LEARNING AT THE VIRTUALIZATION LAYER:
INTRUSION DETECTION AND
WORKLOAD CHARACTERIZATION
FROM WITHIN THE VIRTUAL MACHINE MONITOR

A Thesis Presented
by
Fatemeh Azmandian
to
The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in
Electrical Engineering
in the field of
Computer Engineering
Northeastern University
Boston, Massachusetts
August 2012
© Copyright 2012 by Fatemeh Azmandian
All Rights Reserved
Abstract

Virtualization technology has many attractive qualities including improved reliability, scalability, and resource sharing/management. As a result, virtualization has been deployed on an array of platforms, from mobile devices to high-end enterprise servers. In this work, we contribute to the benefits of virtualization by providing two key features for virtual machines (VMs): enhanced security using a novel intrusion detection system and workload characterization for virtual machine workloads. What makes our contributions unique is the fact that we only make use of low-level architectural data available at the virtual machine monitor (VMM) layer. This gives rise to several advantages, including a reduction in the overhead introduced into the system, as only the VMM need be modified, and ease of deployment since there are no ties to a specific OS and deployment can occur transparently below different operating systems. In addition, it limits the perturbation introduced into the system, thereby reducing the Observer Effect where the phenomena under observation is altered or lost due to the measurement itself.

The low-level VMM-layer data, in itself, lacks the semantic information available at higher computing abstraction layers, such as the application layer or operating system layer. Only with the right set of tools is it possible to realize the richness hidden within the raw data. Thus, we take the approach of learning at the VMM
layer; we apply machine learning and data mining techniques to understand what it means for an execution stream to be identified as “normal”. Then we can flag deviations from normal as suspicious activity, signaling the presence of malware, as well as break down normal behavior into its constituent parts corresponding to prevalent components of a computer system.

Our experiments on over 300 real-world malware and exploits illustrate that there is sufficient information embedded within the VMM-level data to allow accurate detection of malicious attacks, with an acceptable false alarm rate. In this thesis, we also demonstrate that the information available at the VMM level still retains rich workload characteristics that can be used to identify application behavior. We show that we are able to capture enough information about a workload to characterize and decompose it into a combination of CPU, memory, disk I/O, and network I/O-intensive components. Dissecting the behavior of a workload in terms of these components, we can develop significant insight into the behavior of any application.

Finally, in this thesis we propose a novel feature selection algorithm designed to facilitate the process of identifying outliers. It is the first of its kind to tackle the difficult task of selecting features suitable for outlier detection problems. With its opportunities for parallelism, the algorithm becomes an excellent candidate for implementation on a graphics processing unit (GPU). Through the acceleration provided by general purpose computing on a GPU (GPGPU), we demonstrate the benefits of utilizing the proposed approach over popular and state-of-the-art feature selection techniques, and its high applicability to large datasets.
Acknowledgements

In the name of God, the Most Beneficent, the Most Merciful

I dedicate this thesis to my daughter, Ava. You fill my heart with pure love and indescribable joy. For the rest of my life, the title that I am most proud of is Mom. To my husband, for your unwavering encouragement, patience with me and my long hours at the lab, and for being a great husband and father. To my parents, who are my inspiration and role models. You have shaped who I am in every way and were it not for you, I would not be where I am today. To my brother and sister for being my personal cheerleading squad and to my grandparents who have enriched my life with their stories full of valuable lessons to be learned.

My greatest thanks to my research advisor, Dr. David Kaeli, for always offering your support, wisdom, and knowledge of how to enhance my work and propel it to a higher level. Thank you for believing in me and seeing my potential from the beginning. I would also like to thank Dr. Jennifer Dy and Dr. Jay Aslam for their invaluable help throughout my research. Thank you for guiding me along my journey. I have learned so much from all of you and for this, I owe you my sincerest gratitude.
Contents

Abstract iii

Acknowledgements v

1 Introduction 1

1.1 OS Virtualization Methods ................................. 2
1.2 Advantages of Virtualization ................................. 3
1.3 Challenges of Working Purely at the Virtual Machine Monitor Layer ................................. 5
1.4 Contributions of the Work ................................. 11
1.5 Organization of Thesis ................................. 12

2 Background 14

2.1 Intrusion Detection ........................................... 14
2.2 Workload Characterization .................................... 16
2.3 VMM-Level Event Extraction .................................. 17
2.4 Feature Construction ........................................... 20
  2.4.1 Rate Features ........................................... 22
  2.4.2 Relationship Features .................................... 22
2.5 Anomaly Detection Algorithms .................................. 25
2.5.1  K-Nearest Neighbors .............................................. 26
2.5.2  Local Outlier Factor .............................................. 27
2.6  Regression Algorithms .............................................. 29
  2.6.1  Linear Least-Square Regression .............................. 29
  2.6.2  Least Absolute Shrinkage and Selection Operator ........ 30
2.7  GPU and CUDA Background ....................................... 31

3  Related Work .......................................................... 34
  3.1  Related Work in Intrusion Detection ............................ 34
    3.1.1  Program-Level IDS ........................................... 35
    3.1.2  Operating System-Level IDS ................................ 35
    3.1.3  Virtual Machine Monitor-Level IDS ....................... 37
    3.1.4  IDS Comparison ............................................... 39
    3.1.5  Data Mining for Security .................................... 40
  3.2  Related Work in Workload Characterization ................... 41
  3.3  Related Work in Feature Selection .............................. 44

4  Intrusion Detection with the VMM IDS ............................. 47
  4.1  VMM IDS Front-End ............................................... 52
  4.2  VMM IDS Back-End ............................................... 53
    4.2.1  Feature Reduction .......................................... 53
    4.2.2  Model Creation and Anomaly Detection .................... 55
    4.2.3  Raising an Alarm ........................................... 56
  4.3  Evaluation ......................................................... 60
    4.3.1  Experimental Setup ......................................... 60
    4.3.2  Feature Analysis ............................................ 65
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.3</td>
<td>Experimental Results</td>
<td>70</td>
</tr>
<tr>
<td>4.4</td>
<td>Discussion</td>
<td>75</td>
</tr>
<tr>
<td>4.5</td>
<td>VMM IDS Conclusions</td>
<td>78</td>
</tr>
<tr>
<td>5</td>
<td>Workload Characterization with the VMM Application Modeler</td>
<td>79</td>
</tr>
<tr>
<td>5.1</td>
<td>VMM Application Modeler Back-End</td>
<td>83</td>
</tr>
<tr>
<td>5.2</td>
<td>Evaluation</td>
<td>84</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Experimental Setup</td>
<td>84</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Canonical Workloads</td>
<td>84</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Application Workloads</td>
<td>85</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Results</td>
<td>86</td>
</tr>
<tr>
<td>5.3</td>
<td>VMM Application Modeler Conclusions</td>
<td>96</td>
</tr>
<tr>
<td>6</td>
<td>Feature Selection for Outlier Detection</td>
<td>97</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction to Feature Reduction</td>
<td>98</td>
</tr>
<tr>
<td>6.2</td>
<td>Local Kernel Density Ratio Feature Selection</td>
<td>100</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Local Kernel Density Ratio Criterion</td>
<td>101</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Forward Search Strategy</td>
<td>106</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Analysis of Computational Complexity</td>
<td>108</td>
</tr>
<tr>
<td>6.3</td>
<td>GPU-Based CUDA Implementation</td>
<td>110</td>
</tr>
<tr>
<td>6.4</td>
<td>Experimental Evaluation</td>
<td>114</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Datasets</td>
<td>114</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Outlier Detection Algorithms</td>
<td>115</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Feature Selection Algorithms</td>
<td>116</td>
</tr>
<tr>
<td>6.4.4</td>
<td>Results</td>
<td>117</td>
</tr>
<tr>
<td>6.5</td>
<td>LoKDR Feature Selection Conclusions</td>
<td>132</td>
</tr>
</tbody>
</table>
7 Summary

7.1 Contributions of the Thesis ........................................ 134
7.2 Future Work ......................................................... 136

A Additional LoKDR Results ........................................ 138

A.1 Sensitivity Analysis .................................................. 138
    A.1.1 Sensitivity to $k$ ........................................... 138
    A.1.2 Sensitivity to $\sigma$ ....................................... 139
A.2 Remaining AUC Results ............................................ 140
A.3 A Note on Parameter Selection ................................. 140

Bibliography ............................................................. 145
# List of Figures

2.1 Choosing an optimal operating point on the ROC curve ............... 15  
2.2 Constructing rate features .............................................. 23  
2.3 Example of information captured by relationship features (event interleaving) .............................................................. 24  
2.4 Example of local and global outliers ................................... 27  
2.5 CUDA thread organization .................................................. 32  

4.1 High-level design of our VMM IDS ........................................ 51  
4.2 Weights of an exponential distribution in the EWMA model ........ 57  
4.3 Time series plot illustrating the alarm-raising mechanism .......... 59  
4.4 VirtualBox virtualization structure ...................................... 61  
4.5 Timeliness plot for the MySQL test traces ............................. 72  

5.1 High-level design of our VMM Application Modeler .................. 82  
5.2 Linear least-squares regression results for MPrime .................. 87  
5.3 Linear least-squares regression results for Apache ................... 88  
5.4 Linear least-squares regression results for MySQL ................... 88  
5.5 Linear least-squares regression results for mcf ...................... 89
A.6 Average AUC on ARRHY dataset with NN and OCSVM classifiers . 144
List of Tables

2.1 Summary of the virtual events, the additional information extracted for the events, and what they mean for the OS ........................................ 20
2.2 Summary of the VMM events, the additional information extracted for the events, and what they mean for the OS ........................................ 21
3.1 IDS abstraction level comparison ...................................................... 39
4.1 Normal workloads (virtual appliances) ........................................... 62
4.2 Features selected for Apache workload .......................................... 67
4.3 Features selected for MySQL workload .......................................... 67
4.4 Features selected for Exchange.Light workload ................................. 68
4.5 Features selected for Exchange.Heavy workload ................................ 69
4.6 Features selected for Cloud Storage workload .................................. 70
4.7 Accuracy and time-to-detection results on test dataset ......................... 73
5.1 Summary of the canonical workload set .......................................... 85
5.2 Linear regression coefficients for an average window of the target workloads ................................................................. 90
6.1 Overview of datasets ........................................................................ 115
6.2 Lowest average BER results on all datasets ..................................... 124
Chapter 1

Introduction

Virtualization can be used to refer to a number of different technologies, including application virtualization, operating system (OS) virtualization, storage virtualization, and network virtualization [104]. In this work, we will delve into the virtues of OS virtualization and explore the richness available at the virtualization layer. Operating system virtualization, also known as platform\(^1\) virtualization, allows the creation of one or more virtual machines (VMs) that run on top of a single physical machine. A virtual machine is a software implementation of a complete computer system that executes programs just as would a physical computer system. Each virtual machine has its own operating system and is given the illusion that it is the only machine running; it therefore believes that it has exclusive access to the physical hardware. The magician keeping up this illusion is the virtual machine monitor (VMM) or hypervisor, a software layer providing the virtualization. The VMM manages access to the hardware resources, isolating the virtual machines and leaving them unaware of each other’s existence.

\(^1\)It may also be referred to as hardware virtualization.
1.1 OS Virtualization Methods

There are three main types of OS virtualization methods [135]:

1. Full virtualization using binary translation

2. OS-assisted virtualization or paravirtualization

3. Hardware-assisted virtualization

In full virtualization, also known as native virtualization, the guest OS is not aware that it is being virtualized and requires no modification. Sensitive OS calls are trapped using binary translation. Kernel code is translated to replace nonvirtualizable instructions with new sequences of instructions that have the intended effect on the virtual hardware. In contrast, user level code is unmodified and can directly execute on the processor at native speed. As described in [135], full virtualization offers the best isolation and security for virtual machines. It also simplifies migration and portability as the same guest OS instance can run virtualized or on native hardware.

In paravirtualization, the guest OS kernel is modified to replace nonvirtualizable instructions with hypercalls that communicate directly with the virtualization layer VMM. This communication between the guest OS and the VMM improves performance and efficiency by incurring less virtualization overhead. As paravirtualization cannot support unmodified operating systems, its compatibility and portability is poor. Due to the OS kernel modifications, paravirtualization can also introduce significant support and maintainability issues in production environments. The open source Xen VMM is an example of paravirtualization. Xen virtualizes the processor and memory using a modified Linux kernel. It virtualizes the I/O using custom guest OS device drivers.
In hardware-assisted virtualization, privileged and sensitive calls are set to automatically trap to the VMM, removing the need for either binary translation or paravirtualization. Examples include Intel Virtualization Technology (VT-x) and Advanced Micro Devices AMD-V, which both target privileged instructions with a new CPU execution mode feature that allows the VMM to run in a new root mode below ring 0.

1.2 Advantages of Virtualization

Virtual execution environments provide a myriad of advantages over traditional computing environments, such as server consolidation, increased reliability and availability, and enhanced security through isolation of virtual machines (VMs) [98]. In server consolidation, virtual machines can be used to consolidate the workloads of several under-utilized servers to fewer machines, perhaps even a single machine. The consequent benefits of the consolidation include savings on hardware, environmental costs, management, and administration of the server infrastructure.

Through virtualization, multiple operating systems can run simultaneously on the same physical hardware. This includes different versions of the same OS or entirely different operating systems, some of which may be difficult or impossible to run on newer real hardware. The same is true for legacy applications; a legacy application might not run on newer hardware and/or operating systems. If it does, it may underutilize the server. Hence, it would be more efficient to consolidate several applications. Without virtualization, this may be difficult since these applications are usually not written to co-exist within a single execution environment.

Virtualization facilitates tasks such as system migration, backup, and recovery,
making them more manageable. Virtual machines make it easier to migrate software, thus providing application and system mobility. Application suites can be treated as appliances by “packaging” and running each in a virtual machine. These so-called virtual appliances will be the target of our VMM-based intrusion detection system described in Chapter 4.

Virtualization is an important concept in building secure computing platforms. Virtual machines can be used to provide secure, isolated sandboxes for running untrusted applications. As a consequence of isolation, VMs allow fault and error containment. Faults can be injected into software to study its resulting behavior. Another benefit of the isolation provided by virtualization is that virtual machines are safer to work with, making them ideal tools for research and academic experiments. A VM encapsulates the entire state of a running system. The state can be saved, examined, modified, and reloaded. It also provides an abstraction of the workload being run.

Regulation of the virtual machines falls on the shoulders of the virtual machine monitor. The VMM manages access to the physical hardware on behalf of the virtual machines. When an application running inside a virtual machine tries to execute a privileged instruction, the VMM must step in and handle the request. User-mode instructions, on the other hand, can execute unmodified on the virtual machine. The degree to which a VM interacts with the VMM can provide insight into the behavior of the applications running within the VM. Therefore, the data available at the VMM layer contains a rich set of behavioral characteristics of the workloads running inside the virtual machines. The only caveat is that the raw VMM-layer data is not in a form that can be effectively traced back to the actual program execution. In other words, there is a large semantic gap between the low-level data and the true application behavior. Thus arises the need to manipulate the raw data into a form which can
be used by machines to “learn” the higher-level program behavior and help bridge the semantic gap. To this end, we apply machine learning and data mining techniques to the data available at the VMM layer to better understand the behavior of a workload running inside a virtual machine. Only then can the richness hidden within the VMM layer be truly uncovered and put to good use. In this work, we explore two main applications of this idea. In the first, we enhance anomaly detection techniques to distinguish normal behavior from abnormal, potentially malicious, activity. In the second, we adapt regression algorithms to break down a normal workload’s behavior into a mix of simpler behaviors that can provide insight into the execution characteristics of the workload.

1.3 Challenges of Working Purely at the Virtual Machine Monitor Layer

To better understand the challenges of using only data at the VMM layer and the need for applying machine learning techniques, we present a simple example illustrating the type of information available in the progression from the application layer to the operating system layer, and down to the VMM layer. The following piece of code, written in the C programming language, opens a file named myfile.txt and writes: Hello world! (preceded by the line number) ten times and then closes the file.
int main()
{
    int i;
    FILE *fp;
    fp = fopen("myfile.txt", "w");
    for (i = 0; i < 10; i++)
        fprintf(fp, "%d: Hello world!\n", i);
    fclose(fp);
    return 0;
}

We execute the code on a VirtualBox virtual machine with a Windows XP operating system. Using the WinTrace [37] tool, which is the equivalent of the strace Linux utility for Windows-based operating systems, we collect a trace of system calls generated by the program. The system call trace contains information about the sequence of system calls generated during the program execution, including any input and output parameters. In what follows, we provide a snippet of the system call trace produced by the execution of the aforementioned C program.
The first value on each line represents the process ID of the running program. The second value is the name of the system call that is executed. The third value (in brackets) is the number of input arguments for the system call. The next set of values (in parentheses) are the inputs of the system call. The value that follows the equal sign is the output of the system call (if any). In the system call trace segment above, the first line checks to see if the file myfile.txt exists. Since it is the first

---

The arguments shown with question marks are values that were not able to be identified using the WinTrace tool.

---

2The arguments shown with question marks are values that were not able to be identified using the WinTrace tool.
time that the program is executed and no other file by that name exists in the current
directory, the file is not found and a status value of STATUS_OBJECT_NAME_NOT_FOUND
is returned. The next line executes a system call to create the file, as the name
NtCreateFile suggests. The NtWriteFile system call then writes the appropriate
values into the file and finally, NtClose closes the file.

From the system call trace, we see that there are several pieces of key information
available at the operating system level (which are also available at the application
level):

- The act of creating\(^3\) and opening a file
- The name of the file that is accessed (myfile.txt)
- The actual values written into the file
- The act of closing the file

The data available at the VMM level lacks the semantic information available at
higher levels of abstraction (i.e., the application level and OS level). At the virtu-
alization layer, there is no notion of a file system. All that the VMM sees are disk
accesses (in this example, disk writes). When a disk write event occurs, the VMM
intercepts the request and we record the disk sector accessed, the number of bytes
accessed, and the fact that a disk write occurred (versus a disk read).

To see the type of information available at the VMM layer, a small snippet of the
VMM-level event trace corresponding to the C program is shown next.

\(^3\)While we do not explicitly see a line corresponding to the file creation in the C code, it is implicit
during the open operation that if a file does not exist, one is created.
The word ioHD signifies a disk access event and the letter W represents the occurrence of a disk write. The last three values are, respectively, a pointer to a hard disk drive (HDD) container, the offset of the first byte from the start of the disk, and the number of bytes to write. While these low-level events available from the perspective of the VMM can provide valuable information regarding what takes place at a level close to the physical hardware, it still lacks much of the information available at the application and OS levels. Thus, on the path of utilizing raw VMM-level events to help reconstitute an accurate picture of what is really going on at the application layer, the fields of machine learning and data mining can provide powerful tools to extract meaningful information from the VMM-level data.

In this thesis, we contribute additional facets to the attractive qualities provided by virtualization, as explained at the beginning of this introduction. We will show that there is a richness hidden within the raw data available at the virtualization layer that lends itself well to providing intrusion detection and workload characterization.
To help uncover the efficacy of the low-level data, we take advantage of the power of machine learning and data mining techniques to model the normal behavior of a workload. Subsequently, we can detect deviations from that model as suspicious activity, potentially caused by the presence of malware. In addition to this process of detecting intrusions based on a normal model, another application is to decompose the normal activity into a mix of simpler components that can be used to characterize a workload’s behavior. In this thesis, we explore these applications with the unique perspective of working purely at the virtualization layer.

This thesis enhances the current state-of-the-art and makes key contributions to the following fields:

- **Cyber Security for Virtualization** – By adapting sophisticated anomaly detection techniques to build an intrusion detection system for virtual execution environments

- **Workload Characterization for Virtualization** – By extending powerful regression methods to decompose a virtual machine workload’s behavior into primitive computing components

- **Machine Learning** – By proposing and implementing a novel, effective, and efficient feature selection algorithm for outlier detection

Next, we discuss the major contributions of this work and describe the organization of the remainder of the thesis.
1.4 Contributions of the Work

As part of this thesis, we enhanced the functionality of popular outlier detection techniques and adapted them to perform intrusion detection in a virtualized execution setting. Rather than training on a dataset consisting of both normal points and outliers, we trained only on a set of normal points to build a model of normal behavior. During the testing phase, we compared the test set against this model to detect deviations as outliers. In this way, outlier detection is performed in a one-class learning manner. We evaluated our virtual machine monitor intrusion detection system (VMM IDS) on a set of approximately 300 real-world malware and will show that with an average of 3% false alarms, it is possible to achieve an average of 93% true detections. This work has been presented at Women in Machine Learning (WiML) [20] and featured in Secure and Resilient Architectures and Systems (SRAS) [18], the Operating Systems Review (OSR) journal [19], and the International Conference on Machine Learning and Applications (ICMLA) [10]. In addition, an invited talk was given on this work at the Northeastern University/MIT Lincoln Labs Cyber Security Meeting [14].

