. All bootstrap messages are chosen by repeated random sub-sampling with the replacement of the original training set, while matching the size of the bootstrap samples with the training set of the input data (Alam and Vuong 2013). When constructing a decision tree in the Random Forest classifier, each node in the decision tree is made of features that are randomly selected, and it helps to minimize interdependence (correlation) between the feature attributes and makes the results less susceptible to noise.

In Random Forest classifier, training variables such as the number of decision trees and number of features are randomly selected from the input file. The selection of features per decision tree at each node decides the error rate of the classifier. According to Breiman (Breiman 2001), the error rate of the Random Forest classifier is estimated by Out-of-Bag (OOB) error rate. The OOB error rate indicates the true prediction error of the Random Forest classifier. Since each tree uses a different bootstrap sample from the original training dataset, some observations end up in the bootstrap sample more than once, while others are not used in the training set (i.e., OOB). In the Random Forest classifier, about one-third of the bootstrap samples are left out for building k^{th} tree from the bootstrap sample of each tree (Breiman and Cutler 2017). Thus, the error rate of the Random Forest classifier depends on the interdependence between two trees and the classification strength of each decision tree (Alam and Vuong 2013). Finally, the estimation of the Random Forest classifier error rate is based on the aggregation of the OOB prediction. In this study, the Random Forest classifier achieved remarkably high accuracy on both the generated and benchmarked datasets than any other classifiers under the default and increased parameter setting of the WEKA. The complete experimental results' analysis of the Random Forest classifier is discussed in Section 6.5.

Random Tree technique takes the given input FFV by classifying each tree in the forest. The output of the labelled class is an indication of the received majority of the votes. All the trees in the training set are trained with the same parameter using the bootstrap procedure.

J48 is based on the C4.5 decision tree implementation algorithm (Salzberg 1994). It builds a decision tree based on the attribute values present in the testing data to classify new instances using the concept of information entropy. This technique creates a node for each decision tree by splitting the dataset into subsets. The attribute which tops the normalized IG score is used to make the decision of the classifier.

6.5 Analysis of Results

Six popular machine learning techniques (discussed in Section 6.4.4) were used to measure the effectiveness of the proposed approach individually. To measure the generalization capacity of each classifier, we performed a comprehensive set of evaluation runs by following a common experimental procedure on the both the generated and benchmarked datasets. Each dataset has a different training set, testing set, and validation set file. For each feature selection technique, three separate training, testing, and validation files were constructed and each of the three training files is of different feature lengths L=100, L=200, and L=300. Similarly, each of the three testing files and validation files is of different feature lengths L=100, L=200, and L=300. The A-IntExt system was evaluated using the testing set and the validation set, and the obtained results were compared with the standard 10-fold cross-validation test results of the validation set. This was repeated ten times and the observed mean of the results was recorded. All the chosen classifiers were initially trained with the default parameters available in the WEKA. Table 6.6 provides the details of the parameters used by each classifier during the experiment on both the generated and benchmarked datasets.

Table 6.6: List of default (D) parameters used by the classifiers on WEKA. The Increased (I) values of default parameters are highlighted in bold

Classifier name	Chosen parameters
Random Forest	-I 10 -K 0 -S 1, **-I 80 -K 0 -S 1 (HF)**
Naive Bayes	not answered in WEKA 3
SVM(SMO)	-C 1.0 -L 0.001 -P 1.0E-12 -N 0 -V -1 -W 1 -K "weka.classifiers.functions. supportVector.PolyKernel -C 250007 -E 1.0"
Random Tree	-K 0 -M 1.0 -S 1
Simple Logistic	-I 0 -M 500 -H 50 -W 0.0
J48	C 0.25 -M 2

6.5.1 Result Analysis of Generated Dataset

We performed two experiments (Experiment-I and Experiment-II) that correspond to the evaluation results of the testing set and the 10-fold cross-validation of the validation set. The aim of these experimentations was to examine the effectiveness of the proposed feature vector generation method of the A-IntExt system on the generated dataset at the VMM. It represents the real-world scenario of a virtualized environment when an unknown malware infects the Monitored VMs. **Experiment-I:** In the testing phase, each classifier was evaluated individually with the topmost features selected based on the score assigned by the HF and three feature selection techniques such as the CS, NF, and IG. Each classifier performance was then measured separately with three different topmost feature lengths of L=100, L=200, and L=300. Each time we select the topmost features of lengths L=100, L=200, and L=300 from the training set, the same feature length is used for the testing set generated by an appropriate feature selection technique. Thus, we run all the product combination of twelve training files and twelve testing files for each classifier.

In the evaluation of the testing set experiments, it can be observed that the Random Forest classifier achieved the highest accuracy of 96.86% with 0.031 FPR for L= 300 on the HF selection technique as shown in Table 6.7. Further, it also achieved a pretty good accuracy on two different L=100 and L=200 FFVs of other three feature selection techniques as compared to other classifiers. Interestingly, the accuracy results reported by all the classifiers with HF selection technique based FFV is significantly more than the other individual feature selection techniques even under different

Table 6.7: Detection accuracy (%) of malware using different feature selection techniques on both testing and validation sets experiments of generated dataset

Feature selection technique	Classifiers	Feature length of testing set						Feature length of validation set					
		L=100		L=200		L=300		L=100		L=200		L=300	
		ACC	FPR	ACC	FPR	ACC	FPR	ACC	FPR	ACC	FPR	ACC	FPR
IG	Naive Bayes	79.33	0.207	82.00	0.180	83.11	0.169	82.88	0.171	84.66	0.153	85.11	0.149
	SVM(SMO)	80.66	0.193	83.55	0.164	86.00	0.140	83.14	0.169	85.55	0.144	86.66	0.133
	Simple Logistic	82.44	0.176	84.22	0.158	85.11	0.149	81.11	0.189	82.44	0.176	84.88	0.151
	Random Forest	82.66	0.173	84.88	0.151	86.66	0.133	85.77	0.142	86.44	0.136	**87.11**	**0.129**
	Random Tree	80.22	0.198	82.88	0.171	83.33	0.167	80.22	0.198	83.55	0.164	84.15	0.158
	J48	78.22	0.218	79.10	0.209	81.33	0.187	79.11	0.209	80.22	0.198	81.33	0.187
NF	Naive Bayes	81.55	0.184	82.44	0.176	84.22	0.158	80.44	0.196	81.55	0.184	82.22	0.178
	SVM(SMO)	81.15	0.189	84.22	0.158	85.77	0.142	83.14	0.169	84.01	0.161	85.11	0.149
	Simple Logistic	85.11	0.149	86.22	0.138	86.88	0.131	85.11	0.149	86.22	0.138	87.11	0.129
	Random Forest	84.22	0.158	86.22	0.138	87.77	0.122	86.66	0.133	87.11	0.129	**88.22**	**0.118**
	Random Tree	80.00	0.200	81.11	0.189	83.03	0.170	81.33	0.817	82.88	0.171	84.44	0.156
	J48	80.66	0.193	81.77	0.182	82.00	0.180	80.22	0.182	81.33	0.187	82.22	0.178
CS	Naive Bayes	81.77	0.182	82.22	0.178	84.22	0.158	83.14	0.169	84.88	0.151	85.11	0.149
	SVM(SMO)	83.11	0.169	83.77	0.162	84.00	0.160	84.25	0.157	85.33	0.147	85.52	0.145
	Simple Logistic	83.77	0.162	84.88	0.151	85.11	0.149	84.66	0.153	85.71	0.143	86.22	0.138
	Random Forest	86.66	0.133	87.33	0.127	88.88	0.111	87.11	0.129	88.22	0.118	**89.33**	**0.107**
	Random Tree	81.33	0.187	82.22	0.178	83.55	0.164	82.36	0.176	83.55	0.165	84.22	0.158
	J48	81.77	0.182	82.00	0.180	82.66	0.173	81.77	0.182	82.36	0.176	83.07	0.169
HF	Naive Bayes	92.00	0.080	93.11	0.069	94.00	0.060	91.33	0.087	94.66	0.053	95.14	0.048
	SVM(SMO)	91.55	0.084	94.00	0.060	95.14	0.048	94.63	0.054	95.33	0.047	96.22	0.038
	Simple Logistic	92.22	0.078	93.42	0.066	94.88	0.051	94.44	0.056	95.78	0.042	96.44	0.036
	Random Forest (D)	94.22	0.058	95.33	0.047	**96.86**	**0.031**	96.44	0.036	97.11	0.029	**98.00**	**0.020**
	Random Forest (I)	95.55	0.044	96.22	0.038	**97.11**	**0.029**	96.88	0.031	97.55	0.024	**99.55**	**0.004**
	Random Tree	92.92	0.071	93.77	0.062	95.77	0.042	93.63	0.063	94.00	0.060	95.14	0.048
	J48	90.66	0.093	91.53	0.085	93.33	0.067	92.22	0.078	93.55	0.064	94.22	0.058

feature lengths. The overall performance in terms of accuracy of the four classifiers such as Navies Bayes, SVM (SMO), Simple Logistic and Random Tree was greater than 91.55%. The J48 classifier reported the lowest accuracy ranging from 90.66% to 93.33% for L= 100, 200, and 300 with the HF selection technique.

Since the Random Forest classifier achieved the highest accuracy, we further deviated its default parameter by increasing the number of decision trees up to 80 until we noticed stable accuracy. Finally, the Random Forest classifier (I) reported the stable accuracy of 95.55% with 0.044 FPR, 96.22% with 0.038 FPR, and 97.11% with 0.029 FPR for L=100, 200, and 300, respectively, as shown in Table 6.7.

Experiment-II: In order to examine the stability of the proposed A-IntExt system and to avoid `over-fitting`, the 10-fold cross-validation approach was applied and the experimental evaluation results are presented in Table 6.7. Next, during the evaluation of each classifier, we selected three different FFV of lengths L=100, L=200, and 300 from the training set to train the classifier and same feature length from the validation set by following the procedure discussed in Section 6.4.2. Further, these validation set files were used to check the actual maliciousness of the detected processes in terms of detection accuracy.

The evaluation process of the validation set was evaluated by using the 10-fold cross-validation. It randomly splits the original input file into 10 equal subparts, where 9 subparts are used as the training dataset and the remaining 1 subpart is used as the validation data to measure the detection efficiency of the classifier. The cross-validation process is reiterated 10 times (the folds) for every combination with the condition that each subpart is used once as the validation data. Finally, the outcome of each fold is averaged to estimate the overall efficiency of the tested model. In this way, we evaluated the cross-validation phase, and the same steps were repeated separately for different FFVs of different sizes from different feature selection techniques. This approach helps in systematically evaluating the feasibility of our proposed A-IntExt system to measure the detection accuracy of malware from the semantically reconstructed hidden and dubious executables at the VMM.

The evaluation process of all the six classifiers were evaluated with topmost features selected based on the score assigned by the feature selection techniques. In the initial experiments, we observed that the Random Forest classifier achieved the highest accuracy on all three different L for all feature selection techniques, as compared to the other classifiers. The right side of Table 6.7 provides details of the malware detection accuracy achieved by different classifiers for 10-fold cross-validation results of the validation set. In particular, the Random Forest classifier achieved an accuracy of 98.00% with 0.020 FPR for L=300 features suggested by the HF selection technique under the default parameters (tree size, T=10), where, T represents the number of decision trees in the ensemble.

Based on this observation, additional experimental work was carried out to investigate the impact of the number of decision trees on the detection accuracy of the Random Forest classifier. By setting default parameters with 80 decision trees for the Random Forest (I) classifier, we noticed that the HF selection technique based FFV encouraged it to achieve maximum accuracy of 96.88% with 0.031 FPR, 97.55% with 0.024 FPR, and 99.55% with 0.004 FPR for L=100, L=200, and L=300 features, respectively. This is due to the following reasons: 1) The FFV of the HF selection technique consists of features recommended by three different popular feature selection techniques as discussed in Section 6.2.1. These unique features influence the classifier decision; and 2) The Random Forest classifier uses multiple decision trees that are randomly chosen to vote for overall detection performance of the given input file, where each decision tree classifies the new instance of features with a majority of the vote (Oshiro et al. 2012).

For the generated dataset, the Random Forest classifier achieved the highest accuracy of 99.55% with L= 300 features of the HF selection technique, for increased size of the decision trees from 10 to 80. Further, there was no significant improvement in the Random Forest classifier accuracy after 80 decision trees. Another common observation is that the same Random Forest classifier reasonably achieved good accuracy for the IG, NF, and CS feature selection techniques with different feature lengths as compared to the other classifiers.

Since the Random Forest classifier yielded the highest accuracy on default and increased parameter settings, the predictive performance of this classifier was also evaluated while noticing the error rate when decision trees size increased from 10 to 80. Figure 6.4 depicts that lower OOB error rate was achieved at increased trees of 30 for L=200 and L=300 features. Similarly, for L=100 features, lower OOB error

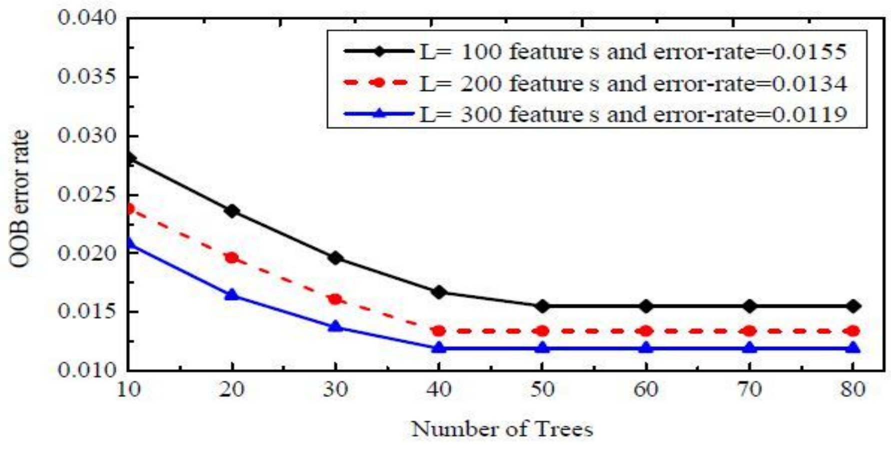


Figure 6.4: OOB error rate comparison with different tree size for HF selection technique on the validation set of generated dataset

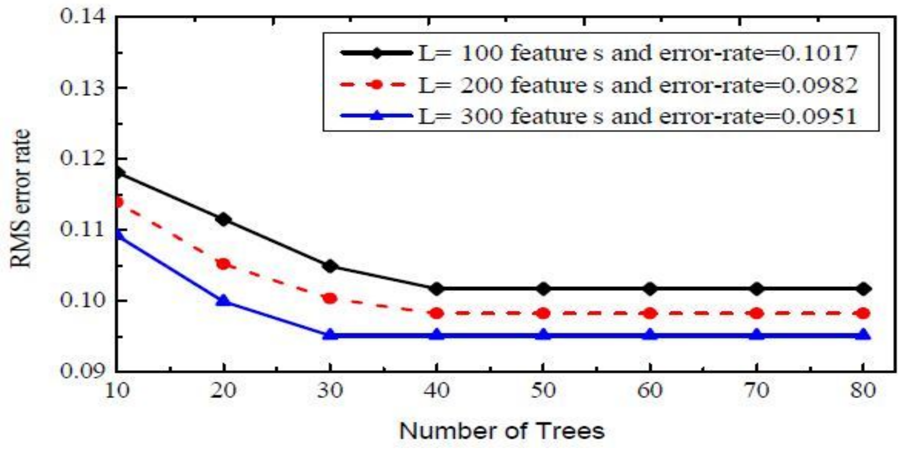


Figure 6.5: RMS error rate comparison with different tree size for HF selection technique on the validation set of generated dataset

rate was achieved at 40 decision trees. The final inference is that, if L is low then, maximum decision trees are required to stabilize the OOB error rate and vice versa. In this experiment, the error rate stabilized after reporting minimum OOB error rate as 0.0155 at 50 trees, 0.0134 at 40 trees, and 0.0119 at 40 trees for L of 100, 200, and 300 features, respectively.