By continuing to learn at the virtualization layer, we developed the ability to perform workload characterization of virtual machine workloads using powerful regression techniques. We evaluated the resulting VMM Application Modeler on a set of real-world benchmarks that exhibit a range of behaviors typical of most workloads. We will show that the activity of a generic workload can be decomposed and modeled as a weighted mix of simpler behaviors consisting of CPU, memory, disk read, disk write, network receive, and network transmit activity. The relative weights of the different components may vary during the execution, thereby providing the ability to perform phase detection. This work has been presented at WiML [21] and featured
in the Institute of Electrical and Electronics Engineers (IEEE) International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS) [22].

We also proposed a novel feature selection algorithm designed specifically to cater to outlier detection problems. It is the first algorithm to tackle the difficult task of selecting features to facilitate the detection of outliers. The main idea is to choose features that maximize the density of normal points while simultaneously minimizing the density of outliers, and therefore increasing the likelihood that outliers stand out from the normal. With the opportunities for parallelism provided by the algorithm, we implemented it on a graphics processing unit (GPU) and achieved significant speedup. This work has been presented at the Society for Industrial and Applied Mathematics (SIAM) International Conference on Data Mining (SDM) [15] and New England Machine Learning (NEML) [17]. It has also been featured in the Asian Conference on Machine Learning (ACML) [16].

1.5 Organization of Thesis

The central focus of this work is to apply data mining and machine learning techniques to extract useful information from the VMM layer and perform intrusion detection and workload characterization. The remainder of the thesis is organized as follows: Chapter 2 presents background information on virtualization, intrusion detection, workload characterization, the process of VMM-level event extraction and feature construction, anomaly detection algorithms, and regression algorithms. In Chapter 3, we present related work in intrusion detection, including a revised IDS taxonomy which is used to classify the current state of the art in IDSs. We also survey the
literature of related work in the area of virtualization-based workload characterization and the application of regression techniques to the task of performance analysis. In addition, we present related work on feature selection and the application of density ratios for outlier detection.

Then, we present our work in applying machine learning and data mining to the task of learning from the VMM-level data for the purpose of intrusion detection and workload characterization. In Chapter 4, we discuss the motivation for the the VMM-based intrusion detection tool and provide details of our evaluation of the IDS. Chapter 5 describes how the VMM Application Modeler can be used to model the behavior of a set of target workloads in terms of a canonical workload set. In Chapter 6, we present our novel feature selection algorithm for outlier detection problems, as well as details of our GPU-based implementation which results in a significant speedup of the algorithm. Finally, we conclude the thesis in Chapter 7 and summarize our work.
Chapter 2

Background

In this chapter, we present background information on concepts used throughout the thesis, including intrusion detection, workload characterization, the extraction and processing of the raw VMM-level data, the anomaly detection algorithms, and the regression algorithms.

2.1 Intrusion Detection

An intrusion is defined as an attempt to compromise the confidentiality, integrity, or availability of a computer system or network, or an attempt to bypass its security mechanisms [23]. Intrusion detection is the process of monitoring the events occurring in a computer system or network and analyzing them for signs of intrusions. An intrusion detection system (IDS) is a software or hardware system that automates the process of monitoring the events occurring in a computer system or network and analyzing them for signs of security problems.

If an IDS correctly identifies an attack, it has made a true detection. The detection rate is the number of correct detections divided by the number of attacks. If the IDS
incorrectly classifies a normal execution as an attack, it raises a false alarm. The false alarm rate is the number of false alarms divided by the number of normal executions.

To evaluate our intrusion detection system, we use a receiver operating characteristic (ROC) curve. An ROC curve is a plot of the true detection rate versus the false alarm rate. It is a technique for visualizing, organizing, and selecting classifiers based on their performance [51]. The ideal operating point on an ROC curve is the point (0,1), where the false alarm rate is 0% and the detection rate is 100%.

![Diagram of ROC curve]

**Figure 2.1**: Choosing an optimal operating point on the ROC curve

Figure 2.1 shows a generic example of an ROC curve. The line $y = x$ represents the true detection versus the false alarm rate of a classifier which makes random guesses as to which class each data point belongs. A viable classifier is one whose
corresponding ROC curve is above this line and, preferably, comes close to the ideal operating point $P$. In this figure, point $A$ is an operating point on the ROC curve. The distance from point $A$ to point $P$ (the length of line segment $AP$) provides an indicator of the optimality of operating point $A$. The smaller the distance, the better the operating point $A$ on the ROC curve. We select the optimal operating point on the curve as the point with the smallest distance to the ideal operating point $P$. For symmetric or near-symmetric ROC curves, such a point will roughly strike a balance between the false positive rate (i.e., false alarm rate) and the false negative rate (i.e., $1 -$ true detection rate) [70]. If two points on the ROC curve have the same distance to point $P$, we choose the operating point with the lower false alarm rate.

2.2 Workload Characterization

Workload characterization consists of a description of the workload by means of quantitative parameters and functions. The objective is to derive a model able to show, capture, and reproduce the behavior of the workload and its most important features [31]. In our work, we apply regression algorithms to data extracted from the VMM layer to characterize a generic workload in terms of a mix of simpler workloads.

Application execution is composed of a number of different behaviors [71]: CPU-activity, memory-activity, and I/O-activity. As such, we begin by capturing the behavior of a set of workloads geared toward each of these components, which we refer to as our canonical workloads. For example, a CPU-intensive workload that focuses on pushing the limits of CPU activity comprises the CPU workload of our canonical set. We further distinguish I/O activity in terms of whether the activity is disk I/O-intensive or network I/O-intensive. Since I/O-intensive workloads are
bidirectional (i.e., they perform both reads and writes), we further decompose disk
I/O-intensive workloads into disk read-intensive and disk write-intensive workloads,
and the network I/O-intensive workloads into network receive-intensive and network
transmit-intensive workloads. Thus, our canonical workload set consists of the fol-
lowing workloads:

1. CPU
2. Memory
3. Disk Read
4. Disk Write
5. Network Receive
6. Network Transmit

2.3 VMM-Level Event Extraction

We use the term events to describe the raw data and information extracted from the
VMM during execution. The information we can extract from different VMMs differs
depending on the specific VMM implementation; this can affect the effectiveness and
accuracy of the IDS. The success of a VMM-based IDS hinges on its ability to take
the extracted events and piece together an accurate picture of the system behavior at
the application level. In particular, it must be able to detect the change in behavior
cau sed by the execution of the malware.

In our work, we target similar VMMs (in terms of performance, target architec-
ture, etc.) such as VMWare Workstation [6], VirtualBox [39], ESX Server [6], and
Xen [26]. A subset of events can be found in all of them. These events are architectural events that the VMM must intercept to guarantee correct execution. They include execution of privileged instructions, access to shared resources (memory), and I/O (disk, network, devices). We rely on this common set of events to create a robust VMM-based IDS.

The VMM lies below the guest OS layer and provides the illusion that the guest OS is running on a real machine. Hence, every time the OS needs to interact with the hardware, the VMM must intervene. It is through this intervention that we are able to collect events. The VMM is instrumented to log the occurrence of an event, as well as any available information relevant to the event. For example, when a disk I/O event occurs, the VMM intercepts the request and we record the disk sector accessed, the number of bytes accessed, and read/write status.

The data available to the VMM dictate the types of events that we are able to monitor. The richness of the events, in terms of the wealth of information they provide to the VMM, is critical to the success of a VMM-based IDS. Consequently, a vast array of events must be collected in order to reconstruct a more accurate picture of what occurs at the application level. Events can be categorized into two main types:

1. Virtual or VM events – architectural-level and system events related to the virtualized guest OS executing inside the VM.
   For example, a guest modifying control registers, flushing the TLB, or writing to the disk are instances of VM events.

2. VMM events – events extracted from the VMM and which relate to the state of the VMM itself.\(^1\)

\(^1\)Naturally, the VMM state is influenced by the state of the guest OS and the interaction between the guest OS and VMM.
For example, VirtualBox has two internal modes: one to execute the guest OS directly on the CPU without intervention (user mode instructions) and another to intercept, instrument, or emulate supervisor mode instructions.

We are also able to extract real events from within the VMM or from the host OS. The semantics of these events are related to the host OS. The real time clock is an example of such an event. In our analysis of the events, we determined that real events are not as helpful in characterizing system-level behavior due to the fact that they are too far removed from the application layer to provide any useful information. More precisely, they do not contain sufficient semantics to infer the true context of the application behavior.

For some events, in addition to determining when they occur, we can also extract useful information about the events. For example, during a disk I/O event, we can determine information such as the disk sector, number of bytes accessed, and whether the disk access was a read or write event. A summary of the Virtual and VMM events that we extract is provided in Table 2.1 and Table 2.2, respectively. In the tables, we also list additional useful information that can be extracted for the events. The events we are interested in gathering should be able to hint at the underlying behavior of the system to allow us to distinguish changes in the system behavior. These high-level semantics of the events are provided in the last column of the tables.

\footnote{It is possible to extract even more information, such as the disk ID (pdisk) and whether the access was synchronous or asynchronous. We have not found them to be particularly useful, but they can be utilized in future work.}
<table>
<thead>
<tr>
<th>Virtual Event</th>
<th>Additional Information</th>
<th>High-Level Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk I/O</td>
<td>Disk sector, Number of bytes,</td>
<td>Disk read/write event</td>
</tr>
<tr>
<td></td>
<td>Whether data was read or written</td>
<td></td>
</tr>
<tr>
<td>Network I/O</td>
<td>Number of bytes</td>
<td>Network read/write event</td>
</tr>
<tr>
<td></td>
<td>Whether data was sent or received</td>
<td></td>
</tr>
<tr>
<td>Programmable Interrupt Timer (PIT)</td>
<td>–</td>
<td>Means by which the OS can track time</td>
</tr>
<tr>
<td>or Real Time Clock (RTC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page mode change</td>
<td>–</td>
<td>Hints at the OS state</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read/write events to control registers</td>
<td>Register number, Value,</td>
<td>Hints at the OS state</td>
</tr>
<tr>
<td></td>
<td>Whether the value was read from or written into the control register</td>
<td></td>
</tr>
<tr>
<td>Page faults</td>
<td>Error code, Virtual EIP register</td>
<td>Hints at memory usage and application behavior</td>
</tr>
<tr>
<td>TLB flush</td>
<td>Global flag, New CR3 register value</td>
<td>Occurs during a context switch</td>
</tr>
<tr>
<td>Invalidate page</td>
<td>Page to invalidate (linear address)</td>
<td>Hints at memory and application behavior</td>
</tr>
<tr>
<td>Local descriptor table (LDT),</td>
<td>Fault address (CR2 register)</td>
<td>Hints at application startup and context switches</td>
</tr>
<tr>
<td>global descriptor table (GDT), and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>task state segment (TSS) access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPUID</td>
<td>Leaf (from ring 0)</td>
<td>Request for the ID by the application</td>
</tr>
<tr>
<td>Current Privilege level (CPL)</td>
<td>Privilege level</td>
<td>Hints at the type of code running ³</td>
</tr>
<tr>
<td>Load segment (descriptor)</td>
<td>Segment register, base, limit,</td>
<td>Whether in user mode or supervisor mode ⁸</td>
</tr>
<tr>
<td></td>
<td>selector, flags</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Summary of the virtual events, the additional information extracted for the events, and what they mean for the OS

### 2.4 Feature Construction

We use the term *features* to describe the information and input format provided to the back-end of our intrusion detection and workload characterization tools. The back-end is sent processed information that can be the result of filtering, aggregating, or correlating events. Features do not contain all the information extracted from the events but incorporate patterns that are effective in identifying normal and abnormal

³The term “hint” is used to imply the possible occurrence of a specific type of event.
⁴For example, when in the real mode, there is no paging and the system is still booting up.
⁵For example, a write event to control register 3 (CR3) denotes a context switch.
⁶The Exception Instruction Pointer (EIP) register specifies the address that caused the page fault.
⁷A page fault can hint at the start of a new application.
⁸The type of code may be privileged OS code or the application code.
Table 2.2: Summary of the VMM events, the additional information extracted for the events, and what they mean for the OS

<table>
<thead>
<tr>
<th>VMM Event</th>
<th>Additional Information</th>
<th>High-Level Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set guest trap handler</td>
<td>Trap number</td>
<td>Useful to identify the type of guest OS running, if needed</td>
</tr>
<tr>
<td>Reschedule execution mode</td>
<td>Values of registers EIP, ESP, CS, SS, EFLAGS, CR0, and CR4</td>
<td>Hints at the type of code running inside the OS</td>
</tr>
<tr>
<td>(RAW, HWACC or REM)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enter execution mode</td>
<td>Depending on the execution mode, the information collected includes CS, EIP, ESP, V86/CPL, EFLAGS.IF, CR0, vmFlags, and the new mode (RAW, HWACC or REM)</td>
<td>Similar to above, hints at the guest OS behavior</td>
</tr>
<tr>
<td>(RAW, HWACC or REM)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

execution. Using features rather than raw events can help identify relevant execution patterns and behaviors.

The space of possible features is very large; there are many ways to process raw events to create features. All of the features we use in this work are constructed directly from the event stream in the following way: first, we divide the event stream into consecutive segments of equal (virtual) time. Then, we apply statistical methods (described next) on the events in the segment to produce a vector of feature-values. Finally, we send a window, containing a vector of all feature-values for a time segment, to our back-end.

The advantages of using time-based windows are two-fold. First, all windows represent approximately the same amount of execution time and are therefore comparable. Second, splitting the event stream into windows provides us with the ability to classify each window on its own, enabling on-line classification. The length of the window (the virtual time captured in the window) introduces a trade-off: longer windows capture more behavior while shorter ones reduce the time-to-detection. We have found that windows of approximately two seconds long provide a good balance.

The type of code could be application code, previously unseen OS code, or previously seen OS code.
between these trade-offs. This time quantum provides the back-end with sufficient information to make accurate classifications on a per-window basis, while also allowing it to identify any malicious activity within seconds of its execution.

In the next sections, we describe how we process events to create the features used in this work.

2.4.1 Rate Features

The first category of features we generate from events are created by storing a running sum of an event. These rate features are constructed by simply counting the occurrences of events in each segment. In Figure 2.2, we show how a stream of events are processed to construct windows containing feature-values for rate features. First, on the left we show a list of events as they occur and how they are divided into segments (in this example, a segment is constructed on each timer event). Next, on the right we show (for each segment) rate feature-values for network I/O and disk I/O, contained in a window.

We build multiple rate features such as page-fault count, control register modifications, disk and network I/O accesses, etc. These features can be constructed efficiently and provide rich information about the execution behavior of the system.

2.4.2 Relationship Features

The next category of features is what we refer to as relationship features, as they capture the relationship between pairs of events. Relationship features are built with the intention of capturing information not covered by rate features. This may happen for example, when different events have the same rate across vastly different windows. We are able to differentiate between these windows by accounting for the order that
Figure 2.2: Constructing rate features

particular events took place. For example, we are able to detect that a sequence of disk I/O read events which is followed by a sequence of disk I/O write events is different than a sequence of interleaving read and write disk I/O events.

In Figure 2.3, we present an example. We show two pairs of windows, each pair containing an equal number of events $A$ and $B$, but different orders in which they occur. We generate features that are able to detect the difference between the top and bottom window in each pair, using statistical methods such as (1) maximum discrepancy, the largest gap between successive occurrences of events $A$ and $B$ within any sub-window and (2) the Mann-Whitney test, a measure of how randomly interleaved events $A$ and $B$ are within the window. Another relationship feature counts
the number of runs, i.e., the number of sub-windows containing a sequence of the same event. For the first pair of windows in Figure 2.3, this relationship feature produces a feature-value of 2 for Window 1.a and 16 for Window 1.b. For the second pair, the value of this relationship feature is 4 for Window 2.a and 7 for Window 2.b. Using relationship features, we can distinguish between the top and bottom windows in each pair.

The significance of relationship features is reflected in the fact that they account for more than half of the features chosen by the feature selection algorithm. As we shall show in Chapter 4, the information conveyed by relationship features allows the machine learning algorithms to distinguish normal behavior from malicious activity.

An example relationship feature chosen for one of our workloads (Exchange.Light) is a feature that measures the random interleaving between writes to the task state segment (TSS) and TLB flushes, both of which can be triggered by a task switch.

In order to fairly compare the feature values, we normalize them to bring them onto the same scale. In our intrusion detection work, we use standardization where the features are normalized to have a mean of zero and standard deviation of one. This is accomplished by subtracting the mean from each feature and dividing the
result by the standard deviation, as shown below:

\[ f'_{ij} = \frac{f_{ij} - \text{mean}_j}{\text{std}_{dev}^j} \quad \text{for} \quad i = 1, \ldots, n \quad \text{and} \quad j = 1, \ldots, d \quad (2.1) \]

where \( n \) is the number of windows, \( d \) is the number of features, \( f_{ij} \) is the \( j^{th} \) feature of the \( i^{th} \) window, \( f'_{ij} \) is its normalized value, \( \text{mean}_j \) is the mean value of the \( j^{th} \) feature, and \( \text{std}_{dev}^j \) is the standard deviation of the \( j^{th} \) feature.

For the workload characterization, we normalize the features to be in the range [0, 1]. This normalization is achieved by subtracting the minimum value from each feature and dividing the result by the original range of the feature, as shown below:

\[ f''_{ij} = \frac{f_{ij} - \text{min}_j}{\text{max}_j - \text{min}_j} \quad \text{for} \quad i = 1, \ldots, n \quad \text{and} \quad j = 1, \ldots, d \quad (2.2) \]

where \( \text{min}_j \) is the minimum value of the \( j^{th} \) feature across all windows and \( \text{max}_j \) is the maximum value of the \( j^{th} \) feature.

### 2.5 Anomaly Detection Algorithms

Various data mining and machine learning techniques have been applied to the design of IDSs [114]. In this work, we employ two well known techniques to build our VMM-based IDS, each taking a different approach to creating a model of normal behavior and performing anomaly detection. They are the distance-based \( K \)-Nearest Neighbors (KNN) and the density-based Local Outlier Factor (LOF) algorithms.

Several studies have applied KNN and LOF classifiers to the area of intrusion detection. Liao and Vemuri [90] employed KNN to categorize program behavior as being either normal or intrusive, using the frequency of system calls. Adebayo et al. [7] compared the performance of the Rough Set (LEM2) algorithm and KNN on the Knowledge Discovery and Data Mining (KDD) dataset for benchmarking intrusion
detection systems. Their results show that KNN outperforms Rough Set in terms of accuracy. Lazarevic et al. [87] performed a study of several anomaly detection schemes, including KNN and LOF, in network intrusion detection using the DARPA 1998 dataset and real network data. They found that the most promising technique for detecting intrusions in the DARPA dataset was the LOF algorithm. In addition, when performing experiments on real network data, the LOF approach was very successful in picking several interesting novel attacks that could not be detected using other state-of-the-art intrusion detection systems such as SNORT [127]. For our VMM IDS, we also find that LOF can be valuable when combined with KNN to detect novel attacks. Next, we describe these algorithms in more detail.