Additionally, for the Random Forest classifier, we measured the best Root Mean Square (RMS) error rate. It can be seen from Figure 6.5 that the error rate of 0.0999 was reported for 20 trees with L= 300 features. In addition, the stable value of the observed RMS error rate was 0.1017 at 40 trees, 0.0982 at 40 trees, and 0.0951 at 30 trees for L=100, L=200, and L=300 features, respectively.

Table 6.8: Experiment results of malware detection with different feature selection techniques for validation set of generated dataset

Feature length		L=100				L=200				L=300			
Classifier	Metrics	IG	NF	CS	HF	IG	NF	CS	HF	IG	NF	CS	HF
Naive Bayes	TPR	0.829	0.804	0.831	0.913	0.847	0.816	0.849	0.947	0.851	0.822	0.851	0.951
	AUC	0.819	0.809	0.853	0.913	0.825	0.812	0.877	0.924	0.839	0.829	0.889	0.938
	MCC	0.658	0.609	0.663	0.827	0.693	0.631	0.698	0.893	0.702	0.645	0.702	0.903
SVM (SMO)	TPR	0.831	0.831	0.843	0.946	0.856	0.841	0.853	0.953	0.867	0.851	0.855	0.962
	AUC	0.823	0.848	0.879	0.925	0.863	0.862	0.891	0.932	0.889	0.868	0.912	0.967
	MCC	0.663	0.663	0.685	0.893	0.711	0.681	0.707	0.907	0.733	0.702	0.710	0.924
Simple Logistic	TPR	0.811	0.851	0.847	0.944	0.824	0.862	0.857	0.958	0.849	0.871	0.862	0.964
	AUC	0.832	0.848	0.869	0.929	0.839	0.838	0.857	0.938	0.857	0.869	0.876	0.948
	MCC	0.622	0.703	0.693	0.889	0.649	0.725	0.714	0.916	0.698	0.742	0.725	0.929
Random Forest (I)	TPR	0.858	0.867	0.871	0.969	0.864	0.871	0.882	0.976	0.871	0.882	0.893	***0.996***
	AUC	0.838	0.856	0.874	0.938	0.845	0.876	0.898	0.946	0.863	0.896	0.928	***0.995***
	MCC	0.716	0.733	0.742	0.938	0.729	0.742	0.765	0.951	0.742	0.765	0.787	0.991
Random Tree	TPR	0.802	0.813	0.824	0.936	0.836	0.829	0.836	0.940	0.842	0.844	0.842	0.951
	AUC	0.826	0.835	0.869	0.924	0.839	0.841	0.876	0.937	0.858	0.867	0.883	0.943
	MCC	0.604	0.627	0.648	0.903	0.671	0.658	0.671	0.916	0.683	0.689	0.685	0.926
J48	TPR	0.791	0.802	0.818	0.922	0.802	0.813	0.824	0.936	0.813	0.822	0.831	0.942
	AUC	0.797	0.825	0.819	0.916	0.856	0.868	0.886	0.928	0.857	0.876	0.899	0.936
	MMC	0.582	0.605	0.636	0.845	0.604	0.627	0.648	0.871	0.627	0.645	0.662	0.884

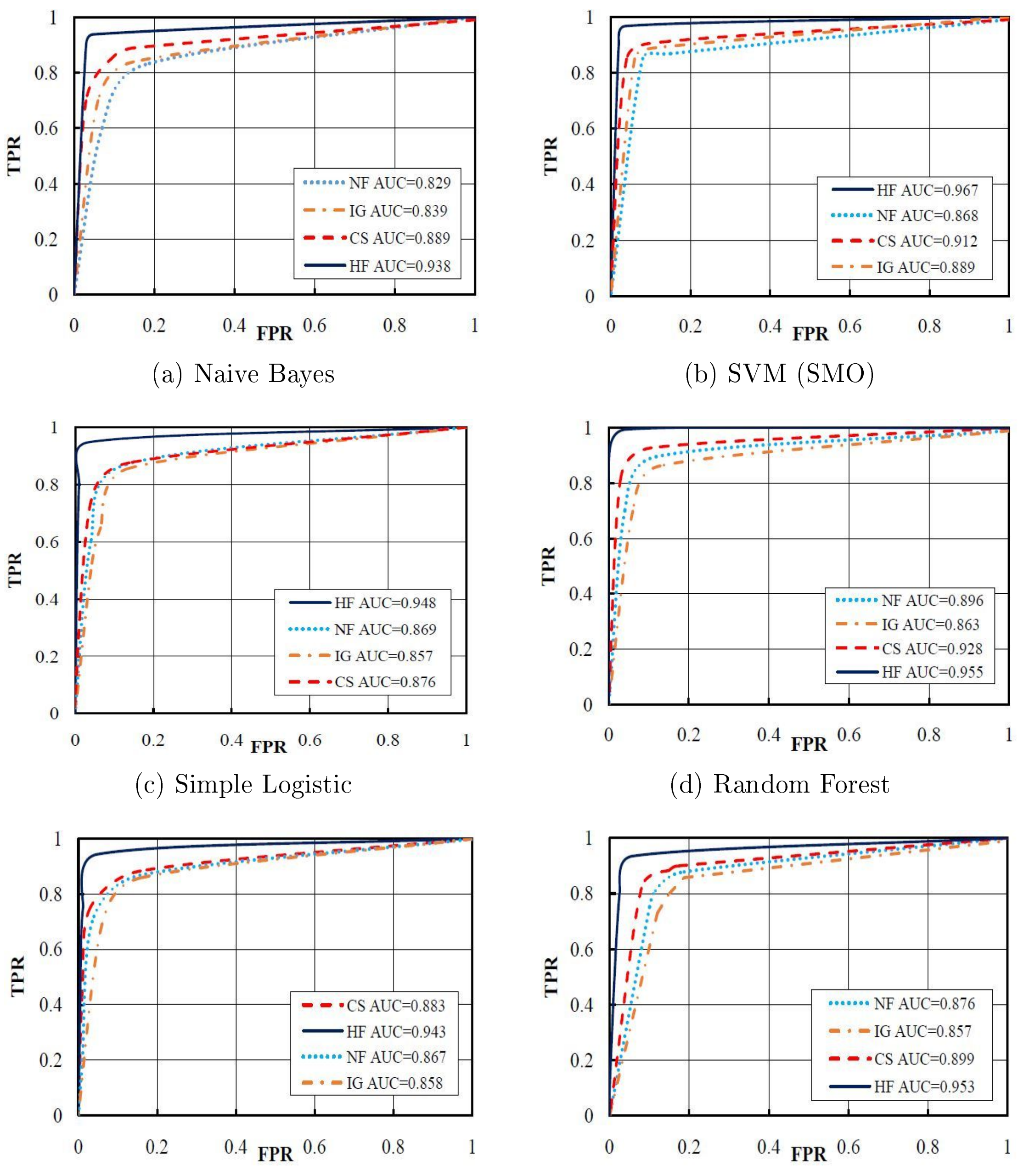
It can also be seen from Table 6.7 that other classifiers such as SVM (SMO), Simple Logistic, Random Tree, and J48 delivered accuracy ranging from 92.22% to 95.14% for L=100 to L=300 and the accuracy produced by the Navies Bayes classifier was the poorest, at 91.33% for L=100 of the this technique. The obtained results of this technique were compared with the other three feature selection techniques (i.e., IG, NF, and CS) of different feature lengths. The Random Forest classifier attained an accuracy of 87.11% with 0.129 FPR, 88.22% with 0.118 FPR and 89.33% with 0.107 FPR for L=300 features of IG, NF, and CS, respectively. In comparison, other classifiers produced the least accuracy of 79.11% and best accuracy of 87.11% for various feature lengths of IG, NF, and CS, which is significantly lesser than the HF selection technique of a similar feature length.

The malware recognition proficiency of different classifiers measured with different evaluation metrics such as TPR, Area Under the Curve (AUC), and MCC is shown in Table 6.8. The TPR represents the correct detection of positive instances (i.e., feature vectors of malicious executables) by the classifier and the MCC considers data imbalance even though the classes are of different sizes by using the amount of correctly and incorrectly classified features (instances) based on -1 (if wrong prediction of the model) and +1 (model prediction is perfect) of model prediction.

The ROC curve represents the trade-off between the TPR and the FPR. The point (0, 1) indicates that the classifier is ideal since it correctly classifies all the negative samples as negative and the positive samples as positive with zero FPR. The performance of each classifier is evaluated separately by plotting the ROC curves for each chosen classifier as shown in Figure 6.6. The AUC metric is utilized to measure the ROC curves. All data points lying on the upper left corner of each ROC curve indicates the detection rate of the classifier as high with low FPR, i.e., optimal high-performance (Egan 1975). The higher value of the AUC denotes better accuracy of the classifier.

Each sub-figure of Figure 6.6 represents the ROC curve of individual classifiers such as Navie Bayes, SVM (SMO), Simple Logistic, Random Forest, Random Tree, and J48 for L= 300 features. Each ROC curve of the classifier corresponds to a separate feature selection technique.

Figure 6.6 and Table 6.8 show the AUC value of all the classifiers. The AUC



(a) Naive Bayes

(b) SVM (SMO)

(c) Simple Logistic

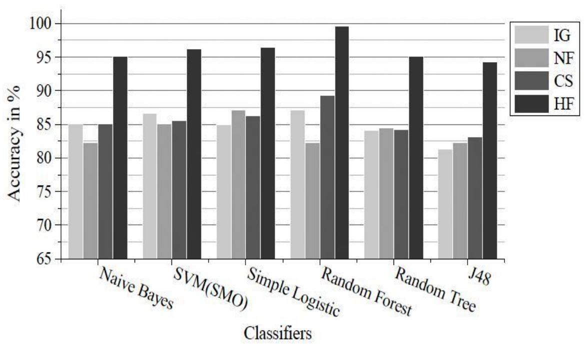
(d) Random Forest

(e) Random Tree

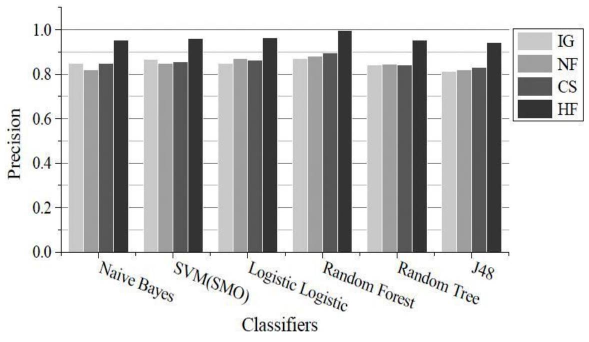
(f) J48

Figure 6.6: ROC curves for feature selection techniques used by different classifiers for L= 300 on the validation set of generated dataset

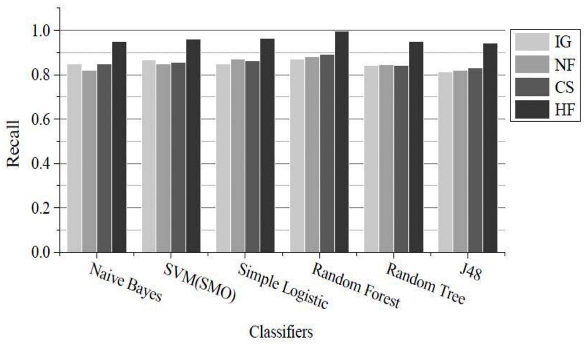
value of the HF selection technique is superior as compared to other feature selection techniques such as IG, NF, and CS. The Random Forest classifier (I) in particular achieved the highest AUC of 0.995, while J48 reported the lowest value of AUC with 0.936 for L=300 features of the HF selection technique. Other classifiers such as Naive Bayes, SVM (SMO), Simple Logistic, and Random Tree achieved AUC of 0.938, 0.967, 0.948, and 0.943, respectively. Furthermore, it can also be observed from Table 6.8,



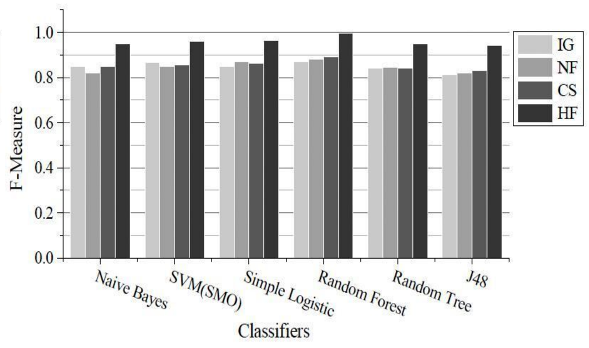
(a) Accuracy



(b) Precision



(c) Recall



(d) F-Measure

Figure 6.7: A-IntExt system malware detection evaluation using different performance metrics for L=300 on the validation set of generated dataset

that the Random Forest classifier reasonably produces a higher value of AUC for IG, NF, and CS based feature selection techniques with different feature lengths as compared to other classifiers.

The analysis of results for each of the chosen machine learning classifiers is measured by other performance metrics such as Accuracy, Precision, Recall, and F-measure as shown in Figure 6.7. The classifiers' performance was evaluated separately for individual feature selection technique and HF selection technique with L= 300 features. The HF selection technique performed exceptionally well as compared to other feature selection techniques on all the performance metrics. We notice that the Random Forest classifier outperformed other classifiers and achieved an accuracy of 99.55% with perfect Precision of 0.996, high Recall of 0.996, and F-measure value equal to 0.997. The minimum performance shown by the classifier for HF selection approach is J48 with accuracy 94.22%, Precision=0.942, Recall=0.942, and F-Measure=0.943.

Additionally, the performance of each classifier was measured with other feature selection techniques such as IG, NF, and CS based FFVs. Since, all the classifiers remarkably produced the best accuracy for L=300 in all feature selection techniques, the obtained accuracy, Precision, Recall, and F-Measure results are depicted in the graphs of Figure 6.7. Overall, the classifiers performed well for the HF selection technique based FFV and produced the best accuracy, Precision, Recall, and F-Measure results as compared to other individual feature selection techniques.

6.5.2 Result Analysis of Benchmarked Datasets

In order to validate the detection ability of the proposed A-IntExt system, we also considered independent public benchmarked malware datasets in this work and extensive experimental analysis was performed. The obtained results are presented similar to the generated dataset. The distribution of the dataset used in this evaluation is discussed in Section 6.3.1. The obtained dataset consists of a maximum number of benign and considerable amount of malicious executables that represents a real world situation and the same dataset was used to validate our proposed A-IntExt system. Since the obtained dataset is static in nature, we generated a separate FFV for each feature selection technique such as CS, NF, IG, and HF. Each individual feature selection technique was evaluated with three different feature lengths L=100, L=200, and L=300. The six classifiers, which were also used in the generated dataset experiments, were used in this experimental work with default parameters setting of WEKA as shown in Table 6.6.

Experiment-I: In the testing set evaluation, the distribution of the samples as shown in Table 6.3 were used and these were excluded from the training set. As the testing phase represents the real-time scenario of unknown malware detection, we conducted comprehensive tests. Table 6.9 provides the experimental results obtained by the testing set evaluation of all the six classifiers with different feature selection techniques. The Random Forest classifier yielded the best accuracy of 93.77% with 0.062 FPR, 94.88% with 0.051 FPR, and 96.00% with 0.040 FPR for feature length of 100, 200, and 300 of the HF selection technique. Similarly, under increased decision tree size of 80, the Random Forest (I) produced a maximum accuracy of 95.11% with 0.049 FPR, 96.22% with 0.038 FPR, and 97.77% with 0.22 FPR for L=100, L=200, and L=300. In addition, it also produced the best accuracy on other feature selection

Table 6.9: Detection accuracy (%) of malware using different feature selection techniques on both testing and validation sets experiments of benchmarked datasets

Feature selection technique	Classifiers	Feature length of testing set						Feature length of validation set					
		L=100		L=200		L=300		L=100		L=200		L=300	
		ACC	FPR	ACC	FPR	ACC	FPR	ACC	FPR	ACC	FPR	ACC	FPR
IG	Naive Bayes	78.66	0.217	79.11	0.209	80.22	0.198	80.22	0.198	82.00	0.180	82.66	0.173
	SVM(SMO)	80.88	0.191	82.66	0.173	84.00	0.160	81.77	0.182	83.33	0.167	85.55	0.144
	Simple Logistic	81.33	0.187	82.22	0.178	83.77	0.162	82.22	0.178	84.03	0.160	84.22	0.158
	Random Forest	82.22	0.178	83.33	0.167	84.88	0.151	83.11	0.169	84.22	0.158	86.22	0.138
	Random Tree	77.11	0.229	78.66	0.213	80.22	0.198	78.22	0.218	80.22	0.198	82.81	0.172
	J48	78.22	0.218	78.88	0.211	80.00	0.200	78.22	0.218	80.04	0.200	81.77	0.182
NF	Naive Bayes	78.00	0.220	78.46	0.215	80.22	0.198	79.11	0.209	80.22	0.198	81.77	0.182
	SVM(SMO)	81.77	0.182	82.88	0.171	83.33	0.167	82.22	0.178	83.33	0.167	84.44	0.156
	Simple Logistic	83.77	0.162	84.22	0.158	85.58	0.144	84.44	0.156	85.33	0.147	86.44	0.136
	Random Forest	84.00	0.160	84.88	0.151	85.87	0.141	84.88	0.151	85.77	0.142	86.88	0.131
	Random Tree	79.55	0.206	80.88	0.191	82.88	0.171	80.22	0.198	82.81	0.172	83.33	0.167
	J48	81.33	0.187	82.66	0.173	83.77	0.162	80.66	0.193	81.33	0.187	82.22	0.178
CS	Naive Bayes	81.11	0.189	82.22	0.178	83.77	0.162	82.22	0.178	83.11	0.169	84.03	0.160
	SVM(SMO)	82.66	0.173	83.11	0.169	84.00	0.160	83.11	0.169	84.22	0.158	84.88	0.151
	Simple Logistic	82.88	0.172	83.55	0.164	84.44	0.156	83.77	0.162	84.66	0.153	85.11	0.149
	Random Forest	85.11	0.149	86.00	0.140	87.11	0.129	86.00	0.140	87.11	0.129	88.22	0.118
	Random Tree	80.00	0.200	81.77	0.182	82.88	0.171	81.11	0.189	82.22	0.178	83.11	0.169
	J48	79.77	0.202	80.22	0.198	81.55	0.184	80.00	0.200	81.33	0.187	82.66	0.173
HF	Naive Bayes	91.11	0.089	92.66	0.073	93.33	0.067	92.88	0.071	93.97	0.060	94.00	0.060
	SVM(SMO)	92.22	0.078	93.33	0.067	94.22	0.058	93.11	0.069	94.66	0.053	95.31	0.047
	Simple Logistic	93.55	0.064	94.22	0.058	95.77	0.042	94.22	0.058	95.11	0.049	96.66	0.033
	Random Forest (D)	93.77	0.062	94.88	0.051	**96.00**	**0.040**	95.11	0.049	96.00	0.040	**97.11**	**0.029**
	Random Forest (I)	95.11	0.049	96.22	0.038	**97.77**	**0.022**	96.88	0.031	97.33	0.027	**98.88**	**0.011**
	Random Tree	92.33	0.078	93.11	0.069	93.77	0.062	92.22	0.078	93.55	0.064	94.22	0.058
	J48	91.33	0.087	92.66	0.073	93.77	0.062	92.00	0.080	93.11	0.069	94.12	0.058

techniques with different feature lengths. Other five classifiers also attained maximum accuracy in the range of 91.11% to 95.77% on different feature lengths with the HF selection technique. While evaluating these five classifiers with the FFV of IG, NF, and CS, the maximum accuracy range reported was 78.22% to 84.88% for IG, 78.00% to 85.58% for NF, and 79.77% to 84.44% for CS.

Experiment-II: For validation set, we conducted the 10-fold cross-validation experiments. Each classifier performance was evaluated independently with three different feature lengths L=100, L=200, and L=300 features and the obtained results are depicted in Table 6.9.

It can be seen that the accuracy of the Random Forest classifier outperformed all the other classifiers for all the four feature selection techniques. In particular, for HF selection based features, it performed exceptionally well by producing a very good accuracy of 95.11% with 0.049 FPR, 96.00% with 0.040 FPR, and 97.11% with 0.029 FPR for feature length of 100, 200, and 300 features, respectively. The obtained high accuracy results of the Random Forest classifier signifies that the A-IntExt system is competent enough in correctly identifying the benchmarked datasets malware also.

Similar to the generated dataset experiments, we increased the number of decision trees in steps of 10 trees in each experiment and fixed the maximum number as 80 trees. From the obtained experimental results, we noticed that the maximum stable accuracy achieved by the Random Forest classifier was 96.88% with 0.031 FPR, and 97.33 with 0.027 FPR for L=100 and L=200, respectively. Specifically for L= 300 features, the Random Forest classifier achieved **98.88%** accuracy and **0.011** FPR.

At the same time, we also measured the error rate of this classifier. Figure 6.8 depicts the estimation of the OOB error rate of the Random Forest classifier for HF selection techniques on the benchmarked datasets. Based on the experimental observations, lower OOB rate was achieved when the number of trees equaled to 40, 30, and 30 with L=100, L=200, and L=300 features, respectively. Further, the OOB error rate of this classifier stabilized by reporting error rate of 0.0178 at 50 trees, 0.0155 at 40 trees, and 0.0131 at 40 trees with L=100, L=200, and L=300 features, respectively. At the same time, the RMS error rate of this classifier stabilized (as shown in Figure 6.9) when the value of the error reached 0.1032 at 40 trees, 0.1002 at 30 trees, and 0.0972 at 30 trees for L=100, L=200, and L=300 features, respectively.

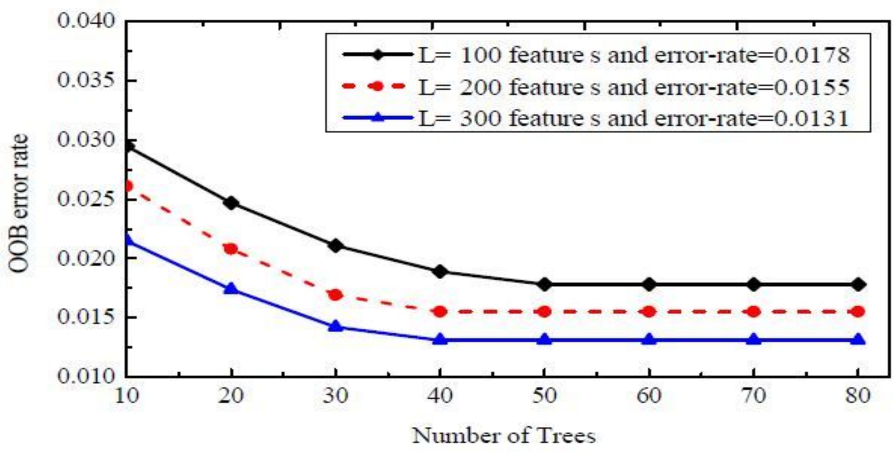


Figure 6.8: OOB error rate comparison with different tree size for HF selection technique on the validation set of benchmarked datasets

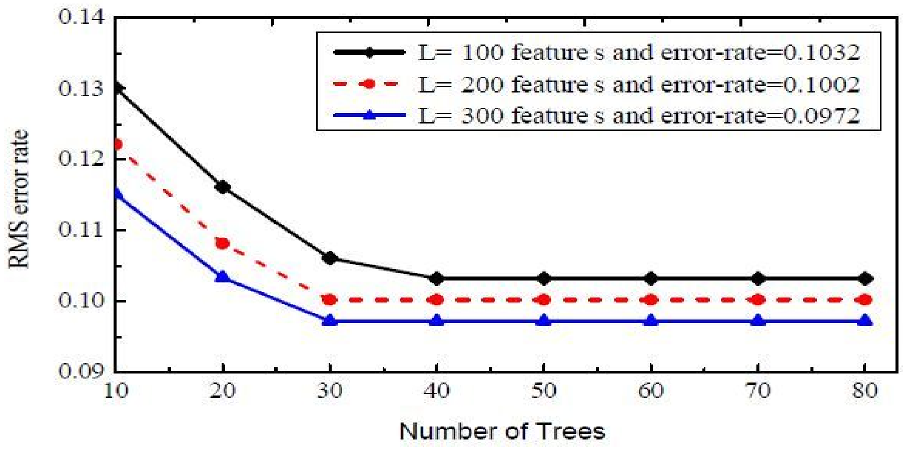
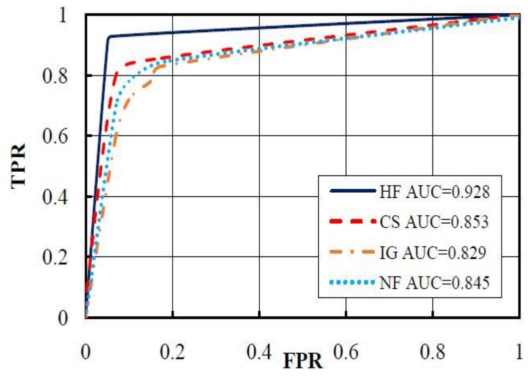
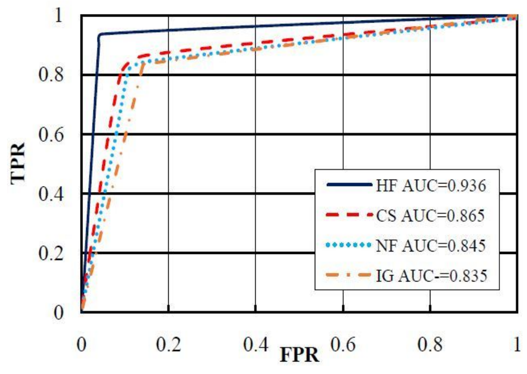


Figure 6.9: RMS error rate comparison with different tree size for HF selection technique on the validation set of benchmarked datasets

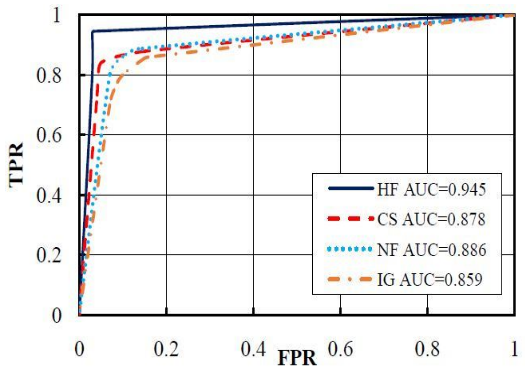
The other five classifiers also achieved accuracy greater than 92.00% for different L, but less than that of the Random Forest classifier. As seen in Table 6.9, for HF selection technique the second highest accuracy was yielded by the Simple Logistic classifier ranging from 94.22% to 96.66%, followed by SVM (SMO) with 93.11% to 95.31%, Random Tree with 92.22% to 94.22%, Navies Bayes with 92.88% to 94.00%, while the J48 classifier yielded lowest accuracy of 92.00% to 94.12% for L=100, L=200, and L=300. The overall performance of these classifiers was also evaluated based on the features recommended by the other three individual feature selection techniques and compared with the HF selection technique. The accuracy yielded by these classifiers for corresponding individual feature selection techniques is outlined in Table 6.9.