2.5.1 K-Nearest Neighbors

The typical method of classifying a test data point with the K-Nearest Neighbors (KNN) algorithm is to find the point’s $k$-nearest neighbors and use the labels of those points to assign a label to the test point. In this manner, the learning is performed in a supervised fashion (due to the use of labeled data). We modify the algorithm to provide semi-supervised (one-class) learning by performing two main phases: (1) a model-creation or profiling phase, and (2) an anomaly detection phase. The profiling phase of the algorithm consists of simply storing the vectors of training data points (in our case, windows of normal activity). In the anomaly detection phase, each (validation or test) data point is assigned a score, or decision value, indicating how “abnormal” it is by calculating the sum of the distances to its $k$-nearest neighbors. The farther a data point is with respect to its $k$-nearest neighbors, the more abnormal it is and the larger the decision value assigned to it. The distance between pairs of data points can be measured using different metrics, such as the Euclidean distance.
2.5.2 Local Outlier Factor

The Local Outlier Factor (LOF) algorithm [28] takes a density-based approach to anomaly detection. Anomalous data points are also referred to as outliers. The strength of this algorithm lies in its ability to identify local, as well as global, outliers. This is illustrated in Figure 2.4. Point $P_1$ is a global outlier. A local outlier is a data point that is outlying when compared to its surrounding local neighborhood. Point $P_2$ is an outlier when compared to cluster $C_1$ and therefore it is a local outlier. Distance-based outlier detection algorithms would only be able to recognize point $P_1$ as an outlier and not point $P_2$. This is because the distance between point $P_2$ and its nearest neighbors is less than the distance between many points in cluster $C_2$ and their nearest neighbors, so a method such as KNN would not be able to distinguish $P_2$ as an outlier without incorrectly classifying many points in $C_2$ as outliers. LOF, on the other hand, is able to correctly identify both the global and local outliers without misclassifying any points in $C_2$.

Figure 2.4: Example of local and global outliers
In the LOF algorithm, the density of a data point is compared to that of its local neighborhood and based on this, the point is assigned a *degree* of being an outlier, known as its local outlier factor. The size of the neighborhood under consideration is determined by the parameter $k$. In the profiling phase, the density of all the training data points is calculated. In the anomaly detection phase, the LOF score of each (validation or test) data point is found by dividing the average density of its $k$-nearest neighbors by its own density. Intuitively, the greater the density of a data point’s neighbors relative to its own density, the more outlying the data point is and hence, the higher the LOF score (decision value) assigned to it. In the case of intrusion detection, if a malware tries to imitate normal behavior and only differs from it slightly, due to its malicious nature, the LOF algorithm should be able to detect this *relative* deviation.

While the LOF algorithm is typically trained on a set of (unlabeled) normal and outlier points, we adapt the algorithm to perform anomaly detection. An advantage of using anomaly detection is its ability to identify novel attacks, which stems from the method by which the classifier is trained. Rather than profiling using a set of normal and outlier data points, the algorithm performs profiling using a set of only normal data points. In this way, it is used to build a model of typical, normal behavior. Subsequently, any deviations from the normal model are flagged as anomalous. In an environment where normal execution tends to change and evolve, it is important to use an algorithm that can dynamically update the model of normal behavior so as to reduce the false alarm rate. A nice feature of the LOF algorithm is that it lends itself to an incremental version [115] that does just that.\textsuperscript{10}

The main steps in the LOF algorithm can be summarized as follows:

\textsuperscript{10}In this work, we used the original LOF algorithm.
1. For each data point \( p \), find \( k\)-distance\((p)\), the distance to its \( k\)-th nearest neighbor. Data points that lie within \( k\)-distance\((p)\) to data point \( p \) are in its \( k\)-distance neighborhood.

2. For each data point \( q \) in the \( k\)-distance neighborhood of \( p \), find the reachability distance of \( p \) with respect to \( q \) as \( \max\{k\text{-distance}(q), d(p, q)\} \), where \( d(p, q) \) is the distance between \( p \) and \( q \).

3. Calculate the average reachability distance of \( p \) with respect to data points in its \( k\)-distance neighborhood. Take the inverse of that average as the local reachability density of \( p \).

4. Determine the local outlier factor (LOF) of \( p \) as the average local reachability density of points in its \( k\)-distance neighborhood divided by the local reachability density of \( p \).

### 2.6 Regression Algorithms

The regression algorithms applied in this work are multiple linear least-squares regression and the LASSO algorithm. The details of these algorithms are provided next.

#### 2.6.1 Linear Least-Square Regression

Multiple linear least-squares regression [50] can be used to describe a variable in terms of a linear combination of a set of independent variables. We use this notion to describe a target workload as a linear combination of a set of canonical workloads. We can represent the target workload and each canonical workload as a vector of its
different feature-values. In the following regression formula, we use \( y_i \) to denote the \( i^{th} \) feature of a target workload, \( x_{ij} \) for the \( i^{th} \) feature of the \( j^{th} \) canonical workload, \( b_j \) for the coefficient of the \( j^{th} \) canonical workload, and \( e_i \) as the \( i^{th} \) error term.

\[
y_i = b_0 + \sum_{j=1}^{c} b_j x_{ij} + e_i \quad \text{for } i = 1, \ldots, d \tag{2.3}
\]

where \( c \) is the number of canonical workloads, \( b_0 \) is an optional constant parameter, and \( d \) is the number of features. The solution to the linear regression is the set of coefficients \( b = (b_0, b_1, \ldots b_c) \) that minimize the sum of the squared errors. These coefficients comprise the workload characterization model for the target workload.

### 2.6.2 Least Absolute Shrinkage and Selection Operator

We also run experiments using a variant of linear regression that can automatically select the important input variables as well. It performs variable selection, or in our case, canonical workload selection by pushing the coefficients of irrelevant workloads toward zero, giving us a sparse solution (i.e., only a few of the canonical workloads have nonzero coefficients). In particular, it minimizes the least squares error criterion subject to the constraint that the sum of the absolute value of the coefficients (\( L_1 \) norm) is as small as possible. This algorithm is named LASSO [132] for *Least Absolute Shrinkage and Selection Operator*. The LASSO solution expects many coefficients to be close to zero, and a small subset to be nonzero [56]. LASSO solves the following optimization problem [55]:

\[
\min_b \frac{1}{2} \sum_{i=1}^{d} \left( y_i - \sum_{j=1}^{c} b_j x_{ij} \right)^2 \tag{2.4}
\]

subject to:

\[
\sum_{j=1}^{c} |b_j| \leq t \tag{2.5}
\]
where \( t \) is a pre-defined threshold. Equivalently, the solution to (2.4) also minimizes the Lagrange version of the problem:

\[
f(b) = \frac{1}{2} \sum_{i=1}^{d} \left( y_i - \sum_{j=1}^{c} b_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{c} |b_j|
\]

(2.6)

where \( \lambda \geq 0 \).

There is a one-to-one correspondence between \( \lambda \) and the threshold \( t \). If \( b(\lambda) \) minimizes (2.6), then it also solves (2.4) with \( t = \sum_{j=1}^{c} |b_j(\lambda)| \).

2.7 GPU and CUDA Background

A graphics processing unit (GPU), with its origins in graphics applications, is designed for compute-intensive, highly data-parallel computations. In a GPU, many transistors are dedicated for data processing, which give a GPU its computational power. In contrast to CPUs that rely on caches and predictors for hiding memory latency and provide single-threaded performance, GPUs rely on multithreading and provide throughput-oriented performance. Therefore, GPUs are well suited for data-parallel and task-parallel applications, resulting in large performance improvements.

Graphics processing units, as their name suggests, were originally designed for graphics acceleration with their fixed-function pipelines. They became more programmable as they evolved and Brook [30] was the first research compiler and runtime system demonstrating the use of GPUs for general purpose computing. After Brook, NVIDIA’s effort in this direction led to the introduction of Compute Unified Device Architecture (CUDA) in 2006 as the solution for general purpose computing on GPUs.

CUDA extends the C language with new programming constructs and provides libraries and a platform for the efficient execution of general-purpose applications.
on GPUs. CUDA provides a parallel computing and programming model. It has abstractions to effectively express data-level parallelism. Each data element maps to a thread and the threads are organized in a hierarchical manner for scalability. Programmers are expected to decompose the problem into coarse-grain subproblems that can be solved independently, and then each of these subproblems can be cooperatively solved by a group of threads at finer granularity. In CUDA, this group of threads is called a thread block. Threads within the same block can communicate cheaply using shared memories and efficient synchronization primitives. The thread blocks form grids at the higher level.

![CUDA thread organization](image)

Figure 2.5: CUDA thread organization

All threads in a grid execute the same kernel function, so they rely on unique coordinates to distinguish themselves from each other and to identify the appropriate
portion of the dataset to process [81]. The threads are organized into a two-level hierarchy. At the top level of the hierarchy, a grid is organized as a two-dimensional array of blocks. Each block is identified by its block index which consists of two components: the $x$ coordinate ($\text{blockIdx.x}$) and the $y$ coordinate ($\text{blockIdx.y}$). The size of a grid in terms of the number of blocks is determined by the grid dimension in the $x$ and $y$ coordinates ($\text{gridDim.x}$ and $\text{gridDim.y}$). At the bottom level of the hierarchy, the blocks of a grid are organized into a three-dimensional array of threads. All blocks in a grid have the same dimensions. Each thread within a block is identified by its thread index which consists of three components: the $x$ coordinate ($\text{threadIdx.x}$), the $y$ coordinate ($\text{threadIdx.y}$), and the $z$ coordinate ($\text{threadIdx.z}$). The size of a block in terms of the number of threads is determined by the block dimensions in the $x$, $y$, and $z$ coordinates ($\text{blockDim.x}$, $\text{blockDim.y}$, and $\text{blockDim.z}$). Figure 2.5 illustrates the CUDA thread hierarchy.

With the introduction of CUDA by NVIDIA, general purpose computing on GPUs very quickly became popular, as convenient programming models and platforms provided the opportunity to harness the computational power of GPUs and run many real-world applications with great performance gains. Following CUDA, other programming models and platforms have also been introduced, including OpenCL [78]. Using these programming platforms, many scientific, medical, and engineering applications in the fields of astronomy, biology, chemistry, physics, data mining, manufacturing, finance, and more have successfully been ported to GPUs.
Chapter 3

Related Work

In this chapter, we provide related work in the areas of intrusion detection, workload characterization, and feature selection.

3.1 Related Work in Intrusion Detection

A great amount work has been done in the area of host-based IDSs. As described in [100], they can be categorized into three main groups according to the information, or semantics, utilized by the IDS:

1. Program-level IDS – An IDS that uses information available at the program/application abstraction layer. This includes source code, static or dynamic information flow, and application execution state.

2. OS-level IDS – An IDS that utilizes information available at the OS layer such as system calls and system state. These IDS are executed ‘below’ the application.

3. VMM-level IDS – An IDS that uses semantics and information available at the virtual machine monitor layer. This includes architectural information available
to it (i.e., low level semantics). These are usually executed below the OS.

This classification is used to compare and contrast current IDSs in the next sections, and to highlight the novelty of our own work.

3.1.1 Program-Level IDS

A program-level IDS has all high level information available to it including source code, application data, and application control flow. There have been a number of information flow tracking systems that fall into this category. These systems include static [42, 106] and dynamic [134, 108, 124] data flow analysis to extract program-level information available to the application only. Monitoring control flow, a subset of information flow, can be used to thwart attacks that change the control flow dynamically. Such a control flow monitoring system, called program shepherding, was introduced by Kiriansky et al. [80].

3.1.2 Operating System-Level IDS

Much work has been done using system calls, available at the OS-level, to detect intrusions [32, 137]. Some focus on using just the sequence of system calls [52, 83, 138, 139, 72, 48]. In [123], a finite-state automaton (FSA) is constructed using a combination of the sequence of system calls and program state information and then used to detect anomalous sequences of system calls. The sequence time-delay embedding (STIDE) algorithm [68] uses a sliding window of length k over a system call trace to construct a database of fixed-length substrings. It then marks a test sequence as anomalous if the number of mismatches in the user-specified locality frame, which is composed of adjacent sequences in the frame, is more than the user-specified threshold. Entropy modeling is used to select an optimal window size in the work of Eskin.
et al. [49]. They also take advantage of the context dependency of the optimal window size and employ dynamic window sizes (rather than the fixed-length window size used in STIDE) by modeling system calls using Sparse Markov Transducers.

Instead of the sequence of system calls, system call arguments are used to build an IDS in [105] and [84]. To characterize system call arguments and identify anomalous occurrences, they build models based on string length, string character distribution, structural inference using Markov models and Bayesian probability, and a token finder to determine whether the values of a certain system call argument are drawn from a limited set of possible alternatives.

Other works combine both the sequence of system calls and their arguments for intrusion detection [93, 130]. In [38], Christodorescu et al. mine the specifications of malicious behavior by building system call dependence graphs from benign and malicious system call traces. They then compute a malicious specification as the difference between a malware dependence graph and a benign dependence graph using a minimal contrast subgraph.

Another approach is to use the occurrence of system calls to create a feature set and apply statistical, machine learning, or data mining techniques for anomaly detection [47]. Kang et al. [77] treat systems calls as a “bag of words” by only considering their frequency. In other words, their feature set consists of the raw count of system calls that are sequentially observed from the program. For anomaly detection, they apply the one class Naive Bayes algorithm and K-Means clustering. In the work of Ye and Chen [141], the feature vector contains an element for every possible system call. The feature values are generated by applying an exponentially weighted moving average (EWMA) [101, 25] technique. Their anomaly detection method is based on a chi-square statistic which signals anomalies at a threshold of 3.
sigma, i.e., when the value of the test statistic is greater than 3 standard deviations from its mean.

In the work by Oliveira et al. [41], a virtual machine is used to provide recovery from zero-day control-flow hijacking attacks. The attack detection mechanism involves augmenting every 32-bit word of memory and general purpose register with an integrity bit, used to determine when a vulnerability is being exploited.

### 3.1.3 Virtual Machine Monitor-Level IDS

To clarify the novelty of our work and distinguish it from previous approaches that incorporate the VMM for intrusion detection, we define two classes of VMM-level intrusion detection systems:

- Hybrid VMM/OS IDS
- Pure VMM IDS

Hybrid VMM/OS intrusion detection systems utilize the VMM as a means to isolate and secure the IDS. However, they rely on OS-level information and therefore are not pure VMM IDSs. Pure VMM intrusion detection systems, on the other hand, only use semantics visible to the VMM to perform intrusion detection. This limits the amount of information available to the IDS and poses a greater challenge. Chen et al. [34] allude to this difficult task as they acknowledge the importance of bridging the semantic gap between virtual machine events and operating system events. The work presented in this thesis is an example of a pure VMM-level IDS that uses machine learning techniques as a powerful tool to help bridge this semantic gap and make the most of the limited information available to the VMM.
Work on hybrid VMM/OS intrusion detection systems include the efforts of Lau-
reano et al. [85, 86]. In their work, a VM is used to encapsulate the system being
monitored. The IDS is implemented as a normal process on the host system, thereby
isolating and securing the IDS outside the guest OS. The VMM is a User Mode
Linux [133] that is modified to extract system calls. Then, STIDE system call se-
quence analysis [68] is used to perform anomaly detection. Zhang et al. [143] use a
Xen [26] VMM to intercept sequences of system calls that are analyzed to detect in-
trusions. In the work of Jin et al. [74], a privileged VM is set up to perform intrusion
detection in a centralized manner for a distributed virtual computing environment.
They use iptables [118] for the firewall and SNORT [127], a network-based IDS, for
the intrusion detection.

Garfinkel et al. present a Virtual Machine Introspection architecture [59] which is
used to create a set of policies for intrusion detection. They use a special OS interface
library to access information available at the OS level. Similarly, the VMwatcher
system uses a VMM to export the guest OS raw disk and memory to a separate VM.
A driver and symbols are used to compare memory views to detect rootkits and run
an anti-virus software on the (exported) disk [73].

The work done by Jones et al. [75, 76] takes an approach similar to ours. They
use only VMM-level semantics and information to detect hidden processes in virtual
machines. While they utilize information available in the VMM, the scope of their
IDS is focused on hidden processes. Malware may create new process (not hidden)
or attach itself to existing processes, thus eluding detection. In our work we develop
a generic IDS able to detect a broader class of intrusions.

Ether, a project developed by Dinaburg et al. is a transparent malware ana-
lyzer [44]. Ether uses hardware virtualization extensions to extract information from
the guest to analyze malware behavior. In our work we are able to extract similar information and use it to identify malicious activity.

3.1.4 IDS Comparison

Table 3.1 shows the trade-offs associated with the different IDS types according to the semantics available to them [100].

<table>
<thead>
<tr>
<th></th>
<th>Program IDS</th>
<th>OS and Hybrid IDS</th>
<th>Pure VMM IDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Applicability</td>
<td>High</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Performance</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Ease of deployment</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Attack Resistance</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 3.1: IDS abstraction level comparison

We identify the following trends in IDS design. First, the more program-level semantics there are available to the IDS, the more accurately it is able to identify and classify the malware. Using fewer semantics (as in the case of a VMM-level IDS) can limit the effectiveness of an IDS and may restrict the guest environments that can be effectively secured. In this sense, the applicability of a VMM IDS to more general computing environments is lower than that of IDSs which have more information available to them.

Second, more semantics may also impact performance, as extracting this information (if available) can introduce significant overhead (e.g., information flow tracking systems). A VMM IDS only requires that information be extracted from the VMM layer and therefore results in significantly less overhead. Since the VMM is the only
element that needs to be modified to extract this information, it also has the advantage of greater ease of deployment. In most VMMs, a majority of the information is already available through standard profiling interfaces (e.g., Vcenter Server in VMware’s ESX).

Third, extracting program-level semantics may be done by modifying or monitoring the execution of the application. Frequently, this is done at runtime in the application address space (e.g., using binary instrumentation). While these techniques are able to protect the IDS from the application they monitor, they are still vulnerable to malware running at a higher privilege level. For example, a root-kit may thwart both application-level and OS-level IDSs. In contrast, malware would first have to detect that an application or operating system is virtualized before a corruption can be launched against a VMM-level IDS. In addition, the VMM provides intrusion detection from outside of the compromised VM. Hence, a VMM-level IDS has higher resistance to a malicious attack.

### 3.1.5 Data Mining for Security

Related work has also been done in the area of applying data mining and machine learning techniques to ensure the security of computing resources. In [122], Schultz et al. use data mining techniques to detect the existence of new malicious executables. They use the static properties of an executable to generate features and apply an inductive rule learner, Naive Bayes classifier, and an ensemble classifier (constructed from several Naive Bayes classifiers) to distinguish malicious executables. Their results show that these techniques outperform a signature-based scanner. In order to discriminate between benign executables and viruses, Wang et al. [136] statically extract dynamically linked libraries and application programming interfaces, utilizing
Support Vector Machines for feature extraction, training, and classification.

In the work of Lee et al. [88, 89] data mining techniques have been applied to system calls and network data to develop intrusion detection models. Their methods include classification, meta-learning, association rules, and frequent episodes. In [60], clustering is applied to network traffic with the goal of detecting botnets. Their detection framework clusters together similar communication traffic and similar malicious traffic, and performs cross-cluster correlation to identify hosts that share both similar communication patterns and similar malicious activity patterns.

The Local Outlier Factor (LOF) and the $K$-Nearest Neighbor (KNN) algorithms have also been applied in host-based and network intrusion detection systems, as we described in Chapter 2. In our work, we apply these machine learning and data mining techniques to low-level architectural data extracted from the VMM-layer in order to protect software appliances and servers in a virtualization setting.

### 3.2 Related Work in Workload Characterization

Prior work on virtualization-based workload characterization has focused on disk I/O workload characterization. Ahmad [9] uses online histograms to analyze the performance of disk workloads running in virtual machines on VMware’s ESX Server. His method is used to characterize several benchmarks, including (1) Filebench – a model-based workload generator for file systems, (2) Open Source Development Labs (OSDL) Database Test 2 – an online transaction processing (OLTP) transactional performance test based on the Transaction Processing Performance Council Benchmark C (TPC-C) specification, and (3) a workload that performs large file copy in virtual machines.
Along the same lines, Gulati et al. [62] perform storage workload characterization and consolidation for enterprise applications using VMware's ESX Server. Their benchmark workloads include Swingbench, DVDStore, Microsoft Exchange, and a TPC-C-like workload running on top of an Oracle database. They characterize workloads by running them inside virtual machines and observing the I/O patterns from the hypervisor using a utility called vscaleStats which generates histograms for various I/O parameters. In [61], the authors use the vscaleStats utility in VMware ESX Server to model workloads and devices for I/O load balancing. They propose a storage resource scheduler (SRS) to manage virtual disk placement and automatic load balancing using Storage VMotion to migrate virtual hard disks from one datastore to another. Their SRS has three main components: 1) Model a workload using workload-specific characteristics 2) Model storage devices by monitoring device-dependent statistics and 3) Suggest VM disk migrations to improve overall performance and do load balancing across devices.