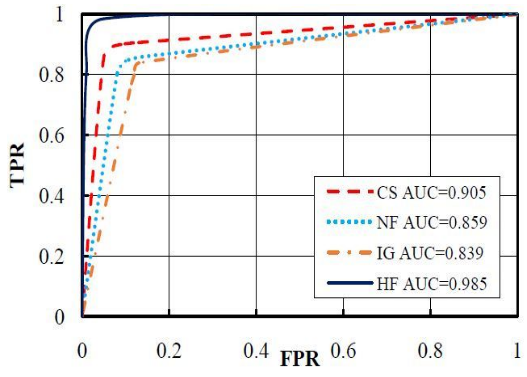
(a) Naive Bayes



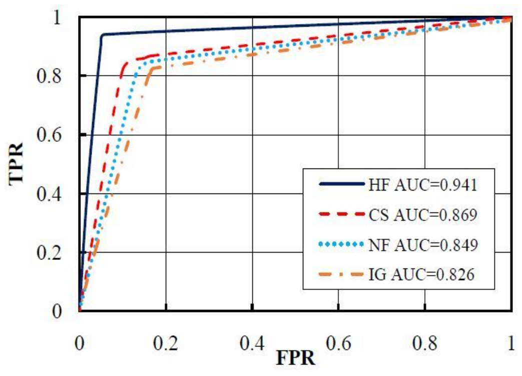
(b) SVM (SMO)



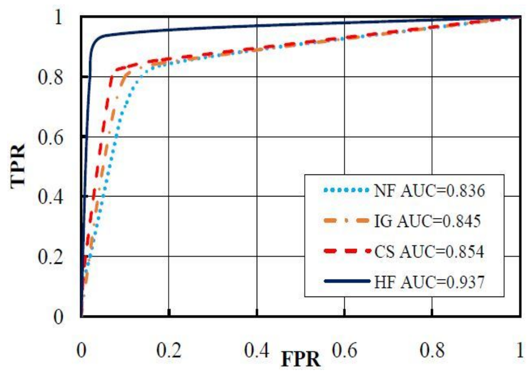
(c) Simple Logistic



(d) Random Forest



(e) Random Tree



(f) J48

Figure 6.10: ROC curves for feature selection techniques used by different classifiers for L= 300 on the validation set of benchmarked datasets

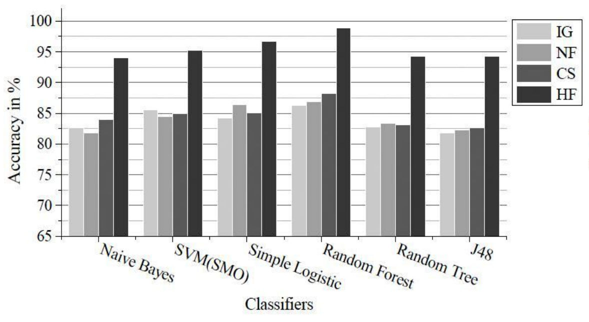
Table 6.10: Experiment results of malware detection with different feature selection techniques for validation set of benchmarked datasets

Feature length		L=100				L=200				L=300			
Classifier	Metrics	IG	NF	CS	HF	IG	NF	CS	HF	IG	NF	CS	HF
Naive Bayes	TPR	0.802	0.791	0.822	0.929	0.820	0.802	0.831	0.929	0.827	0.818	0.840	0.940
	AUC	0.810	0.792	0.821	0.905	0.821	0.834	0.841	0.916	0.829	0.845	0.853	**0.928**
	MCC	0.605	0.582	0.645	0.858	0.640	0.604	0.663	0.858	0.654	0.636	0.681	0.880
SVM(SMO)	TPR	0.818	0.822	0.831	0.931	0.833	0.833	0.842	0.947	0.856	0.844	0.849	0.947
	AUC	0.814	0.822	0.839	0.915	0.825	0.836	0.856	0.923	0.835	0.845	0.865	**0.936**
	MCC	0.636	0.644	0.662	0.862	0.667	0.667	0.685	0.893	0.711	0.692	0.698	0.893
Simple Logistic	TPR	0.822	0.844	0.838	0.942	0.840	0.853	0.847	0.951	0.842	0.864	0.851	0.951
	AUC	0.825	0.868	0.857	0.926	0.837	0.875	0.869	0.938	0.859	0.886	0.878	**0.945**
	MCC	0.645	0.689	0.676	0.884	0.681	0.707	0.693	0.902	0.684	0.729	0.702	0.902
Random Forest (I)	TPR	0.831	0.849	0.860	0.951	0.842	0.858	0.871	0.973	0.862	0.869	0.882	***0.960***
	AUC	0.819	0.826	0.840	0.921	0.823	0.837	0.857	0.932	0.839	0.859	0.905	***0.985***
	MCC	0.662	0.698	0.720	0.902	0.684	0.716	0.742	0.947	0.724	0.738	0.764	0.920
Random Tree	TPR	0.782	0.802	0.811	0.922	0.802	0.828	0.822	0.936	0.828	0.833	0.831	0.936
	AUC	0.798	0.824	0.842	0.915	0.815	0.835	0.856	0.926	0.826	0.849	0.869	**0.941**
	MCC	0.564	0.604	0.622	0.845	0.604	0.656	0.644	0.871	0.656	0.667	0.662	0.871
J48	TPR	0.782	0.807	0.800	0.920	0.800	0.813	0.813	0.931	0.818	0.822	0.827	0.931
	AUC	0.795	0.819	0.826	0.912	0.816	0.825	0.836	0.926	0.845	0.836	0.854	**0.937**
	MMC	0.564	0.613	0.600	0.840	0.601	0.627	0.627	0.862	0.636	0.644	0.653	0.862

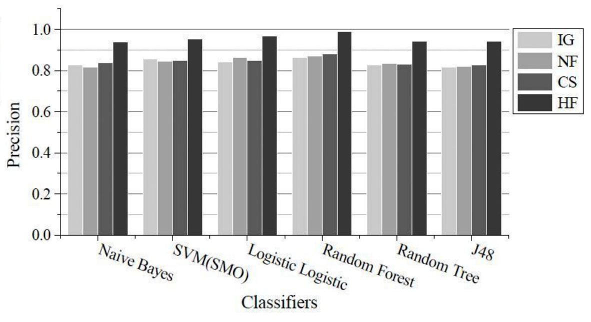
Furthermore, the performance of each classifier was compared in terms of the obtained AUC value by plotting separate ROC graphs for each chosen classifier. From Figure 6.10, it can be observed that the obtained value of AUC for the HF selection technique performed well as compared to other feature selection techniques with L= 300 features. Additionally, from Table 6.10, it can be seen that the Random Forest classifier attained the highest AUC value with 0.985 when L=300 features of the HF selection technique. This is due to the fact that, the more number of decision trees (i.e., T=80), help the Random Forest classifier to produce a better prediction. The other classifiers such as the Naive Bayes with 0.928, SVM (SMO) with 0.936, Simple Logistic with 0.945, Random Tree with 0.941, and J48 with 0.937 predicted excellent AUC for L=300 as well. Overall, the AUC value and ROC curves of all the classifiers confirmed that the A-IntExt system provided compelling results for feature length of 300 features of the HF selection technique. In addition, the Random Forest classifier moderately produced good AUC as compared to the other classifiers for top features recommendation by individual feature selection technique. The averages results of the TPR and MCC yielded by each classifier with features recommended by different feature selection technique is illustrated in Table 6.10. The Random Forest (I) performed best with TPR value of 0.960 for L=300 of the HF selection technique.

Finally, Figure 6.11 shows the evaluation results of other performance metrics such as Accuracy, Precision, Recall, and F-Measure for different classifiers for feature length 300 features. It can be seen from Figure 6.11 (a) that, the accuracy of all the classifiers for the HF selection technique significantly outperformed other feature selection techniques for 300 features. Consecutively, other performance metrics such as Precision, Recall, and F-Measure performed well for L=300 features of the HF selection technique. As per the overall observations, the best performance is shown by Random Forest classifier (I) includes 98.88% accuracy, 0.989 highest Precision, and 0.989 and 0.990 Recall and F-Measure, respectively. The lowest performance was reported by Naive Bayes with 94% accuracy and 0.940, 0.940, and 0.942 of Precision, Recall, and F-Measure, respectively.

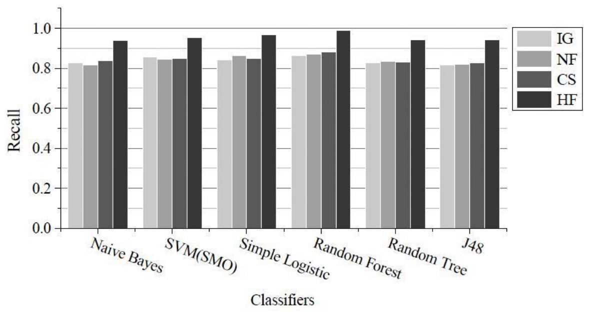
Similarly, the performance of the chosen classifiers were evaluated for other feature selection techniques such as IG, NF, and CS with feature length of 300 features. Finally, the obtained values of Precision, Recall, and F-Measure are illustrated in



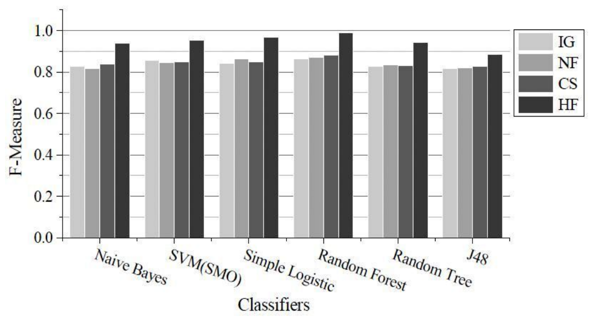
(a) Accuracy



(b) Precision



(c) Recall



(d) F-Measure

Figure 6.11: A-IntExt system malware detection evaluation using different performance metrics for L=300 on the validation set of benchmarked datasets

Figure 6.11 (b), (c), and (d), respectively.

6.5.3 Comparison of Results

The analysis of results on both the generated and benchmarked datasets show that the proposed A-IntExt system performs well on the generated dataset in terms of achieving higher accuracy as compared to the benchmarked datasets by most of the selected classifiers. The last column of Table 6.11 illustrates the difference in accuracy obtained in comparison of both the generated and benchmarked datasets for L= 300 of the HF selection technique. The Navies Bayes performs well on the generated dataset by attaining +1.14% greater accuracy than the benchmarked datasets, followed by the Random Tree with +0.92%, SVM (SMO) with +0.91%, Random Forest (I) with +0.67%, and J48 with 0.10%. Surprisingly, the Simple Logistic failed to perform well in the generated dataset as it attained -0.22% low accuracy compared to the benchmarked datasets. With these comparative results, we can argue that the proposed HF feature selection technique of our A-IntExt system is significant in detecting known and unknown (signature-less) malware for the semantically introspected and foren-

Table 6.11: Comparison of accuracy (%) for validation sets of generated and benchmarked datasets for L=300 of HF selection technique

Classifier	Generated dataset	Benchmarked datasets	Difference
Naive Bayes	95.14	94.00	+1.14
SVM(SMO)	96.22	95.31	+0.91
Simple Logistic	96.44	96.66	-0.22
Random Forest (I)	99.55	98.88	+0.67
Random Tree	95.14	94.22	+0.92
J48	94.22	94.12	+0.10

sically recreated executables (i.e., generated dataset) at the VMM. Further, it also works for other benchmarked datasets with good accuracy of 98.88%.

Finally, we would like to highlight the various observations made from the experiments of both the datasets. Firstly, the Random Forest classifier with the HF selection technique under feature length of 300 attained overall best results on testing and validation sets of both the datasets. The Random Forest classifier has a long history in producing good accuracy on text classification and it achieved similar performance in this domain too. Secondly, the Simple Logistic classifier also showed favorable performance but was lesser as compared to the Random Forest classifier with the HF selection technique on both the datasets. All other classifiers showed satisfactory performance with the HF feature selection technique by producing accuracy ranging from 95.14% to 96.22% on the generated dataset and 94.00% to 94.22% on the benchmarked datasets. Finally, the performance of the J48 classifier was poor as compared to the other classifiers on both the datasets.

6.6 Discussion

The current development of the proposed A-IntExt system included extended functionalities for detection and estimation of the symptoms of malware execution on the live Monitored VM. In addition, the incorporation of machine learning techniques was emphasized as the first scientific in-guest assisted VMI introspection technique to precisely detect and classify the running processes on the Monitored VM as benign or malicious at the VMM. The A-IntExt system performed this task from the semantically reconstructed executables that were introspected and forensically extracted at the VMM. The current demonstration of this approach is specific to the Windows guest OS to automatically detect the execution of large malware on the live Monitored VM by eliminating manual analysis.

The different categories of real world malware executables used in this work included self-hidden behavior malware, which hid on execution on the guest OS. In addition, we used both user-mode and kernel-mode rootkits to explicitly hide some benign and malicious running processes to test the detection feasibility of our proposed A-IntExt system. We practically confirmed the modified functionality of the VMI introspector of the A-IntExt system as being proficient in detecting hidden and malicious processes caused by stealthy malware and rootkits by traversing the semantic view of the `_EPROCESS` data structure of the live Monitored VMs.

In this study, the Random Forest classifier achieved high accuracy on both the generated and benchmarked datasets. The other chosen classifiers also achieved good accuracy, but it was marginally less as compared to the Random Forest classifier. The main reason to choose and increase the Random Forest classifier parameter was that under the default parameters it constantly produced the highest accuracy for dissimilar L of 100, 200, and 300 features. Keeping this in mind and in order to evaluate maximum detection accuracy of our feature selection methodologies, the Random Forest classifier was preferred in this work.

From the overall analysis of both the generated and benchmarked datasets' results, we can argue that the detection proficiency of the proposed A-IntExt system is efficient enough to produce better accuracy even with other classifier techniques. Finally, the systematic design and implementation of the proposed A-IntExt system were signified as a promising VMI-based malware detection system for a virtualized cloud environment.

6.6.1 Limitations

Like several techniques that depend on machine learning techniques for detection of malware, our proposed A-IntExt system has few shortcomings. Since malware authors have created a new class of malware called VM-aware malware (Ferrie 2007; Raffetseder et al. 2007), which is more prevalent in virtualized environments by exploiting some software or hardware artifacts provided by the virtualization layer between the live guest OS and bare hardware. Some malware samples of this type may not run in a virtualized environment or even run, but it performs a mysterious activity to foil the existing VM-based defensive solution. The VM-aware malware was created by adopting various tricks and techniques called primitive or simplistic (Liston and Skoudis 2006) to reluctant malware not to execute on the virtual environments. In addition, malware of this type disguises itself by going dormant or exhibiting non-malignant acts on the targeted guest OS. In this work, we considered the malware executables that run and its file access permission are not being denied by the Monitored guest OSs during evaluation of the A-IntExt system.

Another limitation of the proposed work is that during periodic introspection, the A-IntExt system does not assume that the proliferation of malware may hide another legitimate process either in user-mode or kernel-mode of the guest OS. In addition, the malware may also perform DKOM attack (Sparks and Butler 2005) by explicitly hiding process details at the kernel-mode of the guest OS. In such an instance, the extended functionality of the A-IntExt system is proficient in detecting the process hidden by the rootkit on the Monitored VMs. But, in order to explicitly test and evaluate the running processes hidden by the rootkit on Windows 7 Monitored VM, we did not find any publicly available rootkits, which would work for a Windows 7 Monitored VM. One such available `futo` rootkit (Butler and Silberman 2006) failed to execute on the Windows 7 guest OS. Due to this, the evaluation of the proposed A-IntExt system, particularly for Windows 7 guest OS was restricted to detect the execution of both known and unknown malware as shown in Table 6.1. However, the detection of such user-mode and kernel-mode rootkits based hidden processes on the Windows 7 guest OS and investigation of the malicious check on the detected one can be achieved by the executable file extractor and FVG components of the proposed A-IntExt system by leveraging machine learning techniques.

Since our proposed approach performs detection of malware from forensically extracted executables, it is certainly possible that malware that uses code obfuscation and stealth techniques would be susceptible to binary code features (Alazab et al. 2011; Liangboonprakong and Sornil 2013) where other types of malware would not. In such cases, the proposed approach may not contribute highest detection accuracy for the generated features of this particular family of malware. However, it is proficient in reconstructing a consistent active run state of such executables from the infected memory dump of the guest OS. As a consequence, our proposed approach achieved 99.95% of maximum detection accuracy for forensically reconstructed executables when performing an evaluation of this malware with other types that were partitioned as validation set of the generated dataset at the VMM.

Finally, in this work, the MFA involved extracting executable process information of the live infected memory dump of the Monitored VMs at the VMM. However, the possibility of virtualization-based malware or semantic value attack such as Subvert (King and Chen 2006), Blue Pill (Rutkowska 2006; Rutkowska and Tereshkin 2008), direct kernel structure manipulation attack (Bahram et al. 2010) shows the successful compromise of the virtualized environment. In such cases, the extraction of executables or binary semantic information of the infected memory dump may not be assumed to be legitimate. Moreover, in this work, we emphasize this kind of attacks as related to a theoretical discussion than practice.

6.7 Summary of The Work

In this work, we have presented the design, implementation, and evaluation of the A-IntExt system as an advanced in-guest assisted VMI-based security solution. It periodically scrutinizes the state of the introspected system in order to detect hidden, dead, and dubious processes on the Monitored VM. The intelligent decision functionality of the ICVA precisely estimates the symptoms of malware execution by cross examine the internally and externally gathered processes semantic view of the guest OS, while performing perfect memory acquisition of the Monitored VM. The A-IntExt system reconstructs the executables correspond to identified hidden and dubious processes from the digital artifacts of the Monitored VM. These executables are further analyzed using machine learning techniques to ascertain malicious executables. The A-IntExt system extensively reduces the manual effort required to analyze and identify the malware from the reconstructed semantic view of the executables at the VMM. Our empirical results show that the A-IntExt system is capable of providing digital evidence related to identified malware which helps the cyber security practitioner, forensics investigator or system administrator. Its detection proficiency was evaluated by executing large real-world malware and benign executables on the Monitored VMs. The obtained high accuracy of 99.55% with 0.004 FPR on the generated dataset was compared with other benchmarked malware datasets. The experimental results' analysis signified that the A-IntExt system is robust in real-time and practically feasible to work on any public benchmarked datasets with performance overhead of 6.3% over the Windows benchmark suite.

Chapter 7

Execution Time Measurement of Volatile Artifacts Analyzers

7.1 Introduction

Due to a rapid revaluation in a virtualization environment, VMs are a target point for an attacker to gain privileged access of the virtual infrastructure. The APTs such as malware, rootkit, spyware, etc. are more potent to bypass the existing defense mechanisms designed for VM (Jiang et al. 2007). To address this issue, VMI (Garfinkel et al. 2003) emerged as a promising approach that monitors run state of the VM externally from the hypervisor. The main purpose of the VMI is to monitor the true state of the VM without compromising the performance as well as without the knowledge of the VM is an active research topic. Many proposed solutions have adopted the VMI to identify the malignant activities of the VM (Fu and Lin 2013). The LibVMI is able to provide run state of the live VM and also capable to acquire live VM RAM dump. However, the main limitation of the VMI lies with the semantic gap. An open source tool called LibVMI address the semantic gap.

The MFA tool such as Volatility (Ligh et al. 2014) can also be used to address the semantic gap. But, it needs memory dump (RAM) as input. RAM dump capture time and its analysis time in real time are highly crucial if an IDS depends on the data supplied by the MFA tool or VMI tool. Furthermore, memory analyzer accuracy is also a primary for IDS. Thus, our aim is to 1) Measure the time required to capture a live VM RAM dump of the VMI tool. 2) Measure the performance of the MFA tool such as Volatility in terms of execution time elapsed to analyze the RAM dumps of different size. 3) Compare the performance of the Volatility with another open source

MFA tool called Rekall in terms of execution speed. 4) Inject real world rootkits onto the VM in real-time, view the internal shape of the VM using VMI tool, capture the RAM dump and analyze them using Volatility and Rekall separately to appraise the detection accuracy.

In this work, the live VM RAM dump acquire time of LibVMI is measured. In addition, acquired memory dump analysis time consumed by the Volatility is measured and compared with another memory analyzer such as Rekall (Cohen 2014). It is observed through experimental results that, Rekall takes more execution time as compared to Volatility for most of the plugins. Further, Volatility and Rekall are compared with an open source VMI tool called LibVMI (Payne and Bryan 2008). It is noticed that examining the volatile data through LibVMI is faster as it eliminates memory dump acquire time.

7.2 Motivation and Overview of HyIDS

Once the kernel-mode rootkit or advanced persistent malware penetrates into the core of the operating system kernel, they can change the behavior of the legitimate system by arbitrarily modifying the SCT or Interrupt Descriptor Table (IDT) or any other critical kernel code and data structure. It is very difficult to detect such changes if they occur during the run time of the system. Some advanced rootkits or malware are competent enough to bypass or disable the host-based anti-malware defensive solution to evade the detection. One best solution to catch abnormality of the system is by analyzing the RAM content as it provides accurate state of the system. For example, process list, loaded driver modules, SCT, IDT details etc.

In a virtualized environment, one of the best ways of examining the RAM content of the VM is via VMI tool. For example, LibVMI is proficient to provide the internal shape of the target VM like the active process list, module list, event details, etc. In addition, it supports to capture the RAM dump. However, LibVMI is not rich enough to provide more kernel information due to its limited functionalities as on today. The sophisticated DKOM and Semantic Value Manipulation (SVM) attacks based rootkits are more potent to alter the guest kernel data structure. Another way of viewing semantic information of the VM is by analyzing its RAM dump using MFA

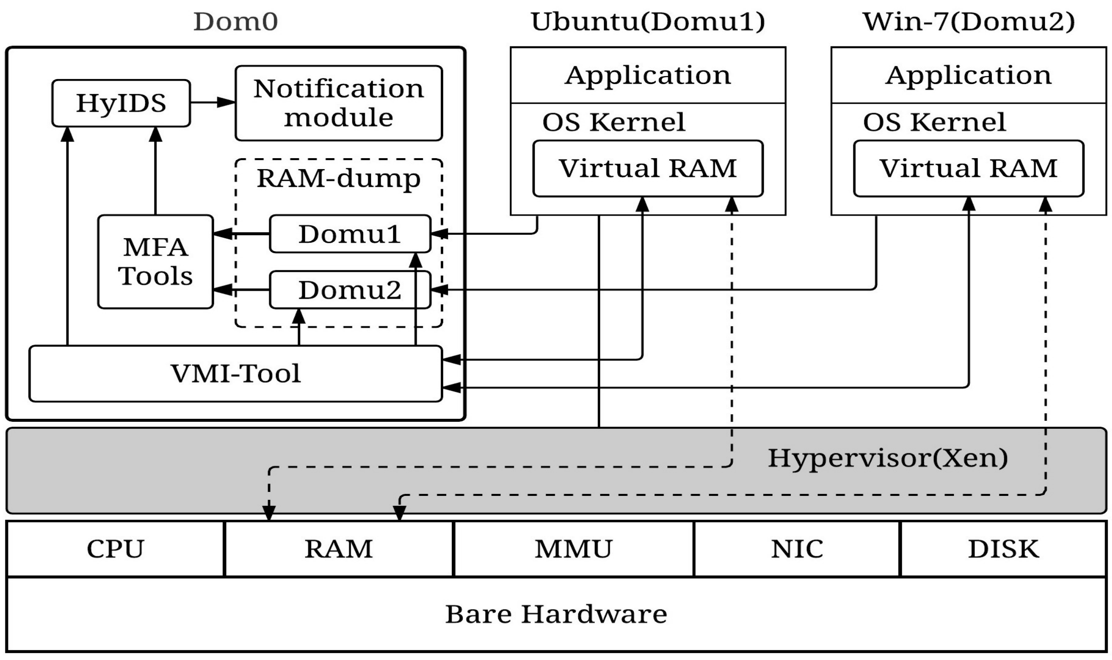


Figure 7.1: High level view of HyIDS architecture for virtualized environment

tool.

Without IDS, only viewing the internal state of the VM either through VMI tool or MFA tool is inadequate to classify the system state as normal or abnormal. Furthermore, without IDS, it is impractical to safeguard the virtualized environment against malware attack or other types of attacks. The prime requirement to safeguard the virtualized environment round the clock is IDS. However, HIDS is an ineffective solution for virtualized environments to thwart advanced malware attacks. Thus, our proposed architectural Hypervisor based Intrusion Detection System (HyIDS) framework is the supreme solution to uncover abnormality of the VM by inspecting volatile data. The HyIDS needs state of the VM to classify the system state as normal or abnormal. If HyIDS depends on the data supplied by the VMI tool or MFA tool, the time needed to fetch the state of the system by the VMI tool or MFA tool plays an important role. Figure 7.1 shows the high-level architectural framework of the HyIDS for a virtualized environment.

The HyIDS receives true state of the VM either by MFA tool or VMI tool. In this scenario, the time required to fetch the real state of the VM by reading volatile data is highly crucial. With this motive in this work, we have measured and compared the execution time of Volatility with Rekall. Further, the speed of LibVMI is compared with Volatility and Rekall. Finally, we figure-out which is the best feasible solution to secure the virtualized environment while addressing so called semantic gap.

7.3 Evaluation and Experimental Results

We have evaluated the execution time of Volatility and Rekall for the different RAM dump size of 1GB, 2GB, and 3GB. Memory dump of both Ubuntu and Windows VMs have captured using LibVMI. The experiments are conducted on the host system which posses the specifications as shown in Table 7.1.

Table 7.1: Experimental setup

Host operating system	Intel (R) core(TM) i7-3770 CPU@ 3.40GHz, 20GB RAM,Ubuntu 14.04 (Trusty tahr) (64-bit)
Hypervisor	Xen 4.4 bare metal hypervisor
Virtual machines	Ubuntu 12.04.3-LTS as Domu1 Windows-7SP0-64x as Domu2
VMI introspection tools	LibVMI version 0.10.1 (Introspect and Capture RAM dumps) LibVMI trap the hardware events and access the vCPU registers
MFA tools	Volatility version 2.4 and Rekall version 1.3.2 (dammastack) employed to examine the captured RAM dump

7.3.1 Detecting Kernel Level Rootkits

To evaluate trustworthiness of live VMI and MFA tools, we have injected real world rootkits on both Windows and Ubuntu guest VMs. We have used seven Linux kernel-mode rootkits such as `Simple rootkit(SR)`, `average coder(AC)`, `Kbeast (KB)`, `chkrootkit-0.50(CK)`, `adore-ng (AD-ng)`, `open-hijack (OH)`, `getpid-hijack (GH)`. Windows operating system based kernel-mode rootkits called `FU rootkit`(FU) and `Hacker Defender`(HD) injected onto Windows-7 VM. Table 7.2 provides rootkits explored in this work with the guest OS on which they were injected. We practically explored that the LibVMI is capable to detect injected rootkits (malicious process ID, hidden modules, etc) on the live running VM. A more semantic information extracted by the Volatility and Rekall on the captured RAM dump of both Windows and Ubuntu monitored guest OS.

Table 7.2: Real-world rootkit experiments under VMs

Rootkits	**OS**	**Functionalities**	**Behavior**
SR, AC, KB	Ubuntu 12.04	lsmod, sys-call, ps, hf	DKSM/SVM
CK, AD-ng	Ubuntu 12.04	Sys-call, ps, mod, strg	DKSM/SVM
OH, GH	Ubuntu 12.04	Sys-call, PID-hijak, ps	DKSM/SVM
HD, FU	Windows-7	Sys-call-hijack, ps, fl	DKSM/SVM

As a first step of rootkit detection, a true run state of the VM viewed using `module-list` plugin of the LibVMI, while working at hypervisor (Dom0). As a proof of experimental results, we have mentioned a snapshot of `average coder` rootkit in the Figure 7.2. The injected rootkit module successfully detected by the LibVMI whereas the same module was unable to view against inspection carried at the VM through `lsmod` command. In Figure 7.2 the background screenshot on the right side shows the output of `module-list` plugin of the LibVM in which inserted rootkit module rootkit is visible whereas same rootkit module is completely hidden against the inspection executed at the infected VM (Domu1) through `lsmod` utility see foreground screenshot on the left side.

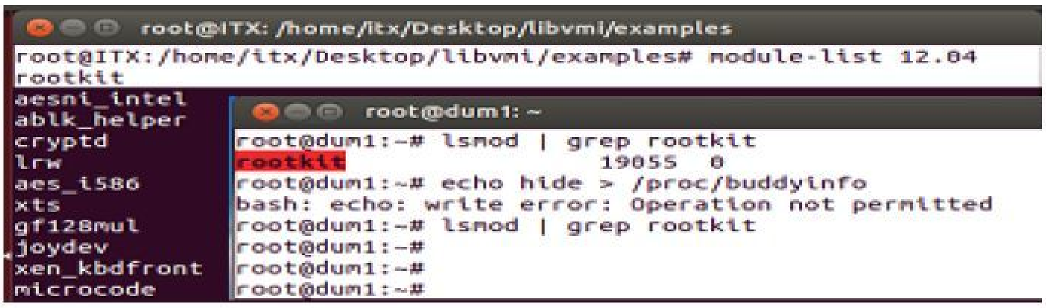
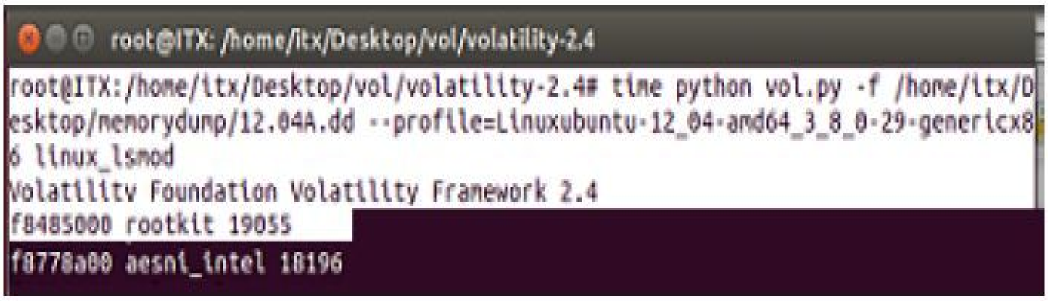

```
root@ITX: /home/itx/Desktop/libvmi/examples
root@ITX:/home/itx/Desktop/libvmi/examples# module-list 12.04
rootkit
aesni_intel
ablk_helper
cryptd
lrw
aes_i586
xts
gf128mul
joydev
xen_kbdfront
microcode
```
```
root@dum1: ~
root@dum1:~# lsmod | grep rootkit
rootkit                19055  0
root@dum1:~# echo hide > /proc/buddyinfo
bash: echo: write error: Operation not permitted
root@dum1:~# lsmod | grep rootkit
root@dum1:~#
root@dum1:~#
root@dum1:~#
```

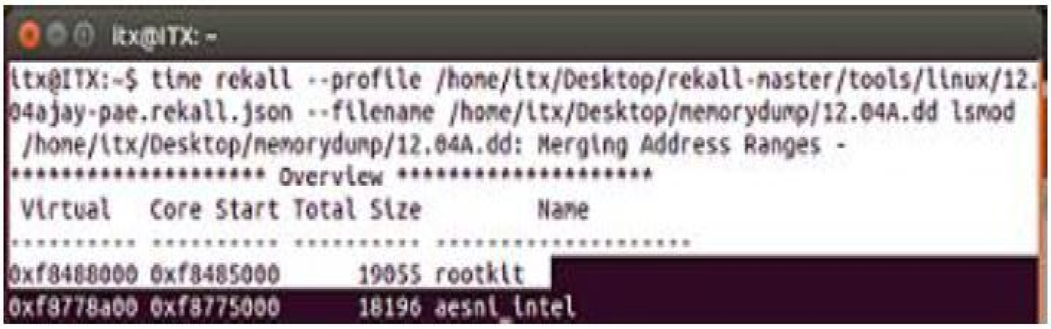
Figure 7.2: `average coder` rootkit module hidden at Domu1 VM the same detected by out-of-the-box VMI solution LibVMI

From Figure 7.3 and Figure 7.4, we can observe that both Volatility and Rekall are capable to report correctly the hidden kernel module of the `average coder` rootkit from RAM dump. The extraction speed of the LibVMI, Volatility and Rekall for `pslist` and `module-list` plugins are tabulated in Table 7.3 and Table 7.4. We can observe that the LibVMI fetching speed is faster as compared to Volatility and Rekall.



```
root@ITX: /home/itx/Desktop/vol/volatility-2.4
root@ITX:/home/itx/Desktop/vol/volatility-2.4# time python vol.py -f /home/itx/D
esktop/memorydump/12.04A.dd --profile=Linuxubuntu-12_04-amd64_3_8_0-29-genericx8
6 linux_lsmod
Volatility Foundation Volatility Framework 2.4
f8485000 rootkit 19055
f8778a00 aesni_intel 18196
```

Figure 7.3: `average coder` rootkit hidden module extracted by the Volatility from raw of physical memory dump



```
itx@ITX: ~
itx@ITX:~$ time rekall --profile /home/itx/Desktop/rekall-master/tools/linux/12.
04ajay-pae.rekall.json --filename /home/itx/Desktop/memorydump/12.04A.dd lsmod
 /home/itx/Desktop/memorydump/12.04A.dd: Merging Address Ranges -
******************** Overview ********************
 Virtual    Core Start Total Size         Name
********* ********* ********* ******************
0xf8488000 0xf8485000      19055 rootkit
0xf8778a00 0xf8775000      18196 aesni_intel
```

Figure 7.4: AC rootkit infected module extracted by the Rekall from raw of physical memory dump

7.3.2 Virtual Machine RAM Dump Analysis using Volatility and Rekall

Volatility is one of the most widely used open source memory forensic tools used to extract digital artifacts from volatile memory (RAM) dumps. It offers a vast number of built-in plugins to investigate different operating system memory dumps. This makes the Volatility to use extensively as a first choice for digital investigation of RAM dumps. OS kernel data structure details are used during analysis time and the details made available to Volatility through the profile. Windows operating system profiles are inbuilt including recent Windows-8.1. In case of Linux based operating systems, Volatility requires the user to create the profile of the respective Linux distribution before the RAM dump analysis. This is due to continuous updation of Linux kernel version.