Regression algorithms have been previously applied to characterize the performance of workloads. In [112], Ould-Ahmed-Vall et al. present a comparison of several regression algorithms for computer architecture performance analysis of software applications. They compare the following algorithms: multiple linear regression, artificial neural networks, locally weighted linear regression, M5’ model trees, and support vector machines (SVMs). They collect a series of micro-architectural events, including those related to execution time, instruction mix, branches, memory subsystem, and translation look-aside buffers (TLBs). Their data is collected on a subset of the Standard Performance Evaluation Corporation (SPEC) CPU2006 workloads. As part of their results, they present the prediction quality and usefulness of the output from the algorithms, with multiple linear regression and model trees producing easily
interpretable results.

In [102], Mousa et al. apply multiple linear regression and model tree analysis to characterize performance in virtualized execution. They break up a workload execution into fixed-sized intervals based on the number of retired instructions. Within each interval, they gather counts of events derived from three primary sources: hardware performance monitoring counters (PMC), guest kernels, and the hypervisor (VMM). Using linear regression, the cycles per instruction (CPI) of the execution is decomposed into a linear combination of the architectural and virtualization events they collect. In [103], the authors present a low overhead profiling tool that automates the collection of hardware and software events spanning the vertical execution stack.

In our work, we break up a workload’s execution into fixed-size intervals based on (virtual) execution time. The events that we collect are those visible to the VMM. Working completely at the VMM layer offers several advantages. By harnessing the ability of the VMM to isolate and manage several virtual machines, it is possible to provide workload characterization at a common level across all VMs. In addition, being situated within the VMM provides ease of deployment as the tool is not tied to a specific OS and can be deployed transparently below different operating systems.

In their paper, Mousa et al. note that due to the dynamic behaviors introduced by virtualization, performance characterization demands high resolution yet minimally intrusive instrumentation. Another benefit of working strictly at the VMM layer relates to the low overhead introduced by our tool as it only requires instrumenting the VMM. An analysis of the current execution overhead of the VMM instrumentation shows that, in terms of wall clock time, the event extraction currently results in about a 10% performance degradation. In other words, adding event extraction to the VMM results in approximately 10% longer execution time.
In [45], an overview of the key characteristics of virtual environment benchmarks is given. They present two examples of benchmarks for consolidated servers. The first, Intel’s vConsolidate (vCon), consists of the following workloads: the CPU-intensive SPECjbb, SysBench [129] (an online transaction processing workload that runs transactions against a MySQL database), Webbench which uses the Apache web server, and a Microsoft Exchange mail server workload that runs transactions on Outlook. The second benchmark, VMware’s VMmark, includes data center workloads such as a mail server, Java server, web server, database server, and file server.

As part of any workload characterization, it may be possible to identify phases of execution within a program. In characterizing the behavior of a workload over time, changes in the regression coefficients produced by our workload characterization tool can provide insight into the time-varying behavior of the workload. The advantage of phase detection, as described in [11], is that it can be used to dynamically adapt multi-configuration hardware to program behavior.

### 3.3 Related Work in Feature Selection

Feature selection is a very important and well-studied problem in machine learning. Most of the work have focused on the area of feature selection for classification and regression [40, 82, 64, 131, 126], and as far as we know, there has been no work done to create a feature selection algorithm that caters specifically to outlier detection problems. In our work, we propose the first feature selection algorithm that takes the subsequent task of outlier detection into consideration and chooses features to intrinsically enhance the identification of outliers.
Chen et al. [35] developed a ranking-based feature selection algorithm for classification of high-dimensional datasets which suffer from the “small sample space” problem and whose class labels are highly imbalanced, the latter being a characteristic inherent in outlier detection. Recent work that look for outliers in high-dimensional datasets deal with the issue of high dimensionality in different ways. Aggarwal et al. [8] use an evolutionary search technique to find multiple lower dimensional projections of the data which are locally sparse in order to detect outliers. Other methods perform feature transformation, rather than feature selection, for outlier detection [109].

There has also been some work done that use the idea of “a ratio of densities” to directly perform outlier detection. Hido et al. [67] use the ratio of the density of a data point in the training set to its density in the test set as a measure of the degree to which the point is an inlier, as opposed to an outlier. Their training set consists of only normal points and the test set consists of both normal points and outliers. To deal with high-dimensional data, Sugiyama et al. [128] use a projection matrix to find the low-dimensional subspace in which the two densities are significantly different from each other. In [125], the novelty of data points in one distribution are assessed relative to another distribution based on the log-likelihood ratio of the two distributions. In our work, we use a ratio of densities to perform feature selection, with the distinction that our method uses the notion of local neighborhoods to measure densities. In the denominator, we utilize the density of only outliers. This ensures that we pick features in which outliers become even more conspicuous as they will be represented in low-density regions of the feature space. For the outlier detection, we take a one-class learning approach and distinguish outliers using well-established methods. In this regard, once features are chosen by our feature selection technique, the outlier detection algorithms proposed in [128] and [125] can be used as alternative methods.
for identifying outliers.
Chapter 4

Intrusion Detection with the VMM IDS

As with traditional computing systems, security remains an integral aspect of virtual computing environments. Anti-virus programs and firewalls can guard a system against known exploits, but these mechanisms provide little protection against new classes of attacks and insider threats. Virtualization can provide the ability to isolate and inspect VM-based execution. Virtual machines themselves are not completely immune to viruses and malicious attacks. To protect the guest OS running inside a virtual machine and guard against the existence of malicious software, or malware, there needs to be an intrusion detection system (IDS) in place.

Traditionally, an IDS can be categorized as one of two types: a host-based intrusion detection system (HIDS) or a network-based intrusion detection system (NIDS). An HIDS resides on the system that is being monitored and thus has the advantage of a rich view of the internal workings of the system. The disadvantage with this approach is that a malware can determine the existence of the HIDS and subsequently
compromise it or attempt to evade detection. An NIDS, on the other hand, performs intrusion detection from outside the target system, using information from the network flow. This makes it more resistant to attacks and evasion, but at the cost of poor visibility of the system.

In a virtualized execution environment, the virtual machine monitor (VMM) is a software layer that allows the multiplexing of the underlying physical machine between different virtual machines, each running its own operating system. In this work we implement a VMM-based IDS, a variant of host-based intrusion detection systems wherein the IDS resides on the physical host machine, yet remains outside of the virtual machine being monitored. As such, a VMM IDS is able to enjoy the advantages offered by both HIDSs and NIDSs: a rich view of the target system (the VM) combined with a greater resistance to attacks and evasion by the malware. The latter is one of the benefits of isolation provided by the VMM.

The VMM IDS only uses information available at the VMM-level to detect intrusions. There exists a large semantic gap between this low-level architectural data and the actual program behavior. Consequently, we utilize sophisticated data mining algorithms to extract meaningful and useful information to distinguish normal (non-malicious) from abnormal (malicious) behavior.

There are two main approaches to intrusion detection: misuse detection and anomaly detection. In misuse detection, the behavior of the system is compared to patterns of known malicious behavior, or attack signatures. A weakness of this approach is its inability to detect new and previously unseen attacks, known as zero-day attacks. In anomaly detection, a profile of normal behavior is built and any deviations from this normal profile is flagged as a potential attack. While anomaly detection has the ability to detect zero-day attacks, it is also prone to false alarms, i.e., previously
unseen normal behavior may incorrectly be identified as an attack. As virtualization and the information available to the VMM facilitate the profiling of normal behavior, in our VMM IDS we take the second approach to intrusion detection. We use system events visible to the VMM and incorporate data mining algorithms to help characterize normal execution patterns and distinguish deviating anomalous behavior, while trying to balance the trade-off between true detections and false alarms.

A key advantage that a VMM-level IDS provides is ease of deployment. Only the VMM needs to be modified to extract low-level architectural events during runtime. This ties the IDS to a particular VMM and instruction set architecture (ISA). No modification to the operating system is required. Hence, it can be deployed in any virtualized computing environment with minimal effort.

In our security work, we focus on virtualized server applications [120]. These applications are combined with a customized commodity operating system to run optimally in a virtual environment. As there are no login operations and typical execution consists of one main process running alongside background processes, we expect the normal behavior of these workloads to be fairly stable in time and space. Our IDS uses data mining algorithms to characterize the normal behavior of the workload. A malicious attack would introduce deviations from the normal behavior, which should be identified by the data mining algorithms and flagged by the IDS. Along these lines, a VMM IDS has the advantage of being able to detect zero-day attacks, in addition to previously known malware.

Next, we describe the details of our VMM IDS. The IDS consists of two key components:

- A front-end, whose duties include:
  - Event Extraction - Capturing the low-level architectural data available to
the VMM such as disk and network I/O accesses, page faults, translation look-aside buffer (TLB) flushes, and control register updates.

- **Feature Construction** - Using statistical techniques to transform the raw data into *features*, which are used by the data mining algorithms.

- A *back-end*, whose duties include:
  
  - **Feature Reduction** - Reducing the large space of possible features, which improves both the time complexity and the accuracy of data mining algorithms.
  
  - **Normal Model Creation** - Profiling the normal execution of the workloads and build a model of normal behavior.
  
  - **Anomaly Detection** - Identifying anomalous behavior as deviations from the model of normal behavior.
  
  - **Raising an Alarm** - Flagging behavior that deviates from the norm as a possible threat.

A high-level overview of our VMM IDS design is presented in Figure 4.1. There are two main phases of the VMM IDS operation: a *calibration* phase and a *testing* phase. In the calibration phase, the front-end extracts the VMM-level events and constructs all the possible features (using methods described in Chapter 2). These features are passed on to the back-end where feature reduction takes place. The reduced set of features are provided to the data mining algorithms to build a model of normal execution behavior. Next, anomaly detection is performed on a set of both normal and abnormal data points, assigning a score to each based on how much they deviate from the normal model. The scores are then passed through a filter to remove noise.
and determine when to raise an alarm.\textsuperscript{1} During the calibration phase, we evaluate the true detection and false alarm accuracy of our IDS to select an optimal set of features and filter configuration.

In the testing phase of the IDS, once the VMM-level events are extracted, only the reduced set of features are constructed. Using the model and IDS configuration from the previous phase, anomaly detection is performed on a previously unseen set of normal and abnormal data points. The scores assigned to them are passed through the filter to distinguish an appropriate time to raise an alarm.

To examine the effectiveness of our VMM IDS in detecting real-word attacks,\textsuperscript{1} when the alarm is raised, we assert that a malware has been found.
we evaluated the IDS on several different server workloads, injecting more than 300 malware obtained from a repository of real attacks. It is important that the IDS not only detect the malware, but do so within a reasonable amount of time. To this end, we present both the accuracy of the IDS (in terms of true detections and false alarms) and the time-to-detection results. We show that on average, we are able to correctly detect about 94% of the malicious attacks within about 20 seconds from the start of the attack, at a cost of only 3% false alarms.

The remainder of the chapter is organized as follows. In section 4.1, we describe the front-end of our VMM IDS, including the information we are able to extract from the VMM and how it is used to build features. In section 4.2, we review the approach taken by our back-end to best learn the normal behavior and identify malware. In section 6.4, we evaluate our VMM-based IDS in terms of its detection and false alarm rate, as well as its ability to detect intrusions in a timely manner. We discuss several aspects of our work in section 4.4. Lastly, we conclude the chapter in section 4.5.

Next, we describe the front-end and a back-end components of our VMM IDS framework in more detail. The front-end works together with a VMM to extract events. The events, which are collected dynamically, are used to construct features and sent to the back-end. The back-end is responsible for detecting abnormal behavior and then raising an alarm.

4.1 VMM IDS Front-End

The front-end of the VMM IDS has the responsibility of extracting low-level architectural events and subsequently transforming them into features used by the data mining algorithms. In Chapter 2, we described the VMM-level events that are captured
and the means by which the events are processed to generate rate and relationship features. Once the features are sent to the back-end and employed in subsequent steps, the utility of the features can be analyzed and used to provide feedback to the front-end. This can result in a reduction in the space of features generated and consequently the number of events that are extracted, thereby improving the performance of the front-end by lowering the overhead of VMM-level event extraction.

Next, we describe the steps performed in the back-end.

4.2 VMM IDS Back-End

The back-end is responsible for reducing the large space of possible features by selecting a subset of them. These features are then passed on to the data mining algorithms to build a model of the normal behavior of the workloads. Once this model is established, anomaly detection is performed on new windows to determine whether they deviate enough from the normal model to warrant the raising of an alarm. In the following sections, we shall delve into the details of these steps.

4.2.1 Feature Reduction

As previously mentioned, the space of possible features is very large. In order to perform accurate anomaly detection, we must select the most useful features to help differentiate normal from abnormal behavior. Along these lines, we reduce the feature space by performing feature subset selection, in which a subset of the original set of features is chosen. The search for the best features is guided by how well the features do when provided to the data mining algorithm. This gives us the ability to understand which features and consequently, which VMM-level events, are significant
in modeling the behavior of normal workload execution and distinguishing irregular, malicious activity. There are also benefits to having a small number of features, which include reducing the computational complexity of the data mining algorithm, as well as improving its accuracy by removing redundant and irrelevant features.

The feature subset selection algorithm we apply is the Sequential Floating Forward Selection (SFFS) algorithm [117], also known as Floating Forward Search. This is a simple, yet effective algorithm that has been shown to yield results that are very close to those obtained through an exhaustive search of all possible subsets of the feature space. At each step, the SFFS algorithm keeps a list of the currently selected features, along with a measure of how well those features do when evaluated on the data mining algorithm. In section 6.4, we describe the criterion we use in this work.

Initially, the algorithm begins with the empty set of features. There are two main phases of the algorithm: a forward step and a sequence of backward steps. In the forward step, all possible subsets of features that are the result of appending one feature to the current list of features are evaluated and the one with the highest criterion value is accepted, as long as it is better than the criterion value of the current list of features. After a forward step, a backward step is attempted. In the backward step, all possible subsets that are the result of removing a feature from the current list of features is evaluated. The best resulting criterion value is compared to that of the recorded best value, for that particular number of features. If the new best value of the criterion is better than the recorded best, the corresponding feature is removed, the best criterion value is updated (for that feature count), and another backward step is attempted. Otherwise, a forward step is attempted. This process is iterated until no backward or forward step results in an improvement in the evaluation criterion.
4.2.2 Model Creation and Anomaly Detection

Once a subset of the features is selected, the window data points only need to contain feature-values that correspond to the chosen features. The windows can now be provided as input to the data mining algorithms. Using only normal windows, we train the classifier to produce a model of normal activity. For this we employ a profiling approach that does not require or make assumptions about malicious behavior, other than that it is “different” from normal behavior. Such an approach fits well with deployment in real production environments. For example, new servers are often “stress tested” for many days or weeks with actual or realistic workloads before they are deployed. During this period, the behavior of the (virtualized) server can be profiled, and substantially different behavior encountered post-deployment can be flagged as potentially malicious.

For any data mining task, the dataset can be divided into three sets: a training set, a validation set, and a test set. The training set is used to build the data mining model. Since we build a model of normal behavior, our training set is comprised of only normal windows, i.e., those that represent time periods in which there is no malicious activity. Once the model is created, its performance is evaluated on a validation set. Our validation set contains both normal and abnormal windows. We evaluate how well the model of normal behavior is able to distinguish abnormal windows (true detections), while making sure that normal windows from the validation set are not incorrectly identified as malicious (false alarms). By tuning the parameters of the intrusion detection process, it is possible to achieve a reasonable trade-off between the true detections and false alarms. Thus, the validation set can be used to find good values for the unknown parameters (in our case, an optimal subset of features and filter configuration). Once these parameters are chosen, they are evaluated on
the test set to see how well the IDS (using those parameters) performs on previously unseen normal and abnormal windows.

As described in Chapter 2, we employ two well known techniques to build our VMM-based IDS: the K-Nearest Neighbor (KNN) and the Local Outlier Factor (LOF) algorithms. Once we train the model on a dataset consisting of only normal data, we assign decision values to data points from a test set consisting of both normal and abnormal data. The decision values provide a measure of how abnormal the test data points are with respect to the model of normal execution. Based on these decision values, we can now choose whether or not to raise an alarm. We describe the method by which we do so in the next section.

4.2.3 Raising an Alarm

When we suspect that an attack has occurred, we raise an alarm. The decision values assigned to the time-based windows are what we use to decide whether to raise the alarm. As we described in section 2.4 of Chapter 2, these windows are found by sampling the event stream. Since it is not clear where one phase of behavior (either normal or abnormal) ends and the next one begins, we must deal with the issue of over-sampling or under-sampling. This introduces noise into the time-based windows. For the case of normal windows, this noise can cause one or two contiguous windows to appear to be outliers, increasing the probability that a high decision value is assigned to them and resulting in a false alarm. To address this issue, we apply a filter on the sequence of decision values to help smooth out the values and lower the false alarm rate. This allows us to filter out the noise introduced by sampling the time-based windows, while remaining sensitive to the intrusion.
Figure 4.2: Weights of an exponential distribution in the EWMA model

**Exponentially Weighted Moving Average Filter**

The exponentially weighted moving average (EWMA) model is often used to smooth out fluctuations in time series data [29]. It has also been applied to the area of intrusion detection by identifying anomalous changes in event intensity [142]. Given a sequence of decision values, the EWMA model applies a set of weights to them that decrease exponentially the further back they are in time, giving more importance to recent values while not entirely discarding previous values. The relative importance of recent decision values can be tuned using the parameter $\alpha$. Figure 4.2 provides an example of the weights of an exponential distribution used in the EWMA model. The sequence of EWMA values are found using the following formula:

$$EWMA_i = \alpha \cdot v_i + (1 - \alpha) \cdot EWMA_{i-1}$$  \hspace{1cm} (4.1)
where $EWMA_i$ denotes the EWMA value at time $i$, $v_i$ is the decision value at time $i$, and $\alpha$ (the decay rate) determines the weight associated with more recent decision values. An $\alpha$ close to 1 gives more importance to recent decision values, whereas an $\alpha$ close to 0 better reflects values seen in the past. It can be represented as a function of the effective filter width, $N$, as: $\alpha = \frac{2}{N+1}$.

If the EWMA value is higher than a threshold, we suspect an attack has occurred and raise the alarm. The EWMA filter has the nice property that by smoothing out fluctuations in the decision values, it can lower the probability of raising a false alarm. The only caveat is that it may also reduce the probability of correctly raising an alarm. Hence, it is important to try different values of the threshold and effective filter width to find a configuration that provides a good balance between the true detections and false alarms. To illustrate the method by which we raise an alarm, we provide a time series plot of the decision values versus the window number in Figure 4.3 for the Exchange.Heavy workload. Normal windows are shown with the ◦ symbol and abnormal windows are represented with the + symbol. The solid, horizontal line corresponds to the threshold, set at 1.81. The dashed line represents the EWMA values, calculated using an effective filter width ($N$) of 40. Note that window 112 is assigned a high decision value (9.92), which causes the EWMA value to rise to 1.50. Since it does not exceed the threshold, we refrain from raising an alarm at that point and thereby avoid a false alarm. Instead, we raise the alarm at window 304 where the EWMA reaches a value of 1.95.

To summarize, the approach of the back-end for detecting malware using information extracted at the VMM-level consists of the following main steps:

1. Use Sequential Floating Forward Selection to select features that can best characterize normal behavior and distinguish normal from abnormal behavior.
Figure 4.3: Time series plot illustrating the alarm-raising mechanism

2. Apply data mining algorithms to build a model using normal data and assign a decision value to each new data point, quantifying how abnormal it is relative to normal data points.

3. Use an exponentially weighted moving average (EWMA) model to smooth out fluctuations in the decision values and determine when to raise an alarm.

The steps described above fall nicely into our vision in which our VMM IDS is automatically calibrated and customized to the behavior of the virtual appliance. In our current work, we relax the constraint of utilizing only normal data in two of the steps: feature reduction and choosing an optimal EWMA filter configuration. In other words, we use both the normal and abnormal data to select the features and
choose an EWMA filter, which we will explain in section 6.4. Conversely, training of the data mining algorithms is done with normal data only.

Once we have used the normal data to generate a model of normal behavior, without any prior knowledge of malicious behavior, we evaluate the model using abnormal data to determine an optimal set of features and the filter configuration. This is similar to a real system where we look for sensitivity to different parameters and adjust the system accordingly. Therefore, our approach is not strictly unsupervised; training of the classifier is done using one-class learning and the IDS calibration uses supervised learning.

4.3 Evaluation

In this section we provide evidence that an effective VMM IDS can be constructed. The section is organized in the following way. First we review our experimental setup. Next, we discuss the features selected and information extracted. Finally, we present the accuracy results we obtain using our two data mining techniques and discuss the advantages of using both these techniques together.

4.3.1 Experimental Setup

In our work, we use the open-source VirtualBox [39] VMM (edition 2.2) developed by Oracle Corporation as part of its family of virtualization products. Virtualbox is a full virtualization environment wherein the VMM runs on a commodity operating system (referred to as the host OS). The OS running inside a virtual machine that executes above the VMM is referred to as the guest OS. Figure 4.4 illustrates the structure of a computer system running VirtualBox.
All the experiments are executed on a Dell XPS710 equipped with an Intel Core2 processor (2 cores) running at 1.86 GHz with 4 GB of RAM. The host operating system is Windows XP (SP2).

The target deployment for our VMM-based IDS is to secure virtual machine appliances. Each appliance is usually prepackaged with a commodity OS (Windows or Linux) and a software stack that is configured to perform one or more specific tasks [107]. To this end, we set up different classes of servers, virtual appliances, and workloads as shown in table 4.1. These systems are chosen to reflect a broad range of behaviors (CPU intensive workloads, disk accesses, network I/O, etc.).

The ab workload is the Apache HTTP server benchmarking tool [53]. It is designed to measure performance in http requests per second. The TPC-C like workload is an on-line transaction processing benchmark [13]. It generates a pseudo-random sequence of client accesses that create a stream of random reads and writes. We
create a mailserver using Microsoft exchange. LoadSIM [99] is a benchmarking tool simulating clients of an Exchange server. We configure LoadSim to generate two distinct workloads. In the first workload, Exchange.Light, we configure LoadSim to simulate 32 clients with a medium load. In the second workload, Exchange.Heavy, we configure LoadSim to simulate 64 clients with a heavy load.

To generate a cloud storage workload, we use the IOzone [3] benchmark. IOzone is a filesystem benchmark tool which generates and measures various file operations. The Tonido software [1] is used to provide cloud storage services. Tonido is a home server network attached storage (NAS) software. By installing the Tonido Desktop software on a computer, it effectively turns the computer into a cloud storage server. Synchronization between the server and remote clients is provided by installing the TonidoSync Client on remote computers. In the virtual machine, we install Tonido Desktop software for Windows (version 2.51) and on a remote client running Windows 7, we install the TonidoSync Client.

Table 4.1: Normal workloads (virtual appliances)
We use these virtual appliances and more than 300 real-world malicious executables to generate our normal and abnormal workloads. All malware binaries are taken from Malfeas [111], an online repository, and are unique by their MD5 checksum. They include very recent (2008 & 2009), real and unknown (zero-day) malware. The malware can be characterized into four main classes of attacks, based on their general behavior. These classes are:

- **Trojan Horse** – A malicious program disguised as something innocuous or desirable. It contains hidden functionality and performs unwanted or malicious activity when activated.

- **Downloader** – A malicious program that connects to the Internet and downloads other Trojan horses or components.

- **Backdoor** – A method of bypassing normal authentication procedures. Once a system has been compromised, backdoors may be installed in order to allow easier access in the future. Backdoors may also be installed prior to malicious software, to allow attackers entry.

- **Infostealer** – A generic class for a Trojan horse that steals online game accounts, such as Lineage, Ragnarok online, Rohan, and Rexue Jianghu.

A normal trace is generated by simply executing our server and its (normal) workload generator for a period of 1 hour. At the end of the execution, we restore the server to a previously set checkpoint to ensure consistent starts to the normal and abnormal traces. This is repeated four times for a total of five hour-long normal traces. These traces are then merged before being processed to generate features and used to build the model of normal activity.
The appliances and normal workloads are also used to generate the abnormal traces. An abnormal trace is generated by executing our server and normal workload, and at a predefined point in time, injecting and executing the binaries of a real-world malware. The abnormal executions are each 15 minutes long and contain both normal and abnormal windows. They consist of a 10-minute period of normal activity, after which a script initiates a malware's executable. This is followed by an additional 5 minutes of execution time. Note that the malware is executed in parallel to the normal workload. After the malware has executed for 5 minutes, we shutdown the VM and restore it to our original checkpoint. This procedure is repeated for each malware (and each appliance). The abnormal traces consist of more than 300 malicious executables per appliance.

For each execution, the stream of events extracted at the VMM-level are output to a text file. This trace file is stored for later use by the back-end (including feature reduction and normal model creation). For deployment in a real production environment, there is no need for the traces to be stored; the events can be used to construct features and discarded thereafter. This enables the IDS to be deployed in an efficient, online manner. Furthermore, in a real production environment, the feature reduction and normal model creation tasks need only be performed periodically, allowing the corresponding features and model to be used until such time as it is deemed necessary to update them. Hence, once the VMM-level events are extracted and used to construct the (selected) features, anomaly detection can take place to determine if the resulting window of feature-values contains malicious activity.
Tagging Malicious Activity

To evaluate our IDS, we need to generate labels for data points (windows), indicating whether they are normal or abnormal. This is straightforward for the normal traces as all data points are labeled normal. Since our abnormal traces include both normal and abnormal activity, we need to differentiate between the two. Differentiating normal from abnormal behavior is difficult when only low-level information from the VMM is available. To resolve this issue, we developed a debugging environment that allows us to validate which low-level events are related to abnormal processes of malicious activity and estimate which low-level events are related to normal processes. Each window corresponds to the events generated by possibly many processes. If a window contains any event generated by any malicious process, then the entire window is tagged as “malicious”; only if a window is free of events generated by malicious processes is it tagged “non-malicious.” Our IDS is then evaluated by testing if it can correctly flag windows that contain any malicious activity.

4.3.2 Feature Analysis

As described previously, we use Sequential Floating Forward Selection (SFFS) to select features that best characterize the normal behavior and which are able to distinguish normal from abnormal activity. In this study, we use LOF as the data mining algorithm incorporated in the feature reduction task. Once the features are selected, we build a model of normal execution and perform anomaly detection using KNN, in addition to LOF. In this way, we can see how well the features chosen to optimize the LOF results generalize to another data mining technique (KNN).

The criterion we chose to optimize during feature reduction is the minimum Euclidean distance to the ideal operating point on the ROC curve (cf. Chapter 2). Each
of our abnormal traces can be divided into two parts: (1) all the windows from the beginning until the point where the malware was injected, and (2) the windows from the malware injection point until the end. Since there is no malicious activity in the left part of the trace, we can treat it as a “normal” data point. The IDS must not raise an alarm during this part of the trace, otherwise it will have produced a false alarm.\textsuperscript{2} The malware is injected during the right portion of the trace and hence it can be treated as an “abnormal” data point. The IDS must raise at least one alarm during this portion of the trace, otherwise it will have produced a false negative (the inability to correctly raise an alarm when it should have been raised). With this viewpoint of an abnormal trace, we can produce proper ROC curves for the collection of abnormal traces.

Each point on the ROC curve is found by varying the threshold value on the EWMA filter output. Since the effective filter width ($N$) also affects the true detection and false alarm rate, we vary $N$ from 1 to 50 and produce their corresponding ROC plots. Then, we find the distance from all the points on the ROC curves to the point (0,1). The smallest distance provides an indication of how well the set of features performed.

Feature reduction was performed for each of our workloads. The features themselves possess low-level architectural information which, for the most part, can be difficult to correlate to high-level program behavior. That is where feature reduction can be a valuable tool to reduce the space of possible features, removing irrelevant and redundant features while maintaining sufficient information to help distinguish normal from malicious behavior.

We analyzed the features chosen and made several observations. Most of the\textsuperscript{2} Since the left portion of the trace is equivalent to a single normal data point, multiple alarms raised during this portion of the trace are counted as a single false alarm.
selected features correspond to events related to page faults, task switches, privilege level changes, disk I/O, and network I/O. Both rate and relationship features were chosen as they each provide different and complementary information. A list of the features selected for each workload, along with a brief description of each, are provided in Table 4.2 through Table 4.6. We also specify whether a feature is based on VM events or VMM events and whether it is a rate or relationship feature.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Feature Type</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR3_WRITE</td>
<td>VM/Rate</td>
<td>Number of writes to control register 3 (CR3)</td>
</tr>
<tr>
<td>INVALIDATE_PAGE</td>
<td>VM/Rate</td>
<td>Number of page invalidation events</td>
</tr>
<tr>
<td>CPL_SET_0.VS_NETWORKIO[numRuns]</td>
<td>VM/Relationship</td>
<td>Number of runs for setting the current privilege level to 0 events versus network I/O events</td>
</tr>
</tbody>
</table>

Table 4.2: Features selected for Apache workload

<table>
<thead>
<tr>
<th>Feature</th>
<th>Feature Type</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR3_WRITE</td>
<td>VM/Rate</td>
<td>Number of writes to CR3</td>
</tr>
<tr>
<td>PAGE_FAULT_P_1</td>
<td>VM/Rate</td>
<td>Number of page faults caused by a page-level protection violation</td>
</tr>
<tr>
<td>PAGE_FAULT_WR_0.VS_PAGE_FAULT_WR_1[numRuns]</td>
<td>VM/Relationship</td>
<td>Number of runs for page faults due to reads vs. page faults due to writes</td>
</tr>
<tr>
<td>CR3_WRITE_VS_PAGE_FAULT[MannWhitney]</td>
<td>VM/Relationship</td>
<td>Measure of the randomness between writes to CR3 and page fault events</td>
</tr>
<tr>
<td>CR3_WRITE_VS_TRAP[avgRunValue]</td>
<td>VM/Relationship</td>
<td>Average run value for writes to CR3 vs. trap events</td>
</tr>
</tbody>
</table>

Table 4.3: Features selected for MySQL workload

A feature that facilitated the detection of attacks for the cloud storage workload is one that evaluates the relationship between disk read and network transmit events. This is consistent with the cloud storage server’s activity in reading from the disk.
Exchange.Light

<table>
<thead>
<tr>
<th>Feature</th>
<th>Feature Type</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGE_FAULT_P_1</td>
<td>VM/Rate</td>
<td>Number of page faults caused by a page-level protection violation</td>
</tr>
<tr>
<td>PAGE_FAULT_US_0</td>
<td>VM/Rate</td>
<td>Number of page faults that occur while in system mode</td>
</tr>
<tr>
<td>CR3_WRITE_VS_CR4_WRITE[avgRunValue]</td>
<td>VM/Relationship</td>
<td>Average run value for writes to CR3 vs. writes to CR4</td>
</tr>
<tr>
<td>TSS_WRITE_VS_TLB_FLUSH[MannWhitney]</td>
<td>VM/Relationship</td>
<td>Measure of the randomness between writes to the TSS and TLB flushes</td>
</tr>
</tbody>
</table>

Table 4.4: Features selected for Exchange.Light workload

and sending updated files and folders over the network to remote clients as part of its synchronization task. Among the features for the Apache workload is a rate feature that counts the number of invalidate page events and a feature that shows the relationship between setting the current privilege level to 0 and network I/O events. As Apache is a network-intensive workload, we expect to see features based on network I/O events playing a role in characterizing its normal execution. A relationship feature chosen for the MySQL workload counts the number of runs for read events that caused a page fault versus write events that caused a page fault. Page faults can provide information about the memory access patterns of the workload. For MySQL, which is disk-intensive, it is intuitive that accesses to random locations on the disk would entail page faults due to the frequent swapping between memory and disk.

A rate feature found in common among three of the workloads (MySQL, Exchange.Light, and Exchange.Heavy) counts the number of page fault events that occurred due to a page-level protection violation. The MySQL and Apache workloads chose a rate feature that counts the number of writes to control register 3 (CR3), indicating that the rate of context switches can be useful in recognizing malicious from normal activity. The Exchange workloads both selected relationship features...
Table 4.5: Features selected for Exchange.Heavy workload

<table>
<thead>
<tr>
<th>Feature</th>
<th>Feature Type</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGE_FAULT_P_1</td>
<td>VM/Rate</td>
<td>Number of page faults caused by a page-level protection violation</td>
</tr>
<tr>
<td>PAGE_FAULT_US_0</td>
<td>VM/Rate</td>
<td>Number of page faults that occur while in system mode</td>
</tr>
<tr>
<td>CR3_WRITE VS CR0_WRITE[MaxRunValue]</td>
<td>VM/Relationship</td>
<td>Maximum run value for writes to CR3 vs. writes to CR0</td>
</tr>
<tr>
<td>CR3_WRITE VS CR4_WRITE[MinRunValue]</td>
<td>VM/Relationship</td>
<td>Minimum run value for writes to CR3 vs. writes to CR4</td>
</tr>
<tr>
<td>CPL_SET_0 VS DISKIO[MannWhitney]</td>
<td>VM/Relationship</td>
<td>Measure of the randomness between setting the current privilege level to 0 events and disk I/O events</td>
</tr>
<tr>
<td>EXECUTION_MODE_ENTER_REM, VS PAGE_FAULT_US_0[MannWhitney]</td>
<td>VMM/Relationship</td>
<td>Measure of the randomness between events related to entering recompiled execution mode and page fault events that occur in system mode</td>
</tr>
</tbody>
</table>

that measure the length of consecutive writes to different control registers when interleaved with writes to other control registers (e.g., writes to CR3 vs. CR4). They also selected a rate feature that counts the number of page faults that occurred while in supervisor mode (i.e., ring 0).

We also observed that VM events are predominantly used to construct the selected features. For the set of workloads that we studied, only one feature based on a VMM event was selected and this occurred for only one workload (Exchange.Heavy). It is a relationship feature that applies the Mann-Whitney test to evaluate the random interleaving of the following two events: (1) an event distinguishing that the VMM has entered a different state of execution wherein it intercepts system-level requests and recompiles code, and (2) page faults that occurred while executing in the supervisor mode.

In our analysis, we have seen correlation among VM and VMM events which
Cloud Storage

<table>
<thead>
<tr>
<th>Feature</th>
<th>Feature Type</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISKIO_READ VS NETWORKIO_WRITE[minRunValue]</td>
<td>VM/Relationship</td>
<td>Minimum run value for disk reads vs. network writes</td>
</tr>
<tr>
<td>DISKIO_READ_BYTES VS DISKIO_WRITE_BYTES[avgRunValue]</td>
<td>VM/Relationship</td>
<td>Average run value for the size of disk reads vs. disk writes</td>
</tr>
<tr>
<td>CR3_WRITE VS CR0_WRITE[maxDiscrepancy]</td>
<td>VM/Relationship</td>
<td>Maximum discrepancy for writes to CR3 vs. writes to CR0</td>
</tr>
<tr>
<td>CR3_WRITE VS TRAP[maxDiscrepancy]</td>
<td>VM/Relationship</td>
<td>Maximum discrepancy for writes to CR3 vs. trap events</td>
</tr>
<tr>
<td>CR3_WRITE VS TRAP[MannWhitney]</td>
<td>VM/Relationship</td>
<td>Measure of the randomness between writes to CR3 and trap events</td>
</tr>
</tbody>
</table>

Table 4.6: Features selected for Cloud Storage workload

may explain why the features selected are mostly those based on VM events. In other words, features based on VMM events do not seem to provide much additional information. Since most of the VMM events are specific to the VMM in our study (VirtualBox), by removing these events and their corresponding features, we can still achieve a good detection rate. This shows the robustness of our IDS with respect to the underlying VMM. As it does not need to be tied to a specific VMM and the majority of the collected events can be found in VMMs that are to similar to Xen, it should produce comparable results across various VMMs.

Based on the features selected, we determined that not all events are equally important. Filtering out unnecessary or infrequently-used events (with respect to the selected features) can optimize the execution performance of the VMM IDS front-end.

### 4.3.3 Experimental Results

Once the features are selected, we run experiments using the KNN and LOF algorithms to evaluate their ability to differentiate abnormal from normal execution. In
order to find the best value for the parameter \( k \) (size of the neighborhood), an appropriate distance metric, and an accurate filter configuration (threshold and effective filter width) to use for each technique, we run the data mining algorithms on the validation set. The validation set consists of a randomly chosen 90\% of the approximately 300 abnormal traces (each containing a malicious attack). Using the validation set, we determine appropriate values to which we should set the parameters of the IDS. The remaining 10\% of the abnormal traces comprise our test set. We use this data to evaluate the ability of the IDS to accurately identify unseen malware. The training set consists of five normal traces for which no malware has been inserted, for the purpose of building a model of normal activity.

We experimented with several distance metrics, including Euclidean distance (\( L_2 \)-Norm), Manhattan distance (\( L_1 \)-Norm), Canberra distance [46], and cosine similarity. The results showed that on all of the workloads, the data mining algorithms performed well using the Euclidean distance.

Once the parameters of the IDS have been set using the validation data, we are now ready to test our IDS against unseen malicious attacks. We use the filter configuration found from the validation set; we apply the IDS using this filter configuration to our test set of feature traces. As discussed in section 4.3.2, we measure performance in terms of the true detection and false alarm trade-off achieved. In addition, we are concerned with how quickly the IDS is able to flag a malicious attack. To this end, we produce a plot of the true detection rate versus the time-to-detection (in units of windows). In an abnormal trace, an alarm raised before the injection of the malware will result in a false alarm. The time-to-detection for such a trace would effectively be a negative value. Even if an alarm is also raised at some point after the injection of the malware, it still does not justify counting it as a successful malware detection.
Hence, when producing the time-to-detection results, we restrict ourselves to showing the results for only traces in which the IDS was able to do everything right: it raised an alarm after the injection of the malware and not before it. The $x$-axis represents the delay from the point where the malware is injected until the alarm is raised. The $y$-axis provides the fraction of traces for which an alarm is correctly raised during that time. Figure 4.5 presents this timeliness plot for the MySQL workload. For MySQL, the IDS using the LOF algorithm is able to detect 100% of the malware instances with a false alarm rate of 6%. Thus, in the timeliness plot, we show the time-to-detection for the remaining 94% of the traces wherein no false alarm was raised.

![MySQL Testing Timeliness Plot](image)

Figure 4.5: Timeliness plot for the MySQL test traces

Note that although the malware is injected at a specific point in time during a normal workload execution, it may be the case that the malware does not begin
showing abnormal behavior until a later time, delaying the time-to-detection by the data mining algorithms. Thus, to describe what happens in the vast majority (95%) of the cases, we calculate the time-to-detection at the 95% level. This gives us an indication of the delay (from the injection point of the malware) to detect a malicious attack in 95% of the tests where we correctly raise a true alarm, without raising a false alarm. The detection and false alarm results on the test set, along with the time-to-detection at the 95% level are presented in Table 4.7.

<table>
<thead>
<tr>
<th>Server</th>
<th>Accuracy</th>
<th>Time-To-Detection (95% Level)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Detections</td>
<td>False Alarms</td>
</tr>
<tr>
<td>Local Outlier Factor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>MySQL</td>
<td>100%</td>
<td>6%</td>
</tr>
<tr>
<td>Exchange.Light</td>
<td>84%</td>
<td>3%</td>
</tr>
<tr>
<td>Exchange.Heavy</td>
<td>87%</td>
<td>3%</td>
</tr>
<tr>
<td>Cloud Storage</td>
<td>98%</td>
<td>3%</td>
</tr>
<tr>
<td>K-Nearest Neighbor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>100%</td>
<td>13%</td>
</tr>
<tr>
<td>MySQL</td>
<td>100%</td>
<td>6%</td>
</tr>
<tr>
<td>Exchange.Light</td>
<td>94%</td>
<td>3%</td>
</tr>
<tr>
<td>Exchange.Heavy</td>
<td>73%</td>
<td>17%</td>
</tr>
<tr>
<td>Cloud Storage</td>
<td>97%</td>
<td>3%</td>
</tr>
</tbody>
</table>

Table 4.7: Accuracy and time-to-detection results on test dataset

The overall LOF results are very promising. On the Apache workload, which exhibits a more stable normal execution behavior, LOF achieves optimal results; 100% of the malware are detected with no false alarms. In addition, for at least 95%
of the traces, the malware is detected within the first window of its execution. LOF is also able to detect all of the malware executed on the MySQL workload, at a cost of 6% false alarms and (at the 95% level) within 11 windows (about 22 seconds) from the introduction of the malware. For the Exchange workloads, over 80% of the malware are detected with 3% false alarms and (at the 95% level) at most 32 windows (64 seconds) from where the malware was injected.