We have created a profile for Ubuntu 12.04 VM and used the same profile during the experiments. Live Ubuntu 12.04 VM RAM dump of size 1GB, 2GB and 3GB have acquired using LibVMI. The captured RAM dumps have analyzed using the Volatil-

Table 7.3: RAM dump analysis time of the Ubuntu VMs

RAM Dump Size	LibVMI Ubuntu 12.04 (guest OS)		Volatility Ubuntu 12.04 (guest OS)		Rekall Ubuntu 12.04 (guest OS)	
	Process List	**Module List**	**Process List**	**Module List**	**Process List**	**Module List**
1GB	0.30s	0.22s	3.31s	3.69s	5.10s	4.19s
2GB	0.32s	0.29s	3.24s	3.85s	5.85s	4.89s
3GB	0.34s	0.34s	3.98s	4.12s	7.85s	5.01s

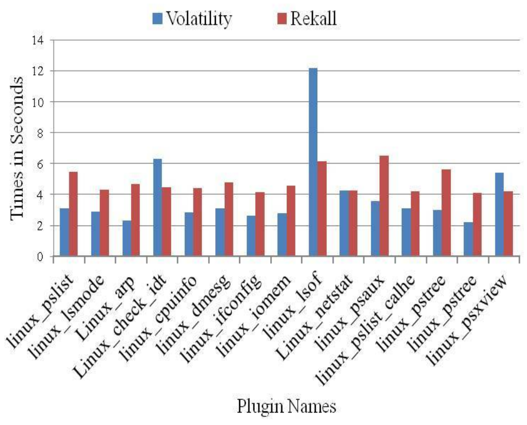
Table 7.4: RAM dump analysis time of Windows VMs

RAM Dump Size	LibVMI Windows-7 (guest OS)		Volatility Windows-7 (guest OS)		Rekall Windows-7 (guest OS)	
	Process List	**Module List**	**Process List**	**Module List**	**Process List**	**Module List**
1GB	0.32s	0.26s	2.25s	2.23s	8.76s	2.92s
2GB	0.38s	0.45s	2.58s	2.64s	10.97s	3.07s
3GB	0.41s	0.58s	2.68s	3.29s	11.80s	7.84s

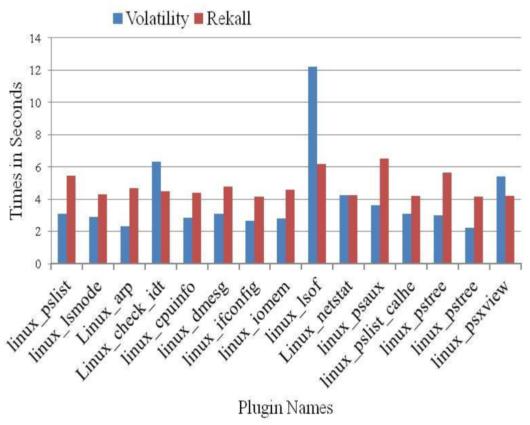
ity Linux plugins such as Linux_pslist, Linux_lsmode, Linux_arp, Linux_psaux, Linux_cpuinfo, Linux_dmesg, Linux_iomem, Linux_lsof, Linux_netstat, Linux_pstree, Linux_pslist_calhe, Linux_check_idt. The same, RAM dumps have also analyzed by another memory analyzer called Rekall. Linux plugins name of Rekall are as same as Volatility Linux plugins name.

We have compared Volatility Linux plugins execution time with Reakll Linux plugins execution time to evaluate the performance in terms of processing time. Figure 7.5a, Figure 7.5b and Figure 7.5c depict execution time taken by Volatility and Rekall for 1GB, 2GB and 3GB RAM dump, respectively. From the experimental results, it is observed that Rekall execution time is more for the following plugins Linux_pslist, Linux_lsmode, Linux_arp, Linux_cpuinfo, Linux_dmesg, Linux_iomem, Linux_netstat, Linux_psaux, Linux_pslist_calhe, Linux_pstree, as compared to Volatility However, Rekall processing time is faster for Linux_check_idt, Linux_lsof, Linux_psview plugins as compared to Volatility.

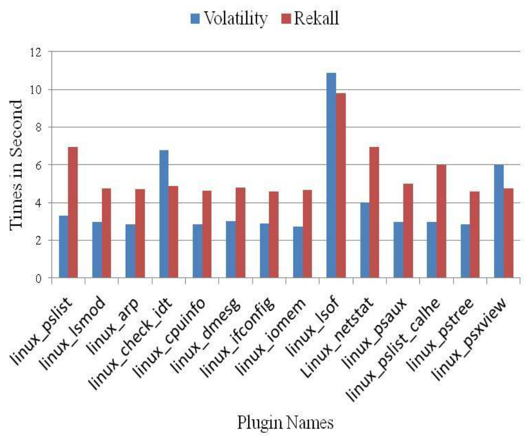
Some of the most common Windows plugins of Volatility and Rekall are tested on Windows-7 VM memory dump of size 1GB, 2GB and 3GB to conduct the experiments. Figure 7.5d, Figure 7.5e and Figure 7.5f depict an execution time taken by



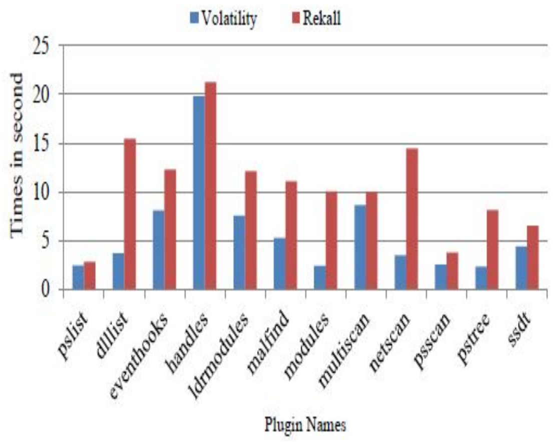
(a) Analysis of Ubuntu 12.04 VM-1GB RAM Dump



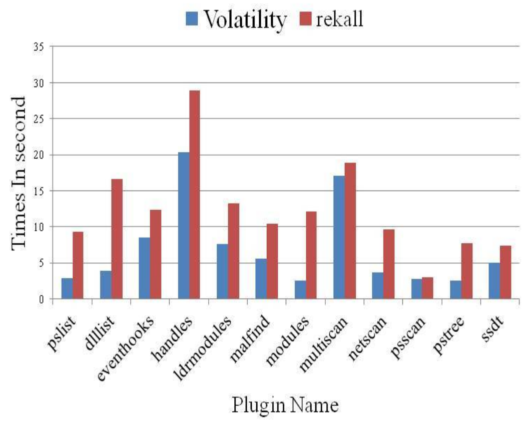
(b) Analysis of Ubuntu 12.04 VM-2GB RAM Dump



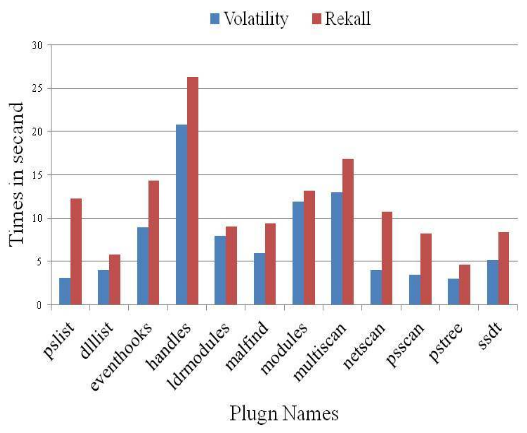
(c) Analysis of Ubuntu 12.04 VM-3GB RAM Dump



(d) Analysis of Windows-7SP0-1GB RAM Dump



(e) Analysis of Windows-7SP0-2GB RAM Dump



(f) Analysis of Windows-7SP0-3GB RAM Dump

Figure 7.5: RAM Dump Analysis Time

Volatility and Rekall for 1GB, 2GB and 3GB RAM dumps, respectively. Our experimental results demonstrate that Rekall takes more time to execute for the following plugins pslist, dlllist, eventhooks, handles, ldmodules, malfind, modules, multiscan, netscan, psscan, pstree, ssdt as compared to Volatility. Another major observation, we found that Volatility reported time for the following plugins Linux_Syscall (110s), Linux_lsof (85s) and Linux_mem(88s) is high compared to other plugins. But, for the same plugins, execution time is drastically reduced in Rekall.

7.3.3 Summary

One way to spot malicious activities of the VM is through viewing run state of the live VM using LibVMI. An alternate way is by analyzing RAM dump of the VM using MFA tool. In this work, the execution speed of the Volatility is measured and compared with Rekall. It is noticed that the Rekall execution speed is slow for most of the plugins as compared to Volatility. Both Volatility and Rekall are capable to address the semantic gap by providing readable information from RAM dump. However, they need memory dump to initiate the analysis. The live VM state information extraction through Volatility and Rekall is slower as compared to LibVMI. However, LibVMI is not matured enough to provide more semantic state information. In other words, currently, LibVMI possessing is limited to few plugins. As there is no memory dump acquire time involved in VMI based approach (LibVMI), the speed of retrieving the data from volatile memory is faster as compared to memory dump based approach (Volatility and Rekall). In this context, the HyIDS gets state information quickly, which helps in determining the intrusions rapidly. As future work, we plan to develop more program module for an existing LibVMI tool to detect intrusions or malware that strengthen virtualized environment.

Chapter 8

Conclusion and Future Work

In this work, we presented VMIDPS and three different malware detection approaches that work at VMM to detect and perform analysis of malware on live introspected guest OS on virtualized cloud computing environment. In the first methodology, the hypervisor and VM dependent based IDPS presented to perform the detection of both Windows and Linux rootkits and other security attacks on Monitored VM. However, this approach uses an agent based solution which functions at each VM that is targeted by the sophisticated rootkit and malware. The limitation of this approach motivated to develop other three VMM-based guest-assisted VM introspection systems which use MFA techniques to perform detection and analysis of malware on live introspected guest OS at VMM. In the second approach, we proposed A-IntExt system that identifies the symptoms of malware execution by scrutinizing the run state information of the of Monitored VMs. The experimental results indicate that proposed approach is proficient in detecting hidden, dead, and malicious processes of any kind of malware or rootkit. Further, it detects and identifies both known and unknown malware processes by performing cross-examination with both local malware database and powerful online malicious content scanners (Virustotal) using computed hashes (MD5, SHA-1, and SHA-256). Since the proposed approach involves static local and online malware check it is infeasible to perform a rapid detection of unknown malware. To address this limitation we further extended the A-IntExt system as AMMDS as the third methodology that performs multilevel-detection of malware by leveraging machine learning techniques at VMM. The OMD and OFMC are two important key components of the AMMDS that perform multi level detection on semantically reconstructed executables. The OFMC implemented with both Odds-ratio and NGL-Correlation Coefficient feature selection techniques. The experiments result demonstrated that AMMDS achieved 100% detection accuracy of malware.

In the fourth methodology, we validated the proposed A-IntExt system by considering larger malware dataset. Since the reconstructed large executables details at the VMM is in dubious form, in order to detect actual malicious executables from the other benign executables, we systematically evaluated the proposed system by generating a dataset of different experimental scenarios at VMM. Further, we have validated the A-IntExt system by considering other public benchmarked malware datasets. In addition, this approach address the `over-fitting` issue by dividing the dataset into training, testing, and validation sets on both generated and benchmarked datasets. The A-IntExt system is implemented by considering popular statistical-based feature selection techniques such as IG, NF, CS, and our HF selection technique. Each feature obtains a score from individual and HF selection technique and selects the topmost best features to the classification techniques. Six machine learning techniques are compared based on the predefined top feature length of the individual feature selection technique. Finally, the Random Forest classifier achieved 99.55% accuracy with 0.004 FPR on the generated dataset. The experimental results' analysis signified that the A-IntExt system is robust in real-time and practically feasible to work on any public benchmarked datasets with performance overhead of 6.3% over the Windows benchmark suite.

As a fifth contribution, the execution speed of Volatility is measured and compared with Rekall. It is noticed that the Rekall execution speed is slow for most of the plugins as compared to Volatility. Both Volatility and Rekall are capable of addressing the semantic gap by providing readable information from RAM dump. Since there is no memory dump acquire time involved in VMI based approach (LibVMI), the speed of retrieving the data from volatile memory is faster as compared to memory dump based approach (Volatility and Rekall). In this context, the HyIDS gets state information quickly, which helps in determining the intrusions rapidly.

In future work, we plan to perform analysis and classification on a group of malware and its related families, which will help the security administrator and forensics analysts to see whether a new malware sample is related to previously known malware and its families. Further, the MFA also facilitates to provide other related artifacts (e.g, registry, API call, files and network details etc.) from the memory dump of the Monitored VM that are most commonly manipulated by sophisticated advanced malware and rootkit. In addition, the obtained artifacts from the introspected live VM memory dump could be used as features to generate a feature vector model to measure the classification of related families of malware using other advanced machine learning techniques. Further, we intend to evaluate the detection proficiency of the A-IntExt systems on other recent versions of Windows OS and Linux-based guest OS in order to measure the detection rate of the proposed approach against the propagation of new variants of malware on specific guest OS.