Although the features chosen during the feature reduction task are those which try to optimize for the LOF performance, we see that they are still able to provide KNN with the ability to detect a large percentage of the malware, but at times, with a higher false alarm rate. In the case of the Exchange.Light workload, KNN is able to identify a greater percentage of the malware with the same false alarm rate as LOF. The reason for this could be that the closest normal windows to the malicious windows are in low-density areas (due to the lighter load of Exchange.Light). This would cause them to be assigned a low LOF decision value, making them indistinguishable from normal behavior. KNN, on the other hand, takes a distance-based approach to identifying anomalous windows and hence determines that, according to the distances to its $k$-nearest neighbors, the malicious windows warrant a high KNN decision value. This results a higher value of the EWMA filter, increasing the probability of correctly raising an alarm.

For the Exchange.Light workload, the IDS using the LOF algorithm achieves a true positive rate of 84% with a false alarm rate of 3%. This means that in the worst case, out of the 84% of the traces in which an alarm was correctly raised after the injection of the malware, 3% of them may have also raised a false alarm.\textsuperscript{3} In this case, at least 81% of the true positives were instances where the IDS was able to correctly

\textsuperscript{3}Upon further examination of the results, there was only one trace which raised a false alarm and it happened to have raised a true alarm as well.
raise an alarm without raising a false alarm. Considering the difficulty of the problem and the extremely limited information available at the VMM level, these results are remarkably encouraging.

Given the operating point\(^4\) at which we present our results, we achieve a high detection rate with a moderate false alarm rate. We can also back off on the detection rate, in favor of improving the false alarm rate. In addition, we can apply more aggressive filtering and thresholding to further reduce the false alarm rate. The ultimate choice of operating point and detection/false alarm trade-off is determined by the application and can be selected accordingly.

### 4.4 Discussion

In this section we discuss several issues that came up during our work.

**Generalization of Results and IDS Robustness** - In our work, we trained the classifier on 90% of the data and tested on the remaining 10% (over 30 malware). To study the robustness of the VMM IDS, we performed cross-validation experiments wherein we calibrated the IDS on three of the four classes of malware (described in section 4.3.1) and tested on the fourth class. The IDS had an average of 94% true detections at a cost of about 9% false alarms. This shows the robustness of our approach in terms of identifying unseen classes of malware.

**Evading the VMM IDS** - A weakness shared by most types of IDSs is one in which an attacker can study the defense methods and create new attacks evading the detectors. Although a VMM IDS is not immune to this kind of attack,

\(^4\)The operating point determines the threshold and filter configuration of the system.
we believe it would be much harder to accomplish. An attacker would need to generate a low-level footprint such that it is either identical to the normal workload executing, or is very light. This task is both difficult to accomplish and is highly dependent on the target machine normal workload. To successfully create a mimicry attack, the attacker would need to not only know the normal workload, but also the type of events that we monitor and the distribution of the events, which are reflected in rate features. For the relationship features, he would need to mimic the order of the events, as well as the randomness with which the different events are interleaved. Many of the events that we track are generated by the OS, so not only should the attack be similar to the workload, but so should the interaction between the OS and the attack. It is not enough for the attack to generate a workload that is similar to that of the normal workload. The footprint of the whole system with the attack should stay the same. For example in the case of a MySQL server, the attack must maintain the same rate of reads and writes to the disk. With an attack, there is likely to be a larger number of page faults, context switches (due to the inclusion of more processes), and other system-related events. The system as a whole will have a different footprint. If an attack is lightweight, it would be more difficult for the VMM IDS to identify. Nonetheless, based on the high detection rate of our IDS on the diverse set of approximately 300 malware, which range in their intensity of attack, we see that the VMM is still able to identify some lightweight attacks.

**Timeliness** - Timely detection is one of the main goals of this IDS. It is clear that a timely and early detection is preferable. Our IDS is able to detect most malware within a minute introduction. Although the detection is not always immediate,
it is better to detect an attack after a few minutes than not at all. And while some damage can be done in the meantime, it is confined to one VM.

**Response** - Generating a response to an attack is an issue left for future work. This issue is not at all trivial as little semantics are available. Initial discussion lead us to believe that OS support can be used to interpret the low level data, identify the malware, and generate a report useful for system administrators.

Additionally, several actions can be taken to resolve the attack. For example, a breached guest VM can be put offline while the attack is analyzed or it can be discarded and destroyed. Moreover, in many cases, appliances can roll back to a last known good configuration (a checkpoint). This action is especially easy to accomplish in a VM environment.

**Execution Overhead** - An analysis of the current execution overhead of the VMM IDS shows that, in terms of wall clock time, the event extraction currently results in about a 10% performance degradation. In other words, adding event extraction to the VMM results in approximately 10% longer execution time. In the future, the execution overhead can be further reduced. This can be accomplished by filtering out unimportant events, thereby removing the need to extract them. Such a process can be guided by the feature reduction task, by means of identifying which events were not used to construct any of the selected features.

**Back-end Performance** - Both KNN and LOF are compute intensive algorithms. While training can take minutes to hours to complete, much of this processing can be done offline. When this system is deployed on a live system, the back-end classification should not be the gating performance factor. A single classification
run on an X86 dual core can take on the order of minutes. This classification can be performed on a Graphics Processing Unit and completed in under a second [10].

4.5 VMM IDS Conclusions

A VMM-based IDS increases the ease of deployment across different operating systems and versions, and as part of a VMM, offers high manageability for server appliances. VMM-based IDSs break the boundaries of current state-of-the-art IDSs. They represent a new point in the IDS design space that trades a lack of program semantics for greater malware resistance and ease of deployment.

In this work, we implemented and evaluated a VMM-based IDS. The open source edition of VirtualBox was used to construct the front-end. We presented the types of information we were able to extract from the VMM and described the procedure used to build features. We also provided an analysis of the important features selected by our feature reduction technique, and discussed the corresponding events that were integral for distinguishing normal from abnormal behavior. Data mining algorithms were utilized as powerful techniques to bridge the semantic gap between the low-level architectural data and actual program behavior. Our results showed that there is enough information embedded within the VMM-level data to be processed and mined to accurately detect an average of 94% of real-world malicious attacks on server appliances and do so in a timely fashion, at a cost of only 3% false alarms.
Chapter 5

Workload Characterization with the VMM Application Modeler

Virtualization has quickly grown in popularity due to its attractive benefits. Some of these features include: improved resource utilization, manageability, and energy savings in data centers and cloud computing, increased availability and reliability of IT services, fast deployment of services and infrastructure, and greater security of virtual machines. In this chapter, we propose an added benefit of virtualization: workload characterization from the vantage-point of the virtual machine monitor using the VMM Application Modeler. Workload characterization is performed by reducing a workload’s behavior down to a mix of simpler behaviors exhibited by our canonical workload set. As described in Chapter 2, the canonical workload set consists of the following workloads: CPU, memory, disk read, disk write, network receive, and network transmit.

Working at the VMM-level, versus the VM-level or guest OS-level, significantly reduces the overhead introduced into the runtime by the profiling system. In addition,
it limits the amount of perturbation introduced into the system, thereby reducing Heisenberg effects where the phenomenon under observation is altered or lost due to the measurement itself [36]. On the other hand, raw VMM-level profile data provides limited program semantic information that is typically available at the application or operating system interfaces. Thus, we harness the power of regression algorithms to manipulate processed, low-level events present at the VMM layer and model high-level workload behavior. We rely heavily on our ability to tap into the wealth of information that lies within the VMM-level data to reconstitute an accurate picture of the high-level program semantics.

We utilize VMM-level raw events present in a number of well-behaved workloads to reconstruct the behavior of a real-world application. Again, we only consider information captured at the VMM-level. The extent to which a program interacts with the VMM can provide insight into the program’s behavior. For example, a CPU-intensive program that mainly executes user-mode instructions should not require significant interaction with the VMM. For a program that is disk-intensive, the VMM will need to intercept requests to access the disk, triggering events that are unique to virtualized environments such as “enter recompiled execution mode (REM)” (cf. section 2.3 of Chapter 2).

The ability to reduce an application’s behavior down to a mix of simpler behaviors can provide great insight into the performance and execution characteristics of an application. We represent the set of simpler behaviors by a set of canonical workloads. For each workload in the canonical set, we capture low-level architectural data available at the VMM layer such as disk and network I/O accesses, page faults, and control register updates. Since we collect this at the VMM level, we can collect this information at one point for all VMs and identify unique processes in each VM. This
raw data is then processed to produce a set of features on a per process or per VM basis. The features from each of the canonical workloads, along with the features of a target\(^1\) workload are provided as input to regression algorithms. The regression algorithms produce a model of the target workload’s activity as a linear combination of the activity of the canonical workload set. From the model, we can gain an understanding of the underlying behavior of the target workload. This model can help identify performance bottlenecks, as well as detect transitions in behavior to provide phase detection.

Our VMM Application Modeler consists of two key components:

- **A front-end**, whose capabilities include:
  - *Event Extraction* - Capturing the low-level architectural data available to the VMM.
  - *Feature Construction* - Using statistical techniques to transform the raw data into features, which are used by the regression algorithms.

- **A back-end**, whose functionality includes:
  - *Regression* - Applying multiple linear least-squares regression and the Least Absolute Shrinkage and Selection Operator (LASSO) [132] algorithm to the features generated from a workload in order to characterize and build a model of the workload’s behavior.

A high-level overview of our VMM-based workload characterization design is presented in Figure 5.1. The front-end extracts the VMM-level events and constructs the

\(^1\)The target workload is the workload whose behavior will be modeled.
features. These features are passed on to the back-end where regression algorithms are employed to build the workload characterization model.

In this thesis, we provide a general framework for modeling the behavior of workloads executing in virtualized environments that can be applied to a range of domains including intrusion detection for VM security, workload scheduling, and VM health monitoring. In this framework, VMM-level events produced during the workload’s execution are captured and processed to generate features containing information about the workload behavior. These features can be further processed in the back-end to produce different outputs, depending on the application of the framework. In this
chapter, we show how this approach can be used to perform workload characterization. The input provided to the back-end consists of the features of a set of canonical workloads and a target workload which we would like to profile. As the output, the back-end produces a qualitative model of the workload behavior in terms of a mix of the canonical workload set.

The remainder of the chapter is organized as follows. In section 5.1, we review the approach taken by our back-end to learn the behavior of the target workload and characterize it in terms of the canonical workload set. In section 5.2, we describe our experimental setup and present the results of our VMM Application Modeler on several target workloads. Finally, we conclude the chapter in section 5.3.

5.1 VMM Application Modeler Back-End

At the back-end, the features from the canonical workload set and the target workload are passed to the regression algorithms in order to produce a model of the behavior of the target workload. The resulting model characterizes the target workload in terms of a linear combination of the canonical workloads. The regression algorithms applied in this work are multiple linear least-squares regression and the LASSO algorithm. The details of these algorithms are provided in Chapter 2. The linear regression and LASSO results can be produced for each time window of the target workload, as well as an average-case window. We present both these results in section 5.2.4.

Next, we describe our experimental setup and workload characterization results.

---

2Since the front-end of the VMM Application Modeler is the same as that of the VMM IDS, we refer the reader to chapters 4 and 2 for a detailed description of the front-end.
3The features for the canonical workloads are created by passing through the front-end of the framework.
5.2 Evaluation

In this section, we describe the details of our experiments and present the results of applying regression to data extracted from the VMM layer, to produce workload characterization models.

5.2.1 Experimental Setup

For our virtual machine monitor, we use version 2.2 of VirtualBox [39]. All the experiments are executed on a Dell XPS710 machine equipped with an Intel Core2 processor (2 cores) running at 1.86 GHz with 4 GB of RAM. The host operating system is Fedora 9 [2]. In the virtual machine, we installed version 9.10 of the Ubuntu [5] OS as the guest operating system. For the Apache and MySQL workloads, the server is executed within the virtual machine, but the client is executed on a separate (non-virtualized) Dell machine with an Intel Pentium III processor running at 866 MHz with 256 MB of RAM and Fedora Core 6 as its operating system.

5.2.2 Canonical Workloads

As described in section 2.2 of Chapter 2, a generic workload can be comprised of the following main components: CPU activity, memory (RAM) activity, disk I/O (read and write), and network I/O (transmit and receive) [71, 97]. As such, we generated a series of workloads targeted to each of these components, which we refer to as the canonical workload set. These workloads are based on the benchmarks provided in the Isolation Benchmark Suite [96] and they stress different aspects of a computer system. The Isolation Benchmark Suite is designed to quantify the degree to which a virtualization system limits the impact of a misbehaving virtual machine on other
Table 5.1: Summary of the canonical workload set

<table>
<thead>
<tr>
<th>Canonical Workload</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>A tight loop containing integer arithmetic operations</td>
</tr>
<tr>
<td>Memory</td>
<td>A loop that constantly allocates and touches memory without freeing it</td>
</tr>
<tr>
<td>Disk Read</td>
<td>Runs 20 threads of IOzone [3], each performing 4 KB reads from a 100 MB file (iozone -i 1 -r 4 -s 100M -t 20)</td>
</tr>
<tr>
<td>Disk Write</td>
<td>Runs 20 threads of IOzone, each performing 4 KB writes to a 100 MB file (iozone -i 0 -r 4 -s 100M -t 20)</td>
</tr>
<tr>
<td>Network Receive</td>
<td>Runs 4 threads, each constantly reading 60 KB packets over UDP from external senders</td>
</tr>
<tr>
<td>Network Transmit</td>
<td>Runs 4 threads, each constantly sending 60 KB packets over UDP to external receivers</td>
</tr>
</tbody>
</table>

well-behaving virtual machines running on the same physical machine. A summary of each workload in the canonical set is provided in Table 5.1.

Once we characterized our canonical workloads, each stressing a different aspect of a computer system, we selected a set of application workloads whose behavior we would like to characterize. These workloads are described next.

### 5.2.3 Application Workloads

To create a diverse set of evaluation workloads, we use the following set of applications: MPrime [140] for Mersenne prime numbers, ApacheBench (ab) for Apache web servers, a TPC-C-based online transaction processing benchmark for MySQL databases [13], and mcf from the SPEC CPU2006 benchmark suite. For MPrime, we run one thread performing small fast Fourier transforms (FFTs), wherein the data fits in the L2 cache and the floating point unit (FPU) is heavily stress-tested. For the Apache workload, we installed version 2.2.15 of the Apache web server on the virtual
machine. The web page requests were performed from a remote client desktop and for the ab command line, we use the following options: -c 10 (for 10 simultaneous requests) -k (to perform multiple requests in a session) and -n 150000 (for 150,000 requests). For the MySQL workload, we simulate one terminal performing database queries and updates to the stock items of ten warehouses. For the input to the mcf benchmark, we use the reference input.

The behaviors of these workloads exhibit a range of the activity that can be represented by a mix of the canonical workloads. For the MPrime workload, we execute torture test number 1, which is predominantly CPU-intensive. We expect the Apache workload to show strong network activity. For the MySQL workload, the MySQL server is run on the virtual machine and the MySQL client is executed on a separate, remote machine. The two machines communicate over an Internet connection. Therefore, in addition to the expected disk read and disk write activity that is typical with processing queries that access database tables, there should also be network activity between the client and servers machines. The mcf benchmark is a pointer-intensive integer benchmark known for its dominant memory behavior [116].

We use regression algorithms to decompose the activity of the target workloads into a linear combination of the activity of the canonical workloads. The results are presented in the following section.

5.2.4 Results

Each of the canonical and target workloads are executed for one hour and the VMM-level events are extracted. These events are then processed to generate rate and
relationship features for time windows of length 500 interrupt timer events\textsuperscript{4} (approximately 2 seconds long). For each target workload, we pass the features of the workload along with the features of all of the canonical workloads to the regression algorithms to produce a model of the workload behavior. The linear least-squares regression results are presented in figures 5.2 through 5.5.\textsuperscript{5}

The $x$-axis corresponds to the window index and the $y$-axis represents the regression coefficients. For each time window, the linear least-squares regression coefficients are produced for each of the canonical workloads. We also use an optional constant parameter, although from the figures we see that its value is close to zero. A large non-zero value assigned to the constant parameter would indicate that the workloads

\textsuperscript{4}We also experimented with other time quantums and found the results somewhat insensitive to this parameter.

\textsuperscript{5}For the purpose of clarity, we zoomed into a portion of the hour-long run, though the pattern seen in the figures are fairly consistent throughout the execution.
Figure 5.3: Linear least-squares regression results for Apache

Figure 5.4: Linear least-squares regression results for MySQL
To summarize the results, we also find the regression coefficients for the average window\(^6\) of each target workload. These results are presented in Table 5.2. For the MPrime workload, the CPU-intensive workload is assigned the largest coefficient, confirming the strong CPU activity of MPrime. These results are very encouraging because although we only use data available at the VMM layer, we are able to show that the Mprime workload, which is a CPU torture test, exhibits behavior that is consistent with that of the CPU-intensive workload. This is a testament to the richness of the information available at the VMM layer, as long as you have the tools to extract that information. There is a large semantic gap between the VMM-level

\(^6\)The average window is found by taking the average of all the windows during the hour-long execution.
Table 5.2: Linear regression coefficients for an average window of the target workloads

<table>
<thead>
<tr>
<th>Canonical Workloads</th>
<th>MPrime</th>
<th>Apache</th>
<th>MySQL</th>
<th>mcf</th>
<th>mcf using h2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>0.00</td>
<td>0.15</td>
<td>0.07</td>
<td>-0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>CPU</td>
<td>0.93</td>
<td>-1.17</td>
<td>-0.07</td>
<td>0.94</td>
<td>0.48</td>
</tr>
<tr>
<td>Memory</td>
<td>0.00</td>
<td>0.53</td>
<td>0.14</td>
<td>0.10</td>
<td>0.63</td>
</tr>
<tr>
<td>Disk Read</td>
<td>-0.08</td>
<td>-0.14</td>
<td>0.15</td>
<td>-0.04</td>
<td>-0.02</td>
</tr>
<tr>
<td>Disk Write</td>
<td>0.14</td>
<td>0.71</td>
<td>0.34</td>
<td>0.00</td>
<td>0.08</td>
</tr>
<tr>
<td>Network Receive</td>
<td>-0.03</td>
<td>-0.11</td>
<td>0.00</td>
<td>0.05</td>
<td>0.06</td>
</tr>
<tr>
<td>Network Transmit</td>
<td>0.06</td>
<td>0.73</td>
<td>0.32</td>
<td>0.03</td>
<td>-0.01</td>
</tr>
</tbody>
</table>

data and the high-level program behavior, but with the right tools, the data can be processed to provide insight into the application behavior.

For Apache, the results show that the network transmit, disk write, and memory-intensive canonical workloads are assigned large positive coefficients and the CPU-intensive workload is assigned a large negative coefficient. This indicates that the Apache workload exhibits activity similar to that of the network transmit, disk write, memory, and CPU-intensive workloads. Based on an Apache web server’s main task of providing web pages to a client machine over a network connection, we expect to see network transmit activity. Using the Linux I/O performance monitoring utility `iostat`, we were able to confirm that the Apache workload also exhibits disk write activity. The Linux `top` utility shows that due to simultaneous web page requests, Apache displays stronger CPU and memory activity than that of a single web page request.

For the MySQL workload, we see an interesting alternating behavior where there is strong network transmit activity in addition to disk activity and then there are short phases where the network transmit activity is low and the disk activity is predominant. Further inspection of the data reveals that during the short phases
where the network transmit-intensive canonical workload is assigned a low regression coefficient, the number of network transmit events are indeed low and during phases where it is assigned a large coefficient, the number of network transmit events are more than an order of magnitude larger.\(^7\) It seems that during these short phases, the workload is more focused on performing database queries and updates, and during the longer phases, the results of the queries are transferred back to the MySQL client machine. This shows that our VMM-based workload characterization approach is able to capture these phase changes. Overall, for the MySQL workload we see strong correlation to the disk write, network transmit, disk read, and memory-intensive workloads. This validates our expectations of seeing disk activity due to database table queries and updates, as well as network activity due to the MySQL server (situated within the VM) sending the results of queries back to the client machine over a network connection.

The SPEC CPU2006 mcf benchmark exhibits strong CPU activity with several intervals of activity which resemble the activity of our memory-intensive canonical workload. Since we expect to see a more prevalent memory-intensive behavior, we decided to take a further look at the raw VMM-level data produced for the mcf benchmark and our memory-intensive canonical workload. This analysis revealed that the memory-intensive workload produced many page faults and disk writes, whereas mcf does not exhibit this kind of behavior. This is due to the fact that the memory-intensive canonical workload basically allocates memory in a loop and once the memory is exhausted, pages are swapped back to disk. Mcf shows a different type of memory-intensive behavior, wherein the memory blocks it accesses fit into main memory, resulting in few page faults and thus does not need transfer data to and

\(^7\)The number of network transmit events go from less than 50 events per second to about 1500 events per second.
To validate our hypothesis that mcf exhibits a different type of memory-intensive behavior than our current memory-intensive canonical workload, we decided to replace the canonical workload with another memory-intensive workload that does not allocate so much memory that it requires constant swapping back to disk. We selected the h2 benchmark in the DaCapo benchmark suite [27] to produce memory-intensive activity, yet without excessive page fault activity. It is a JDBCbench-like [4] in-memory benchmark, executing a number of transactions against a model of a banking application. Figure 5.6 and the last column of Table 5.2 show the linear regression results produced by replacing the memory-intensive canonical workload with the h2 benchmark. From this figure, we see that a large positive coefficient is assigned to the h2 benchmark. This confirms the similarity of the memory activity between mcf and
Using the Linux top utility, we have seen mcf exhibit strong CPU activity as well, and this is validated by the large regression coefficient assigned to the CPU-intensive canonical workload.

In addition to multiple linear regression, we also applied the LASSO algorithm to the average window of our target workloads and found the different coefficients assigned to the canonical workloads, using various values of $\lambda$. These results were calculated using the GLMNet [56] Matlab source files and are presented in figures 5.7 through 5.10. As the value of $\lambda$ increases, the coefficient of less relevant workloads are forced to zero while more relevant workloads are assigned non-zero coefficients.

The LASSO results for MPrime also confirm the strong CPU-activity that was
Figure 5.8: LASSO regression results for Apache

Figure 5.9: LASSO regression results for MySQL
shown in the multiple linear regression results. For the Apache workload, the canonical workload that retains a non-zero coefficient the longest is the network transmit-intensive workload. Since the Apache ab benchmark services web page requests, transmitting them across a network connection, we expect its behavior to display the most similarity with the network transmit canonical workload.

For MySQL, the most prevalent canonical workloads are the disk read, network transmit, disk write, and memory-intensive workloads. Again, this makes logical sense since MySQL server’s main tasks include updating database tables and responding to queries over a network connection. The LASSO results confirm our expectation of MySQL's behavior. For mcf, we produce the LASSO results with the h2 benchmark playing the role of the memory-intensive workload. As a result, h2 is the most prevalent workload, followed by the CPU-intensive workload. This also validates our
hypothesis that mcf exhibits memory-intensive behavior similar to that of the h2 benchmark.

5.3 VMM Application Modeler Conclusions

In this work, we present a general framework for modeling the behavior of virtual execution environment workloads. As a specific application of this framework, we implemented and evaluated VMM Application Modeler, a VMM-based workload characterization tool. The open source edition of VirtualBox was used to construct the front-end. We presented the types of information we were able to extract from the VMM and described the procedure used to build features. Regression algorithms were utilized as a powerful technique to bridge the semantic gap between the low-level architectural data and actual program behavior. Our results showed that there is enough information embedded within the VMM-level data to be processed and produce a model of workload behavior in terms of CPU, memory, disk, and network activity.
Chapter 6

Feature Selection for Outlier Detection

Effective outlier detection requires the data to be described by a set of features that captures the behavior of normal data while emphasizing those characteristics of outliers which make them different than normal data. In this work, we present a novel non-parametric evaluation criterion for filter-based feature selection which caters to outlier detection problems. The proposed method seeks the subset of features that represents the inherent characteristics of the normal dataset while forcing outliers to stand out, making them more easily distinguished by outlier detection algorithms.

Experimental results on real datasets show the advantage of our feature selection algorithm compared to popular and state-of-the-art methods. We also show that the proposed algorithm is able to overcome the small sample space problem and perform well on highly imbalanced datasets. Furthermore, due to the highly parallelizable nature of the feature selection, we implement the algorithm on a graphics processing unit (GPU) to gain significant speedup over the serial version. The benefits of the
GPU implementation are two-fold, as its performance scales very well in terms of the number of features, as well as the number of data points.

### 6.1 Introduction to Feature Reduction

An integral part of any data mining task is having a good set of features that can be used to accurately model the inherent characteristics of the data. In practice, the best set of features is not known in advance. Therefore, a pool of candidate features are collected and processed to removed irrelevant and redundant features. This can improve both the memory and computational cost of the data mining algorithm, as well as the accuracy of the learner. Reducing the space of possible features is done in two ways: feature transformation and feature (subset) selection. In the former, the original space of features is transformed into a new, typically lower-dimensional, feature space. Examples include Principal Components Analysis (PCA), Kernel PCA [121], and Kernel Dimensionality Reduction [57].

In the latter approach to reducing the size of the feature space, the original set of features remain unchanged and a subset of those features are selected. A simple way to perform feature selection is to use a feature evaluation function, such as relevance [79], to rank the features on an *individual* basis. Then, a subset of the features are selected by taking the top-ranked features (for example, the top $m$ features). Another feature selection methodology is to search the space of feature subsets and select the subset that optimizes a criterion function. For a dataset with $d$ features, there are $2^d$ possible subsets of features. For even a moderate value of $d$, an exhaustive search would be too computationally expensive, so it is common to use a greedy search strategy such as sequential forward selection or backward elimination [91].
In addition to the search strategy, the other important component of feature selection is the criterion to be optimized. A brute force way to evaluate features is to utilize the classifier that will ultimately be used. Such a method is called a wrapper approach [82]. Another, less computationally expensive, way is to evaluate features based on some criterion function. This is referred to as a filter method. Existing criteria include measures based on distance, information, dependency, and consistency [40]. A third approach is to perform feature selection as part of the learning task, known as an embedded method, such as a decision tree.

In this work, we present a novel optimization criterion inspired by outlier detection problems where the data is highly imbalanced and outliers comprise a small portion of the dataset. The criterion tries to find the set of features that maximizes the density\(^1\) of normal data points while minimizing the density of outliers. In other words, it seeks feature subsets wherein normal data points fall in high-density regions and outliers fall in low-density regions of the feature space. The goal is to make outliers stand out more prominently from normal data points, which allows outlier detection algorithms to more easily identify them.

Most of the work on dimensionality reduction for outlier detection have tackled the problem using a feature transformation approach. In these methods, a projection of the feature space is sought which can be used to detect outliers. While this approach subsumes feature selection (i.e., projecting onto the original feature axes is equivalent to selecting a subset of the features), there is a case to be made for the understandability that comes with feature subset selection as opposed to linear or non-linear combinations of the features. Retaining a subset of the original features goes a long way towards understanding the nature of the underlying data and the

\(^1\)Here, the term density refers to how dense and closely situated the data points are in the feature space, not to be confused with the probability density function.
features that contribute to an outlier’s deviant behavior. This can be advantageous in domains such as fraud detection and intrusion detection, in which case anomalous activity can be narrowed down to a subset of the collected features.

In addition to proposing a feature selection algorithm that seeks to intrinsically enhance the quality of outlier detection, an important contribution of this work is the implementation of the algorithm on a graphics processing unit (GPU) and the substantial speedup that is acquired. GPUs are massively parallel floating point processors attached to dedicated high speed memory, at a fraction of the cost of traditional parallel processing computers. They are designed for compute-intensive, highly data-parallel computation and rely on multithreading to provide throughput-oriented performance. Therefore GPUs are well suited for data-parallel applications, resulting in large improvements in running time.

The remainder of the chapter is organized as follows. In section 6.2, we provide the details of our proposed local kernel density ratio feature selection algorithm. In section 6.4, we describe the datasets, outlier detection algorithms, and feature selection algorithms with which we compare our method. We also present the results of our feature selection algorithm on several real-word datasets and include a comprehensive analysis of the performance gained by a GPU-based implementation. Finally, section 6.5 concludes the chapter.

6.2 Local Kernel Density Ratio Feature Selection

The inspiration for our feature selection algorithm came from an approach taken to solve the outlier (or anomaly) detection problem. Hawkins [66] describes an outlier
as “an observation that deviates so much from other observations as to arouse suspi-
cion that it was generated by a different mechanism.” In outlier detection problems, outliers typically comprise a small portion of the dataset. Examples include intrusion detection data, where malicious attacks are less frequent than normal activity, and certain tumor datasets where patients with a particular type of cancer are few com-
pared to the number of healthy patients. In some cases, obtaining samples from the outlier class may be difficult or expensive, such as a fault detection system where it can be expensive to obtain outlier data representing the faulty situations of a machine. Therefore in the outlier detection domain, learners must deal with highly imbalanced data. Next, we describe our criterion function for feature selection which caters to outlier detection problems and is insensitive to the degree of imbalance in the data as it is based on a ratio of average normal to outlier densities.

6.2.1 Local Kernel Density Ratio Criterion

In this work, we propose a novel feature selection criterion for outlier detection which tries to find the set of features that best describes the normal class while ensuring that outliers “stand out”. While most outlier detection techniques take an unsupervised approach, it is not uncommon to have samples that belong to the outlier class. For example, in intrusion detection there may be many instances of malware attacks and malicious executions that can provide guidance as to how normal executions differ from malicious ones. In our case, we utilize the supervised information to select features which are intrinsically suitable for outlier detection. Normal data points come from the same distribution while outliers can be any point, from a completely different distribution. Taking advantage of information from both normal and outlier points (if available) is more powerful than normal alone. The aim is to have something
to compare the normal distribution against and find out what separates outliers from
the normal data. Let $X = [x_1, x_2, ..., x_n]$ represent a dataset with $n$ data points
where each data point $x \in \mathbb{R}^d$ has $d$ features and is associated with a class label
$y \in \{-1, +1\}$, where $-1$ denotes the normal class and $+1$ denotes an outlier.

The Local Outlier Factor (LOF) algorithm [28] solves the outlier detection problem
using a ratio of densities. In the LOF algorithm, the density of a data point is
compared to that of its neighbors and based on this, the point is assigned a degree
of being an outlier, known as its local outlier factor. The LOF of a data point is
calculated as the average density of data points within its neighborhood divided by
its own density. When a data point has a low density compared to points in its
neighborhood, it is more likely to be an outlier. Conversely, outliers should have a
lower density compared to its neighbors. Therefore, it stands to reason that a feature
set which emphasizes this phenomenon would facilitate the detection of outliers.

To test this hypothesis, we developed a criterion that measures the quality of
features based on the density induced for normal and outlier data points. More
specifically, to maximize the density of normal data points while minimizing the
density of outliers, the criterion function takes the ratio of the two, with a focus on
the local neighborhood density of each data point. To measure density, we make no
assumptions about the form of the underlying distribution of the data. Instead, we
take a non-parametric approach and calculate the kernel density estimate of the data
points with a Gaussian kernel. The objective is to find the optimal set of features $w^*$
that maximizes the described criterion function $J(w)$:

$$w^* = \arg \max_{w \in \{0,1\}^d} J(w)$$

(6.1)
We formally define the criterion function as:

\[ J(w) = \frac{1}{|X_-|} \sum_{x_+ \in X_-} \frac{1}{|N_k(x_-)|} \sum_{x \in N_k(x_-)} K(w \circ x, w \circ x) \]

(6.2)

In the above equation, \( K \) is a kernel function. In our experiments, we use the Gaussian (or Radial Basis Function) kernel:

\[ K(x, x') = \exp \left( -\frac{\|x - x'\|^2}{2\sigma^2} \right) \]

(6.3)

The vector \( w = (w_1, w_2, ..., w_d) \) is a binary vector signifying which features are selected; for \((j = 1, ..., d)\), \( w_j = 1 \) denotes the presence of feature \( j \) and \( w_j = 0 \) denotes its absence. We use \( x_- \in X_- \) and \( x_+ \in X_+ \) to represent data points from the normal and outlier class, respectively. The parameter \( k \) determines the size of the local neighborhood of a data point, \( \sigma \) is the width of the Gaussian kernel, and the symbol \( \circ \) represents the Hadamard product [69]. The Hadamard (or Schur) product of two matrices is their element-wise product.

The size of the local neighborhood of a data point \( x \) is determined by the distance to its \( k \)-th nearest neighbor, referred to as its \( k \)-distance. All of the data points whose distance to \( x \) is less than this distance comprise its \( k \)-distance neighborhood, \( N_k(x) \). The \( k \)-distance neighborhood of a data point \( x \) is formally defined as follows:

\[ N_k(x) = \{ x' \in X \setminus x \mid d(x, x') \leq k\text{-distance}(x) \} \]

(6.4)

where \( d(x, x') \) is the distance between the two points.

The number of data points in the \( k \)-distance neighborhood of a point may exceed \( k \), due to possible ties in distance. Therefore in the criterion function, once we sum up the contributions to the density by the \( k \)-distance neighbors, we divide it by the number of points in the \( k \)-distance neighborhood, \(|N_k(x)|\). The local neighborhood
density of a data point \( x \) can be thought of as a measure of the similarity of points within that neighborhood to \( x \). Since a kernel function can be used as a measure of the similarity between two data points [24], in Equation 6.2 other kernel functions can be used in place of the Gaussian kernel. We use the Gaussian kernel and effectively perform kernel density estimation (KDE), also referred to as the Parzen-Rosenblatt Window method [113, 119]. This provides a standard, non-parametric notion of density for the data points.\(^2\)

Our criterion function tries to optimize the ratio of local kernel density estimates for normal and abnormal points (outliers). In the numerator, we sum the local kernel density of all normal data points and in the denominator, we sum the local kernel density of all outliers. By maximizing the ratio of the two, the goal is to find the subset of features that maximizes the density of normal data points and simultaneously minimizes the density of outliers. Intuitively, we would like to find a lower dimensional subspace that corresponds to a subset of the features wherein normal data points are in closely compacted regions of the space while outliers are dispersed, allowing them to be more easily distinguished as anomalous with respect to the normal data. By using a local density approach, our criterion can aid outlier detection algorithms in detecting local, as well as global, outliers [28]. In particular, we are already thinking in terms of local neighborhoods, as reflected in the KDE calculations. This notion can be carried over to the outlier detection phase, especially in the case of a local density-based outlier detection algorithm such as LOF which calculates the density of each point within a local neighborhood. This allows the detection of data points that seem to be outlying when considered within the scope of its local neighborhood, not just on a global scale.

\(^2\)We use a single density estimator for the entire data, rather than having one per class.
To illustrate the advantage of using our proposed criterion function, we show a simple two-dimensional example in Figure 6.1 and 6.2. Assume we have a dataset with two features and we would like to select the single best feature for outlier detection. In Figure 6.1, we use $\times$ to represent normal points, $+$ for outliers, and we show the projections of the data points onto each of the feature axes. In Figure 6.2, for each data point we use a rectangular bar to indicate the magnitude of its density. By projecting onto Feature 1, the two classes will have high separability, yet the outliers will have high density which makes it difficult for an outlier detection algorithm to identify them. Projection along Feature 2 gives the normal points high density and
the outliers low density, facilitating their detection. A method that tries to best separate the classes, such as Linear Discriminant Analysis (LDA) which maximizes the trace of the between-class to within-class scatter ratio, would fail to select Feature 2 over Feature 1. Our criterion will correctly select Feature 2 as it maximizes the density ratio of normal points to outliers.

### 6.2.2 Forward Search Strategy

Using the described criterion function to evaluate features, the next component of our feature selection algorithm is its search strategy. There are many approaches
to searching the space of possible feature subsets, from the naïve exhaustive search
to more sophisticated search strategies such as genetic algorithms. In this work,
we apply sequential forward selection (SFS), also referred to as sequential forward
search [43]. This is a greedy search technique that begins with an initially empty
set and adds features one at a time such that the feature added at each round is
the one that best improves the criterion. Note that one can utilize other search
strategies, such as backward selection, sequential forward floating search, or applying
sparse optimization [131, 95]. For the purposes of this work, whose goal is to test
the appropriateness of the newly introduced criterion for outlier detection, we find
it sufficient to utilize a simple search strategy that takes feature interaction into
account, such as sequential forward search. In section 6.4, we show that this search
strategy when combined with our novel criterion function is able to select features
that can facilitate the detection of outliers in real-world datasets. In addition, our
feature selection algorithm is a filter method, which is less computationally expensive
than wrapper or embedded methods for feature selection [64]. We name our proposed
feature selection algorithm for outlier detection *Local Kernel Density Ratio* (LoKDR)
feature selection.

LOF has been shown to perform well in detecting outliers using a ratio of densities.
However, it uses a *heuristic* notion of density; the density of a data point is the inverse
of the average *reachability* distance between the point and its neighbors, which is the
maximum of the actual distance between two points and the *k*-distance of the latter
point [28]. We utilize the success of the main notion in LOF with the goal of producing
a cleaner, simpler model that requires no heuristics and is based on KDE, which has
a solid statistical foundation. By maximizing the ratio of densities, we emphasize
the differences between outliers and normal points, enhancing the ability of outlier
detection algorithms to correctly identify outliers.

Sugiyama et al. [128] incorporate dimensionality reduction into their density ratio estimation procedure for outlier detection. Their idea is to identify a subspace in which the two densities are significantly different and perform density ratio estimation only in this subspace. A limitation of their approach is that they make a key assumption about the conditional densities of the two data distributions, which forms the basis of their proposed algorithm. Our novel feature selection algorithm makes no assumptions about the data distribution and thus is applicable to a wider range of datasets. Furthermore, they approximate the ratio of densities whereas we measure the densities directly, incorporating the notion of local neighborhood density which, in practice, reduces the computational cost of the kernel density estimation. This is due to the fact that only $k$ calculations of the Gaussian kernel (which requires taking the exponent of the squared distances between pairs of points) need to be done, rather than $n$ calculations, and for most applications: $k << n$.

### 6.2.3 Analysis of Computational Complexity

One of the main components of calculating the criterion function is the $k$-nearest neighbor ($k$-NN) search. A simple brute force approach is to calculate the distance between all pairs of points, requiring $\frac{n(n-1)}{2} \times O(d)$ calculations, and then sort the distances to find the $k$-nearest neighbors of each point using $n \times O(n \log n)$ comparisons, for a total computational complexity of $O(n^2(d + \log n))$. Other $k$-NN algorithms have been proposed to reduce the computation time, where the main idea is to reduce the number of distances computed [92]. For example, some algorithms partition the data points using an indexing structure, such as a $kd$-tree, and only compute distances within nearby volumes [12]. This method has been shown to be faster than the brute
force approach by up to a factor of 10 [58].

Once the $k$-NNs of a point is found, the $k$-nearest neighbor distances are used to calculate its kernel density estimate. These KDE values are then summed up and averaged for the normal points and outliers ($O(n)$), after which the ratio is taken ($O(1)$). This produces the criterion value for one set of features. Assuming feature set $F_i$ has $d_i \leq d$ features, the computational complexity of calculating the criterion function for $F_i$ is: $O(n^2(d_i + \log n)) + O(n) + O(1) = O(n^2(d_i + \log n))$. Therefore, the run-time of the algorithm is bounded by the time it takes to perform the $k$-nearest neighbor search.

Since there are no dependencies between individual $k$-NN queries, the $k$-nearest neighbor search is highly task-parallel, as all queries can be performed in a simultaneous manner, each one independent of any other query. The process is also known to be data-parallel, meaning the data values required to calculate pairwise distances are used for several different calculations. This makes it an excellent candidate for implementation on a GPU. General purpose computing on a graphics processing unit (GPGPU) has become a popular, cost effective approach for high performance parallel computing. Garcia et al. [58] have shown that implementing the $k$-nearest neighbor search on a GPU accelerates the search by up to a factor of 400 compared to the brute force CPU-based implementation.