References

Abou-Assaleh, T., Cercone, N., Keselj, V., and Sweidan, R. (2004). "detection of new malicious code using n-grams signatures". In *PST*, 193–196.

Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., and Giacinto, G. (2016). "novel feature extraction, selection and fusion for effective malware family classification". In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, 183–194. ACM.

Alam, M. S. and Vuong, S. T. (2013). "random forest classification for detecting android malware". In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, 663–669. IEEE.

Alazab, M., Venkatraman, S., Watters, P., and Alazab, M. (2011). "zero-day malware detection based on supervised learning algorithms of api call signatures". In *Proceedings of the Ninth Australasian Data Mining Conference-Volume 121*, 171–182. Australian Computer Society, Inc.

Azab, A. M., Ning, P., Wang, Z., Jiang, X., Zhang, X., and Skalsky, N. C. (2010). "hypersentry: enabling stealthy in-context measurement of hypervisor integrity". In *Proceedings of the 17th ACM conference on Computer and communications security*, 38–49. ACM.

Bahram, S., Jiang, X., Wang, Z., Grace, M., Li, J., Srinivasan, D., Rhee, J., and Xu, D. (2010). "dksm: Subverting virtual machine introspection for fun and profit". In *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, 82–91. IEEE.

Bai, J. and Wang, J. (2016). "improving malware detection using multi-view ensemble learning". *Security and Communication Networks*, *9*(17), 4227–4241.

Bauman, E., Ayoade, G., and Lin, Z. (2015a). A survey on hypervisor based monitoring: Approaches, applications, and evolutions. *ACM Computing Surveys*, *48*(1), 10:1–10:33.

Bauman, E., Ayoade, G., and Lin, Z. (2015b). A survey on hypervisor-based monitoring: approaches, applications, and evolutions. *ACM Computing Surveys (CSUR)*, *48*(1), 10.

Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., and Kruegel, C. (2009). "a view on current malware behaviors". In *LEET*.

Bharadwaja, S., Sun, W., Niamat, M., and Shen, F. (2011). Collabra: a xen hypervisor based collaborative intrusion detection system. In *Information technology: New generations (ITNG), 2011 eighth international conference on*, 695–700. IEEE.

Blum, A. L. and Langley, P. (1997). "selection of relevant features and examples in machine learning". *Artificial intelligence*, *97*(1), 245–271.

Bray, R., Cid, D., and Hay, A. (2008). "ossec host-based intrusion detection". https://ossec.github.io/.

Breiman, L. (2001). "random forests". *Machine learning*, *45*(1), 5–32.

Breiman, L. and Cutler, A. (2017). Random forests [internet].

Butler, J. (2005). Fu rootkit.

Butler, J. and Silberman, P. (2006). "raide: Rootkit analysis identification elimination". *Black Hat USA*, *47*.

Carbone, M., Conover, M., Montague, B., and Lee, W. (2012). "secure and robust monitoring of virtual machines through guest-assisted introspection". In *International Workshop on Recent Advances in Intrusion Detection*, 22–41. Springer.

Case, A., Marziale, L., and Richard, G. G. (2010). "dynamic recreation of kernel data structures for live forensics". *Digital Investigation*, *7*, S32–S40.

Chaâri, R., Ellouze, F., Koubâa, A., Qureshi, B., Pereira, N., Youssef, H., and Tovar, E. (2016). "cyber-physical systems clouds: A survey". *Computer Networks*, *108*, 260–278.

Chen, Z., Xu, G., Mahalingam, V., Ge, L., Nguyen, J., Yu, W., and Lu, C. (2016). "a cloud computing based network monitoring and threat detection system for critical infrastructures". *Big Data Research*, *3*, 10–23.

Chung, C.-J., Khatkar, P., Xing, T., Lee, J., and Huang, D. (2013). "nice: Network intrusion detection and countermeasure selection in virtual network systems". *IEEE transactions on dependable and secure computing*, *10*(4), 198–211.

Cohen, M. (2014). "rekall memory forensics framework". *DFIR Prague*.

Dash, M. and Liu, H. (1997). "feature selection for classification". *Intelligent data analysis*, *1*(1-4), 131–156.

Dave, K. (2011). "study of feature selection algorithms for text-categorization".

David, O. E. and Netanyahu, N. S. (2015). Deepsign: Deep learning for automatic malware signature generation and classification. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, 1–8. IEEE.

Dinaburg, A., Royal, P., Sharif, M., and Lee, W. (2008). "ether: malware analysis via hardware virtualization extensions". In *Proceedings of the 15th ACM conference on Computer and communications security*, 51–62. ACM.

Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W. (2011). "virtuoso: Narrowing the semantic gap in virtual machine introspection". In *2011 IEEE Symposium on Security and Privacy*, 297–312. IEEE.

Dolan-Gavitt, B., Payne, B., and Lee, W. (2011). "leveraging forensic tools for virtual machine introspection".

Domingos, P. and Pazzani, M. (1997). "on the optimality of the simple bayesian classifier under zero-one loss". *Machine learning*, *29*(2-3), 103–130.

Egan, J. P. (1975). "signal detection theory and roc analysis".

F-Secure (2003). F-secure. f-secure virus descriptions: Agobot.

Ferrie, P. (2007). "attacks on more virtual machine emulators", 1–13.

Florio, E. (2005). "when malware meets rootkits". *Virus Bulletin*, *12*.

Fu, Y. and Lin, Z. (2013). "bridging the semantic gap in virtual machine introspection via online kernel data redirection". *ACM Transactions on Information and System Security (TISSEC)*, *16*(2), 7.

Fu, Y., Zeng, J., and Lin, Z. (2014). "hypershell: a practical hypervisor layer guest os shell for automated in-vm management". In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 85–96.

Garfinkel, T., Rosenblum, M., et al. (2003). "a virtual machine introspection based architecture for intrusion detection". In *NDSS*, volume 3, 191–206.

Garnaeva, M. (2012). "kelihos/hlux botnet returns with new techniques". *Securelist, http://www. securelist. com/en/blog/655/Kelihos_Hlux_botnet_ returns_with_new_techniques*.

GmbH, I. (2007). "oracle vm virtual box". https://www.virtualbox.org/.

Goudey (2012a). "threat report: Rootkits".

Goudey, H. (2012b). Microsoft malware protection center, threat report: Rootkits. Technical report, Tech. rep., Microsoft Corporation, June 2012. http://www. microsoft. com/en-us/download/confirmation. aspx.

Gu, Z., Deng, Z., Xu, D., and Jiang, X. (2011). "process implanting: A new active introspection framework for virtualization". In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, 147–156. IEEE.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). "the weka data mining software: an update". *ACM SIGKDD explorations newsletter*, *11*(1), 10–18.

Han, J., Pei, J., and Kamber, M. (2011). *Data mining: concepts and techniques*. Elsevier.

Hay, B. and Nance, K. (2008). "forensics examination of volatile system data using virtual introspection". *ACM SIGOPS Operating Systems Review*, *42*(3), 74–82.

Hellal, A. and Romdhane, L. B. (2016). "minimal contrast frequent pattern mining for malware detection". *Computers & Security*, *62*, 19–32.

Ho, T. K. (1998). "the random subspace method for constructing decision forests". *IEEE transactions on pattern analysis and machine intelligence*, *20*(8), 832–844.

Horne, C. (2007). Understanding full virtualization, paravirtualization and hardware assist. *White paper, VMware Inc.*

Huda, S., Miah, S., Hassan, M. M., Islam, R., Yearwood, J., Alrubaian, M., and Almogren, A. (2017). "defending unknown attacks on cyber-physical systems by semi-supervised approach and available unlabeled data". *Information Sciences*, *379*, 211–228.

Hwang, T., Shin, Y., Son, K., and Park, H. (2013). Design of a hypervisor-based rootkit detection method for virtualized systems in cloud computing environments. In *Proceedings of the 2013 AASRI Winter International Conference on Engineering and Technology*, 27–32.

Intel (2016). "intel trusted exceution technology". In *Accessed on September 2016.*

Islam, R., Tian, R., Batten, L. M., and Versteeg, S. (2013). "classification of malware based on integrated static and dynamic features". *Journal of Network and Computer Applications*, *36*(2), 646–656.

Jablkowski, B., Gabor, U. T., and Spinczyk, O. (2017). "evolutionary planning of virtualized cyber-physical compute and control clusters". *Journal of Systems Architecture*, *73*, 17–27.

Jablkowski, B. and Spinczyk, O. (2015). "cps-xen: A virtual execution environment for cyber-physical applications". In *International Conference on Architecture of Computing Systems*, 108–119. Springer.

Jain, B., Baig, M. B., Zhang, D., Porter, D. E., and Sion, R. (2014). "sok: Introspections on trust and the semantic gap". In *2014 IEEE Symposium on Security and Privacy*, 605–620. IEEE.

James, b. (2010). Security and privacy challenges in cloud computing environments.

Jiang, X., Wang, X., and Xu, D. (2007). "stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction". In *Proceedings of the 14th ACM conference on Computer and communications security*, 128–138. ACM.

Jin, S., Ahn, J., Cha, S., and Huh, J. (2011). Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 272–283. ACM.

Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2006). "antfarm: Tracking processes in a virtual machine environmen". In *USENIX Annual Technical Conference, General Track*, 1–14.

Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2008). "vmm-based hidden process detection and identification using lycosid". In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 91–100. ACM.

Kearns, M., Mansour, Y., Ng, A. Y., and Ron, D. (1997). "an experimental and theoretical comparison of model selection methods". *Machine Learning*, *27*(1), 7–50.

Kim, S., Park, J., Lee, K., You, I., and Yim, K. (2012). A brief survey on rootkit techniques in malicious codes. *J. Internet Serv. Inf. Secur.*, *2*(3/4), 134–147.

King, S. T. and Chen, P. M. (2006). "subvirt: Implementing malware with virtual machines". In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, 14–pp. IEEE.

Kolter, J. Z. and Maloof, M. A. (2006). "learning to detect and classify malicious executables in the wild". *Journal of Machine Learning Research*, *7*(Dec), 2721–2744.

Kumar, A., Kuppusamy, K., and Aghila, G. (2017). "a learning model to detect maliciousness of portable executable using integrated feature set". *Journal of King Saud University-Computer and Information Sciences*.

Kwak, N. and Choi, C.-H. (2002). "input feature selection for classification problems". *IEEE Transactions on Neural Networks*, *13*(1), 143–159.

Lamps, J., Palmer, I., and Sprabery, R. (2014). "winwizard: Expanding xen with a libvmi intrusion detection tool". In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, 849–856. IEEE.

Langley, P. et al. (1994). "selection of relevant features in machine learning". In *Proceedings of the AAAI Fall symposium on relevance*, volume 184, 245–271.

Lengyel, T. K., Maresca, S., Payne, B. D., Webster, G. D., Vogl, S., and Kiayias, A. (2014). "scalability, fidelity and stealth in the drakvuf dynamic malware analysis system". In *Proceedings of the 30th Annual Computer Security Applications Conference*, 386–395. ACM.

Liangboonprakong, C. and Sornil, O. (2013). "classification of malware families based on n-grams sequential pattern features". In *Industrial Electronics and Applications (ICIEA), 2013 8th IEEE Conference on*, 777–782. IEEE.

Ligh, M. H., Case, A., Levy, J., and Walters, A. (2014). *"The art of memory forensics: detecting malware and threats in Windows, Linux, and Mac memory"*. John Wiley & Sons.

Lin, D. and Stamp, M. (2011). "hunting for undetectable metamorphic viruses". *Journal in computer virology*, *7*(3), 201–214.

Liston, T. and Skoudis, E. (2006). "on the cutting edge: Thwarting virtual machine detection".

Litty, L., Lagar-Cavilla, H. A., and Lie, D. (2008). "hypervisor support for identifying covertly executing binaries". In *USENIX Security Symposium*, 243–258.

Liu, Y., Xia, Y., Guan, H., Zang, B., and Chen, H. (2014). Concurrent and consistent virtual machine introspection with hardware transactional memory. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, 416–427. IEEE.

Ma, J., Choo, K.-K. R., Hsu, H.-h., Jin, Q., Liu, W., Wang, K., Wang, Y., and Zhou, X. (2016). "perspectives on cyber science and technology for cyberization and cyber-enabled worlds". In *Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, 1–9. IEEE.

Martignoni, L., Fattori, A., Paleari, R., and Cavallaro, L. (2010). "live and trustworthy forensic analysis of commodity production systems". In *International Workshop on Recent Advances in Intrusion Detection*, 297–316. Springer.

Masud, M. M., Khan, L., and Thuraisingham, B. (2008). "a scalable multi-level feature extraction technique to detect malicious executables". *Information Systems Frontiers*, *10*(1), 33–45.

Matthews, B. W. (1975). "comparison of the predicted and observed secondary structure of t4 phage lysozyme". *Biochimica et Biophysica Acta (BBA)-Protein Structure*, *405*(2), 442–451.

Menahem, E., Shabtai, A., Rokach, L., and Elovici, Y. (2009). "improving malware detection by applying multi-inducer ensemble". *Computational Statistics & Data Analysis*, *53*(4), 1483–1494.

Miao, Q., Liu, J., Cao, Y., and Song, J. (2016). "malware detection using bilayer behavior abstraction and improved one-class support vector machines". *International Journal of Information Security*, *15*(4), 361–379.

Mitchell, T. et al. (1997). "machine learning. wcb".

Mladenic, D. and Grobelnik, M. (1999). "feature selection for unbalanced class distribution and naive bayes". In *ICML*, volume 99, 258–267.

Modi, C., Patel, D., Borisaniya, B., Patel, H., Patel, A., and Rajarajan, M. (2013). A survey of intrusion detection techniques in cloud. *Journal of Network and Computer Applications*, *36*(1), 42–57.

Moser, A., Kruegel, C., and Kirda, E. (2007). "limits of static analysis for malware detection". In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, 421–430. IEEE.

Moskovitch, R., Elovici, Y., and Rokach, L. (2008). "detection of unknown computer worms based on behavioral classification of the host". *Computational Statistics & Data Analysis*, *52*(9), 4544–4566.

Musavi, S. A. and Kharrazi, M. (2014). Back to static analysis for kernel-level rootkit detection. *IEEE Transactions on Information Forensics and Security*, *9*(9), 1465–1476.

Nagarajan, A. B., Mueller, F., Engelmann, C., and Scott, S. L. (2007). "proactive fault tolerance for hpc with xen virtualization". In *Proceedings of the 21st annual international conference on Supercomputing*, 23–32. ACM.

Nance, K., Bishop, M., and Hay, B. (2009). Investigating the implications of virtual machine introspection for digital forensics. In *Availability, Reliability and Security, 2009. ARES'09. International Conference on*, 1024–1029. IEEE.

Ng, A. Y. (1997). "preventing overfitting of cross-validation data". In *ICML*, volume 97, 245–253.

Nguyen, A. M., Schear, N., Jung, H., Godiyal, A., King, S. T., and Nguyen, H. D. (2009). Mavmm: Lightweight and purpose built vmm for malware analysis. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, 441–450. IEEE.

Nikolai, J. and Wang, Y. (2014). Hypervisor-based cloud intrusion detection system. In *Computing, Networking and Communications (ICNC), 2014 International Conference on*, 989–993. IEEE.

Nissim, N., Moskovitch, R., Rokach, L., and Elovici, Y. (2012). "detecting unknown computer worm activity via support vector machines and active learning". *Pattern Analysis and Applications*, *15*(4), 459–475.

Oshiro, T. M., Perez, P. S., and Baranauskas, J. A. (2012). "how many trees in a random forest". In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, 154–168. Springer.

Ozsoy, M., Khasawneh, K. N., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., and Ponomarev, D. (2016). "hardware-based malware detection using low-level architectural features". *IEEE Transactions on Computers*, *65*(11), 3332–3344.

Patel, A., Taghavi, M., Bakhtiyari, K., and Júnior, J. C. (2012). Taxonomy and proposed architecture of intrusion detection and prevention systems for cloud computing. In *CSS*, 441–458. Springer.

Payne and Bryan (2008). "libvmi introduction: Vmitools, an introduction to libvmi". `http://libvmi.com/`.

Payne, B. D., Carbone, M., Sharif, M., and Lee, W. (2008). "lares: An architecture for secure active monitoring using virtualization". In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, 233–247. IEEE.

Payne, B. D., De Carbone, M., and Lee, W. (2007). "secure and flexible monitoring of virtual machines". In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, 385–397. IEEE.

Pearce, M., Zeadally, S., and Hunt, R. (2013). Virtualization: Issues, security threats, and solutions. *ACM Computing Surveys (CSUR)*, *45*(2), 17.

Perdisci, R., Lanzi, A., and Lee, W. (2008). "mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables". In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, 301–310. IEEE.

Pfoh, J., Schneider, C., and Eckert, C. (2009). A formal model for virtual machine introspection. In *Proceedings of the 1st ACM workshop on Virtual machine security*, 1–10. ACM.

Pietrek, M. (1994). "peering inside the pe: a tour of the win32 (r) portable executable file format". *Microsoft Systems Journal-US Edition*, 15–38.

Platt, J. C. (1999). "12 fast training of support vector machines using sequential minimal optimization". *Advances in kernel methods*, 185–208.

Poisel, R., Malzer, E., and Tjoa, S. (2013). "evidence and cloud computing: The virtual machine introspection approach". *JoWUA*, *4*(1), 135–152.

Poore, J., Flores, J. C., and Atkison, T. (2013). Evolution of digital forensics in virtualization by using virtual machine introspection. In *Proceedings of the 51st ACM Southeast Conference*, 30. ACM.

Prakash, A., Venkataramani, E., Yin, H., and Lin, Z. (2013). Manipulating semantic values in kernel data structures: Attack assessments and implications. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, 1–12. IEEE.

Prakash, A., Venkataramani, E., Yin, H., and Lin, Z. (2015). On the trustworthiness of memory analysis-an empirical study from the perspective of binary execution. *IEEE Transactions on Dependable and Secure Computing*, *12*(5), 557–570.

Quinlan, J. R. (1986). "induction of decision trees". *Machine learning*, *1*(1), 81–106.

Raff, E., Zak, R., Cox, R., Sylvester, J., Yacci, P., Ward, R., Tracy, A., McLean, M., and Nicholas, C. (2016). "an investigation of byte n-gram features for malware classification". *Journal of Computer Virology and Hacking Techniques*, 1–20.

Raffetseder, T., Kruegel, C., and Kirda, E. (2007). "detecting system emulators". In *International Conference on Information Security*, 1–18. Springer.

Reddy, D. K. S., Dash, S. K., and Pujari, A. K. (2006). "new malicious code detection using variable length n-grams". In *International Conference on Information Systems Security*, 276–288. Springer.

Reddy, D. K. S. and Pujari, A. K. (2006). "n-gram analysis for computer virus detection". *Journal in Computer Virology*, *2*(3), 231–239.

Reddy, Y. B. (2015). "security and design challenges in cyber-physical systems". In *Information Technology-New Generations (ITNG), 2015 12th International Conference on*, 200–205. IEEE.

Rhee, J., Riley, R., Xu, D., and Jiang, X. (2009). "defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring". In *Availability, Reliability and Security, 2009. ARES'09. International Conference on*, 74–81. IEEE.

Richer, T. J., Neale, G., and Osborne, G. (2015). "on the effectiveness of virtualisation assisted view comparison for rootkit detection". In *Proceedings of the 13th Australasian Information Security Conference (AISC 2015)*, volume 27, 30.

Rieck, K., Holz, T., Willems, C., Düssel, P., and Laskov, P. (2008). "learning and classification of malware behavior". In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 108–125. Springer.

Rieck, K., Trinius, P., Willems, C., and Holz, T. (2011). "automatic analysis of malware behavior using machine learning". *Journal of Computer Security*, *19*(4), 639–668.

Rokach, L., Chizi, B., and Maimon, O. (2007). "a methodology for improving the performance of non-ranker feature selection filters". *International Journal of Pattern Recognition and Artificial Intelligence*, *21*(05), 809–830.

Russinovich, M. E., Solomon, D. A., and Ionescu, A. (2012). *"Windows internals"*. Pearson Education.

Rutkowska, J. (2006). "introducing blue pill". *The official blog of the invisiblethings. org*, *22*.

Rutkowska, J. and Tereshkin, A. (2008). "bluepilling the xen hypervisor". *Black Hat USA*.

Saberi, A., Fu, Y., and Lin, Z. (2014). "hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization". In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*.

Saleh, M., Ratazzi, E. P., and Xu, S. (2014). "instructions-based detection of sophisticated obfuscation and packing". In *2014 IEEE Military Communications Conference*, 1–6. IEEE.

Salzberg, S. L. (1994). "c4. 5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993". *Machine Learning*, *16*(3), 235–240.

Santos, I., Brezo, F., Ugarte-Pedrero, X., and Bringas, P. G. (2013). "opcode sequences as representation of executables for data-mining-based unknown malware detection". *Information Sciences*, *231*, 64–82.

Scarfone, K. and Mell, P. (2007). "guide to intrusion detection and prevention systems (idps)". *NIST special publication*, *800*(2007), 94.

Schmidt, M., Baumgartner, L., Graubner, P., Bock, D., and Freisleben, B. (2011). Malware detection and kernel rootkit prevention in cloud computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, 603–610. IEEE.

Schultz, M. G., Eskin, E., Zadok, F., and Stolfo, S. J. (2001). "data mining methods for detection of new malicious executables". In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, 38–49. IEEE.

Shabtai, A., Moskovitch, R., Elovici, Y., and Glezer, C. (2009). "detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey". *information security technical report*, *14*(1), 16–29.

Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., and Elovici, Y. (2012). "detecting unknown malicious code by applying classification techniques on opcode patterns". *Security Informatics*, *1*(1), 1.

Shafiq, M. Z., Tabish, S. M., Mirza, F., and Farooq, M. (2009). "pe-miner: Mining structural information to detect malicious executables in realtime". In *International Workshop on Recent Advances in Intrusion Detection*, 121–141. Springer.

Shahzad, F., Shahzad, M., and Farooq, M. (2013). "in-execution dynamic malware analysis and detection by mining information in process control blocks of linux os". *Information Sciences*, *231*, 45–63.

Shan, Z. and Wang, X. (2014). "growing grapes in your computer to defend against malware". *IEEE Transactions on Information Forensics and Security*, *9*(2), 196–207.

Sharif, M., Lanzi, A., Giffin, J., and Lee, W. (2009). "automatic reverse engineering of malware emulators". In *Security and Privacy, 2009 30th IEEE Symposium on*, 94–109. IEEE.

Sharif, M. I., Lee, W., Cui, W., and Lanzi, A. (2009). "secure in-vm monitoring using hardware virtualization". In *Proceedings of the 16th ACM conference on Computer and communications security*, 477–487. ACM.

Shaw, A. L., Bordbar, B., Saxon, J., Harrison, K., and Dalton, C. I. (2014). Forensic virtual machines: dynamic defence in the cloud via introspection. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, 303–310. IEEE.

Shevchenko, A. (2007). "the evolution of self-defense technologies in malware". *Available from webpage: http://www. viruslist. com/analysis.*

Shi, J., Yang, Y., Li, C., and Wang, X. (2015). Spems: A stealthy and practical execution monitoring system based on vmi. In *International Conference on Cloud Computing and Security*, 380–389. Springer.

Sparks, S. and Butler, J. (2005). "shadow walker: Raising the bar for rootkit detection". *Black Hat Japan*, *11*(63), 504–533.

Srinivasan, D., Wang, Z., Jiang, X., and Xu, D. (2011). "process out-grafting: an efficient out-of-vm approach for fine-grained process execution monitoring". In *Proceedings of the 18th ACM conference on Computer and communications security*, 363–374. ACM.

Stüttgen, J., Vömel, S., and Denzel, M. (2015). Acquisition and analysis of compromised firmware using memory forensics. *Digital Investigation*, *12*, S50–S60.

Sung, A. H., Xu, J., Chavez, P., and Mukkamala, S. (2004). "static analyzer of vicious executables (save)". In *Computer Security Applications Conference, 2004. 20th Annual*, 326–334. IEEE.

Symantec (2016). Internet security threat report.

Takabi, H., Joshi, J. B., and Ahn, G.-J. (2010). Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy*, *8*(6), 24–31.

Team, T. P. S. (1998). "packet storm security". https://packetstormsecurity.com/.

Valipour, M. (2016). "optimization of neural networks for precipitation analysis in a humid region to detect drought and wet year alarms". *Meteorological Applications*, *23*(1), 91–100.

Vollmar, W., Harris, T., Long, L., and Green, R. (2014). Hypervisor security in cloud computing systems. *ACM Comput. Surv*, 1–22.

Wang, X. and Karri, R. (2013). Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *Proceedings of the 50th Annual Design Automation Conference*, 79. ACM.

Wang, Y.-M., Beck, D., Vo, B., Roussev, R., and Verbowski, C. (2005). "detecting stealth software with strider ghostbuster". In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, 368–377. IEEE.

Watson, M. R., Marnerides, A. K., Mauthe, A., Hutchison, D., et al. (2016). "malware detection in cloud computing infrastructures". *IEEE Transactions on Dependable and Secure Computing*, *13*(2), 192–205.

Wesley Vollmar, Thomas Harris, L. L. J. and Green, R. (2014). "hypervisor security in cloud computing systems". *ACM Computing Surveys*, 1–22.

Westphal, F., Axelsson, S., Neuhaus, C., and Polze, A. (2014). Vmi-pl: A monitoring language for virtual platforms using virtual machine introspection. *Digital Investigation*, *11*, S85–S94.

Willems, C., Holz, T., and Freiling, F. (2007). "toward automated dynamic malware analysis using cwsandbox". *IEEE Security & Privacy*, *5*(2).

Willems, C., Hund, R., and Holz, T. (2013). Cxpinspector: Hypervisor-based, hardware-assisted system monitoring. *Ruhr-Universitat Bochum, Tech. Rep*, 12.

Witten, I. H. and Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

Wojtczuk, R. (2008). "subverting the xen hypervisor". *Black Hat USA, 2008*.

Wojtczuk, R. and Rutkowska, J. (2009). "attacking intel trusted execution technology". *Black Hat DC, 2009*.

Xie, X. and Wang, W. (2013). Rootkit detection on virtual machines through deep information extraction at hypervisor-level. In *Communications and Network Security (CNS), 2013 IEEE Conference on*, 498–503. IEEE.

Xuan, C., Copeland, J. A., and Beyah, R. A. (2009). "toward revealing kernel malware behavior in virtual execution environments". In *RAID*, volume 9, 304–325. Springer.

Yan, L.-K., Jayachandra, M., Zhang, M., and Yin, H. (2012). V2e: combining hardware virtualization and softwareemulation for transparent and extensible malware analysis. *ACM Sigplan Notices*, *47*(7), 227–238.

Yang, Y. and Pedersen, J. O. (1997). "a comparative study on feature selection in text categorization". In *ICML*, volume 97, 412–420.

Zawoad, S. and Hasan, R. (2013). Cloud forensics: a meta-study of challenges, approaches, and open problems. *arXiv preprint arXiv:1302.6312*.

Zhang, Y., Huang, Q., Ma, X., Yang, Z., and Jiang, J. (2016). "using multi-features and ensemble learning method for imbalanced malware classification". In *Trustcom/BigDataSE/ISPA, 2016 IEEE*, 965–973. IEEE.

Zhao, X., Borders, K., and Prakash, A. (2009). Virtual machine security systems. *Advances in Computer Science and Engineering*, *1*, 339–365.

Zhong, X., Xiang, C., Yu, M., Qi, Z., and Guan, H. (2015). A virtualization based monitoring system for mini-intrusive live forensics. *International Journal of Parallel Programming*, *43*(3), 455–471.

List of Publications Based on Dissertation

Journals

1. Ajay Kumara. M.A, and Jaidhar C.D., "*Automated Multi-level Malware Detection System based on Reconstructed Semantic View of Executables using Machine Learning Techniques at VMM*". The International Journal of Future Generation Computer Systems, Elsevier, Volume 79, Part 1, Pages 431-446, February 2018.

2. Ajay Kumara. M.A, and Jaidhar C.D., "*Leveraging virtual machine introspection with memory forensics to detect and characterize unknown malware using machine learning techniques at hypervisor*". The international journal of Digital investigation, Elsevier, Volume 23, Pages 99-123, December 2017.

Conferences

1. Ajay Kumara M.A and Jaidhar.C.D., "Hypervisor and Virtual Machine Dependent Intrusion Detection and Prevention System for Virtualized Cloud Computing Environment". In proceedings of the 1st IEEE International Conference on Telematics and Future Generation Network (TAFGEN-2015) University Technology Malaysia, Pages 28-33.

2. Ajay Kumara M.A and Jaidhar.C.D., "Virtual Machine Introspection based Spurious Process Detection in Virtualized Cloud Computing Environment". In proceedings of the 1st International Conference on Futuristic Trends on Computational Analysis and Knowledge Management, Feb 25-27, 2015. Pages 309-315.

3. Ajay Kumara M.A and Jaidhar.C.D., "VMI Based Automated Real-Time Malware Detector for Virtualized Cloud Environment". In proceedings of the 6th International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE-2016). Pages 281-300.

4. Ajay Kumara M.A and Jaidhar.C.D., "Execution Time Measurement of Virtual Machine Volatile Artifacts Analyzers". In proceedings of the 21st IEEE International Conference on Parallel and Distributed Systems (ICPADS 2015). Pages 314-319.

Brief Bio-Data

AJAY KUMARA M.A

Full-Time Ph.D Research Scholar,

Department of Information Technology

National Institute of Technology Karnataka, Surathkal

P.O. Srinivasnagar

Mangalore, 575025

Email: ajaykumar.ak99@gmail.com

Permanent Address

Ajay Kumara M.A. S/O Annaiah

Makanahalli (Village), Periyapattana (Taluk)

Mysore District

Karnataka State, India

Mangalore, 571108

Qualification

M. Tech. Computer Science and Engineering, Visvesvaraya Technological University, Belgaum, Karnataka, 2012.

B. E. Computer Science and Engineering,Visvesvaraya Technological University, Belgaum, Karnataka, 2005.