We build upon this achieved speedup for the $k$-NN search and implement our entire feature selection algorithm on a GPU. In addition to the parallelism that exists during the $k$-NN search, we also take advantage of the parallel nature of the forward feature search technique, i.e., each of the candidate feature subsets can be evaluated independently. Thus during each step of the feature search, it possible to calculate the criterion function for all of the possible feature subsets (which is at most $d$)
concurrently on a GPU. This, combined with the parallelism of the $k$-NN search, enables the algorithm’s performance to scale efficiently in terms of both the number of features and the number of data points. It is interesting to note that in an ideal situation where all of the $k$-NN queries and feature subset evaluations are performed simultaneously, round $i$ of the algorithm would only require $O(d_i + n)$ time, with $O(d_i)$ time for a distance calculation consisting of $d_i$ features and $O(n)$ time for the KDE summation. Since there are at most $d$ rounds of the algorithm (corresponding to the addition of every feature), the best-case computational complexity of a parallel implementation of the LoKDR algorithm is $O(d^2 + nd)$. In practice, the forward search is cut off after a certain constant number of features $c << d$ are selected, yielding a ideal run-time of $O(c(d + n)) = O(d + n)$, making the algorithm linear in the size of the features and sample space.

6.3 GPU-Based CUDA Implementation

As previously mentioned, there are two main opportunities for parallelism in the LoKDR feature selection algorithm:

1. $K$-nearest neighbor queries

2. Feature subset evaluation

In the search for the nearest neighbors of data points, there is task parallelism as distance calculations between pairs of points can be performed independently. In addition, sorting to find the $k$-nearest neighbors can be done concurrently for different data points. There is also data parallelism in calculating the distances between pairs of points and shared memory can be used to store portions of data point feature vectors, to be reused in the distance calculations.
The second aspect which makes the LoKDR algorithm suitable for implementation on a GPU is the task parallel nature of evaluating the candidate feature subsets. Each subset of features can be evaluated independent of all other subsets, and hence this can be performed concurrently on a GPU.

Figure 6.3 illustrates the data structures we use to exploit the parallelism of the LoKDR feature selection algorithm. In Figure 6.3(a), we show how to take advantage of the data parallelism. The goal is to find the (squared) distance between every
point in matrix $A$ to every point in matrix $B$. The LoKDR algorithm requires finding the distance between every pair of points in the dataset, so the two matrices $A$ and $B$ are identical and equivalent to the entire dataset. To optimize memory allocation on the device (GPU), we allocate memory using `cudaMemcpy2D` for two-dimensional matrices, whose width is assigned in the `pitch` parameter. The dimension of a thread block (number of threads in a particular coordinate) is specified in the `BLOCK_DIM` parameter. We experimented with several block sizes and found that a block size of $16 \times 16$ for 2-dimensional processing and $256 \times 1$ for 1-dimensional processing resulted in the best GPU performance. An example of 2-dimensional processing is the pairwise distance calculations between points in the data matrix (c.f. Figure 6.3). Examples of 1-dimensional processing include the concurrent sorting of the $k$-nearest neighbor distances (each thread is responsible for sorting one column of the distance matrix) and the concurrent calculation of the KDE values (each thread uses the sorted $k$-NN distances to calculate the KDE of a data point in parallel).

The dataset consists of `data_count` number of points and `feature_count` number of features. The value at position $A_{ij}$ (and hence $B_{ij}$) is the value of the $i$th feature of the $j$th data point. For a particular feature set $F_m$, the distance between data points $A_p$ and $B_q$ is found by taking the sum over all elements $f \in F_m$ of $(A_{fp} - B_{fq})^2$ and then taking the square root.\(^3\) When calculating the distance between a particular data point and all other points, there is a redundancy that occurs when accessing the feature vector of that point in global memory. To reduce the number of global memory accesses, we can load a portion of the feature vector into shared memory to be reused by several computations. In the kernel function, this is performed by all the threads that belong to the same block. In the figure, each blue square represents

\(^3\)In LoKDR, the distances are used to calculate the KDE values, which require the squared distances, so we can omit taking the square root and leave the distances in their squared form.
the elements that will be processed by a block of threads.

Each thread is responsible for calculating the distance between one point in $A$ to one point in $B$. To this end, it first loads one element of matrix $A$ and one element of matrix $B$ into shared memory. The thread identified by indices $(tx, ty)$ loads element $A_{xIdx \ yIdx}$ into shared memory $A$ and element $B_{xIdx \ yIdx}$ into shared memory $B$, where

$$
\begin{align*}
    xIdx &= blockIdx.x \times blockDim.x + tx \\
    yIdx &= blockIdx.y \times blockDim.y + ty
\end{align*}
$$

Next, synchronization among the threads in the same block is performed to ensure that the elements in shared memory are ready to be used by all threads. Then, a for loop with index $k$ (see Figure 6.3(a)) is used to sum over the squared differences between feature values $A_{kty}$ and $B_{ktx}$ over all features associated with that block. This process is then repeated for the next thread block underneath it (the lighter blue square in the figure) and continues to the very last thread block at the bottom (resulting in the sum over all pertinent\footnote{Note that before we add the contribution of a feature to the distance calculation, we check to see if the feature belongs to the particular feature set we are evaluating.} feature values). In addition to the data parallelism that exists in calculating the pairwise distances, there is also task parallelism as the distances are computed concurrently and independent of each other.

Figure 6.3(b) reflects more opportunities to perform independent tasks in parallel on a GPU. In this figure, we show the Distance matrix which contains the pairwise distances between every point its columns to every point in its rows. Every row corresponds to one data point in the dataset. More interesting is the columns of the matrix; not only do they correspond to every point in the dataset, but they also account for every possible feature subset that we would like to evaluate during one round of the feature space search. In other words, each column of the distance matrix
contains the distances between a particular point in the dataset to all other points, with respect to a particular subset of features. The task parallelism ensues from the ability to process the elements in each column independently. In particular, we can sort each column concurrently to find a data point’s \( k \)-nearest neighbor distances. In addition, we can then use those distances to calculate the point’s kernel density estimate – all performed in parallel. Once complete, we can calculate the criterion function for every feature subset, resulting in a forward\(^5\) step in the feature search to the subset of features which produces the best criterion value.

### 6.4 Experimental Evaluation

In this section, we describe the datasets and outlier detection algorithms used in this study to evaluate the quality of features selected by the LoKDR algorithm. We also briefly describe other feature selection techniques with which we compare our results. We then present the outlier detection results for each of the feature selection methods.

#### 6.4.1 Datasets

The datasets used to evaluate our feature selection algorithm are shown in Table 6.1. CNS and LYMPH are microarray gene expression datasets and OVARY and PROST are mass spectrometry datasets provided by Chen et al. [35]. These datasets are examples of real-world problems that consist of a small set of samples and imbalanced class labels. We also evaluate our feature selection algorithm on the ARRHY dataset\(^6\) [63] from the UCI Machine Learning Repository [54], a dataset which has

---

\(^5\)This can easily be extended to backward search, where a backward step is taken and a feature is removed.

\(^6\)Preprocessing was done to account for missing values, resulting in the removal of three features and two instances.
### Table 6.1: Overview of datasets

<table>
<thead>
<tr>
<th>Name</th>
<th>Features</th>
<th>Samples</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNS</td>
<td>7129</td>
<td>90</td>
<td>Central Nervous System Embryonal Tumor Data: 60 samples have medulloblastomas and 30 samples have other tumors or no cancer.</td>
</tr>
<tr>
<td>LYMPH</td>
<td>7129</td>
<td>77</td>
<td>Lymphoma Data: 58 samples are diffuse large B-cell lymphomas and 19 samples are follicular lymphomas.</td>
</tr>
<tr>
<td>OVARY</td>
<td>6000</td>
<td>66</td>
<td>Ovarian Cancer Data: 50 samples are benign tumors and 16 samples are malignant tumors.</td>
</tr>
<tr>
<td>PROST</td>
<td>6000</td>
<td>89</td>
<td>Prostate Cancer Data: 63 samples have no evidence of cancer and 26 samples have prostate cancer.</td>
</tr>
<tr>
<td>ARRHY</td>
<td>276</td>
<td>450</td>
<td>Cardiac Arrhythmia Data: 244 samples are from class 01 and 206 samples are from classes 02-16.</td>
</tr>
</tbody>
</table>

neither highly imbalanced data nor a small sample space. The goal is to distinguish between the presence and absence of cardiac arrhythmia, where class 1 is the normal class and classes 2 to 16 are outliers. Table 6.1 provides the number of features, number of samples, and a summary description on each of these datasets.

### 6.4.2 Outlier Detection Algorithms

For outlier detection, we use one-class classifiers which are trained on only normal data (inliers). For each data point, the classifier produces a decision value that represents its confidence in that point being an outlier. We apply a threshold on the decision value as a cutoff point for decision making. A data point is flagged as an outlier if the decision value exceeds a threshold. Varying the threshold varies the number of correctly classified outliers (true positives) and incorrectly classified normal data (false positives). Using this information, we plot a curve of the true positive rate versus the false positive rate, known as the Receiver Operating Characteristic (ROC) curve [51].
In section 6.4.4, we perform an evaluation of several feature selection techniques in terms of the area under the ROC curve (AUC) achieved by the outlier detection algorithms on different feature subsets chosen by the feature selection techniques.

The classifiers used to evaluate the feature subsets are Nearest Neighbor (NN), Local Outlier Factor (LOF), and One-Class Support Vector Machines (OCSVM). The (one-class) Nearest Neighbor classifier is a distance-based outlier detection algorithm wherein a data point’s decision value is the distance to its nearest neighbor. The greater the distance, the more likely that point is an outlier. The LOF algorithm takes a density-based approach to detect outliers; the greater the density of a point’s nearest neighbors compared to its own density, the more outlying the data point. The decision value of a data point is its local outlier factor.

The OCSVM classifier [94] uses a kernel function to map the data into a feature space $H$ with the goal of capturing most of the data vectors within a “small” region. It then tries to separate the mapped vectors from the origin with a hyperplane that has the maximum margin. The origin and data points “close enough” to it are assumed to be outliers. The decision value of a data point is $f(x) = \sum_{i=1}^{n} \alpha_i K(x_i, x) - \rho$ where $\alpha_i$ are the support vector coefficients, $K$ is a kernel function, and $\rho$ represents the margin. The parameter $\rho$ is effectively the threshold that determines whether points are “close enough” to the origin to be considered outliers. For our OCSVM experiments, we use LIBSVM (version 3.11) [33] with the standard parameters for one-class SVMs.

### 6.4.3 Feature Selection Algorithms

To compare our results with other popular filter-based feature selection algorithms, we evaluate the features selected by RELevance In Estimating Features (RELIEF),
Feature Assessment by Sliding Thresholds (FAST), Least Absolute Shrinkage and Selection Operator (LASSO), and BAckward elimination with the Hilbert Schmidt Independence Criterion (BAHSIC). RELIEF [79] is a filter feature selection method that evaluates individual features based on how well they differentiate between neighboring instances from different classes versus from the same class. We use the Weka toolbox [65] to select features with RELIEF. FAST [35] is a feature selection algorithm for small sample and imbalanced data classification problems. The main idea is to rank features based on the area under the ROC curve generated by each feature.

LASSO [131] solves the linear regression problem formulation with an added constraint on the sum of the regression coefficients. This drives the coefficients of less relevant features towards zero. We use the logistic regression variant provided in the GLMNet [56] Matlab toolbox and rank features based on the absolute value of the coefficients. BAHSIC [126] uses the backward elimination search strategy on features evaluated using the Hilbert Schmidt Independence Criterion (HSIC). We use the Python code provided by Song et al. [126] to perform feature selection with BAHSIC.

6.4.4 Results

For the training phase of the outlier detection algorithms, we take a one-class (or semi-supervised) learning approach and train only on normal data points. During testing, both normal and outlier data points are used to see how well the model is able to detect outliers. We perform 10-fold cross validation by dividing the normal data points into ten folds and training on nine of them while testing on the tenth. Since no outliers are used during the training phase, we use all outliers during the testing phases of the cross validation. In what follows, we present two sets of experiments: in the first, we

\footnote{This is also the approach taken by Hido et al. [67].}
evaluate the quality of the solution achieved by our novel feature selection algorithm compared to other state-of-the-art methods. In the second set of experiments, we quantify the speedup obtained with a parallel GPU-based implementation of the LoKDR feature selection algorithm, compared to a serial CPU-based implementation.

6.4.4.1 Quality Experiments

We evaluate the quality of the outlier detection results using the area under the ROC curve (AUC) and the balanced error rate (BER). The ROC curve is a plot of the true positive rate (fraction of outliers correctly detected) versus the false positive rate (fraction of normal points misclassified as outliers). It represents the behavior of a classifier across a range of thresholds on the decision values. The BER, on the other hand, represents the behavior of the classifier at a particular operating point and is the average of the false positive rate and the false negative rate (fraction of outliers misclassified as normal) at that threshold.

For the LoKDR feature selection algorithm, there are two main parameters that can be tuned: $k$ which determines the size of the local neighborhood and $\sigma$, the Gaussian kernel width. By varying the value of $k$ in $[1, n - 1]$ and $\sigma$ in $[1, 5]$, we observed that most of the results were not drastically sensitive to the choice of these parameters, though some values produced slightly better results than others. The benefits of this are two-fold; first, it shows the stability of the criterion function, as it is not extremely sensitive to these parameters. Second, with smaller values of $k$, we can achieve similar (if not better) results than larger values, thereby reducing the computational cost. In section A.1 of the appendix, we provide an extensive analysis of the sensitivity of the results to these parameters. For our experiments, we set the values of the parameters based on 10-fold cross validation on the training set. We
found that a value of $\sigma = 3$ provided good results across all of the datasets. For the microarray and mass spectrometry datasets, we set $k \approx n/3$ (with $k = 30, 25, 20,$ and $30$, respectively) and for ARRHY, we set $k = 400$.

The results of our experiments using NN, LOF, and OCSVM show that in general,\(^8\) the classifiers perform comparably across the various feature selection algorithms.

\(^8\)The only exceptions to this were the poor performance of the nearest neighbor classifier on the LYMPH dataset using the BAHSIC and FAST features, and PROST with FAST features, and OCSVM on LYMPH with FAST features, which is shown in the in the appendix.
and datasets with no clear winner. As an example, in Figure 6.4 we present the average area under the curve (AUC) results as a function of the number of selected features using 10-fold cross validation for all three outlier detection algorithms on the CNS dataset with features selected by LoKDR, BAHSIC, RELIEF, and LASSO. On the x-axis, we vary the number of selected features and on the y-axis, we plot its corresponding average AUC.

As the goal is not to compare the outlier detection algorithms themselves, but rather to compare the proposed feature selection algorithm (LoKDR) with previous feature selection methods, for the remaining figures of this section we shall present the outlier detection results using the LOF classifier. The other classifiers produce similar results, as shown in the appendix. In Figure 6.5, we show the AUC results of the feature selection algorithms on the microarray and mass spectrometry datasets. With a horizontal line, we show the AUC obtained when using all of the features. This displays the importance of performing feature selection, as all of the feature selection algorithms are able to surpass the AUC achieved with the entire feature set. The figure also highlights the strength of the LoKDR algorithm in selecting features for outlier detection. Across the datasets, the features chosen by LoKDR enable the outlier detection algorithm to identify outliers with a high detection rate and few false positives, as reflected in the high average AUC. For the CNS, LYMPH, and OVARY datasets, as the number of selected features increases, the average AUC for LoKDR rapidly exceeds that of the other methods. For the PROST dataset, the performance of LoKDR starts out the strongest and continues to be competitive with the other methods.

To see how well our feature selection algorithm performs on a more general dataset that does not have imbalanced data with a small sample size, we also ran experiments
Figure 6.5: Average AUC results for microarray and mass spectrometry datasets on the ARRHY dataset and present results in Figure 6.6(a). The results show that while features selected by BAHSIC produce the highest average AUC for most of the feature subsets, the LoKDR algorithm still performs well and produces AUC values that are comparable with those of BAHSIC.

In addition to the area under the ROC curve which provides a picture of the outlier detection results across a range of thresholds on the decision values, we also provide a “snapshot” of the outlier detection results using the average minimum balanced error.
rate (BER). These results are presented for the various feature selection algorithms in Figure 6.6(b) and Figure 6.7. On the $x$-axis, we have the number of selected features and on the $y$-axis, the average minimum BER. For each dataset, the two lowest BER values are highlighted, with an asterisk at the lowest BER and a plus sign at the second lowest BER. The corresponding values are shown in the color of the feature selection algorithm that produced it. The results show that LoKDR consistently produces low values for the BER and in most cases, outperforms all of the other feature selection methods. For the CNS dataset, it is even able to achieve perfect detection (100% true positives and 0% false positives) with only 14 features. For the LYMHP and OVARY datasets, LoKDR achieves the lowest BER at a cost of a few more features than LASSO (which came in second). For the PROST and ARRHY datasets, LoKDR places a very close second to LASSO and BAHSIC, respectively.

We also vary the number of selected features up to 100 and present the lowest average BER results for the various feature selection algorithms (FS) and outlier

---

9We find the minimum BER for different thresholds and then take the average across the cross-validation runs.
Figure 6.7: Average BER results for microarray and mass spectrometry datasets detection algorithms (OD) in Table 6.2. For each FS/OD pair, we show the number of selected features next to the balanced error rate. For each dataset and outlier detection algorithm, the two top feature selection algorithms with the lowest BER are shown in boldface, with an asterisk next to the winner. These results show that LoKDR always achieves very low BER values and in every case, either outperforms all of the other methods or places a close second.

Using the paired Student’s t-test, we confirmed that our experimental results
Table 6.2: Lowest average BER results on all datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>OD</th>
<th>FS</th>
<th>NN BER</th>
<th>Features</th>
<th>LOF BER</th>
<th>Features</th>
<th>OCSVM BER</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNS</td>
<td>Od</td>
<td>LoKDR</td>
<td>0.000*</td>
<td>14</td>
<td>0.000*</td>
<td>14</td>
<td>0.000*</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BAHSC</td>
<td>0.035</td>
<td>27</td>
<td>0.035</td>
<td>13</td>
<td>0.042</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RELIEF</td>
<td>0.047</td>
<td>41</td>
<td>0.048</td>
<td>37</td>
<td>0.055</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LASSO</td>
<td>0.060</td>
<td>37</td>
<td>0.050</td>
<td>36</td>
<td>0.050</td>
<td>37</td>
</tr>
<tr>
<td>LYMPH</td>
<td>Od</td>
<td>LoKDR</td>
<td>0.008*</td>
<td>42</td>
<td>0.008*</td>
<td>41</td>
<td>0.008*</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BAHSC</td>
<td>0.183</td>
<td>9</td>
<td>0.065</td>
<td>4</td>
<td>0.082</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RELIEF</td>
<td>0.043</td>
<td>17</td>
<td>0.059</td>
<td>33</td>
<td>0.077</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LASSO</td>
<td>0.041</td>
<td>21</td>
<td>0.043</td>
<td>21</td>
<td>0.043</td>
<td>21</td>
</tr>
<tr>
<td>OVARY</td>
<td>Od</td>
<td>LoKDR</td>
<td>0.010*</td>
<td>42</td>
<td>0.019*</td>
<td>35</td>
<td>0.016*</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BAHSC</td>
<td>0.060</td>
<td>21</td>
<td>0.065</td>
<td>8</td>
<td>0.056</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RELIEF</td>
<td>0.045</td>
<td>92</td>
<td>0.064</td>
<td>38</td>
<td>0.064</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LASSO</td>
<td>0.039</td>
<td>23</td>
<td>0.048</td>
<td>20</td>
<td>0.045</td>
<td>20</td>
</tr>
<tr>
<td>PROST</td>
<td>Od</td>
<td>LoKDR</td>
<td>0.007</td>
<td>37</td>
<td>0.007</td>
<td>19</td>
<td>0.007</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BAHSC</td>
<td>0.019</td>
<td>48</td>
<td>0.017</td>
<td>45</td>
<td>0.027</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RELIEF</td>
<td>0.023</td>
<td>97</td>
<td>0.017</td>
<td>93</td>
<td>0.025</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LASSO</td>
<td>0.000*</td>
<td>18</td>
<td>0.000*</td>
<td>16</td>
<td>0.000*</td>
<td>16</td>
</tr>
<tr>
<td>ARRHY</td>
<td>Od</td>
<td>LoKDR</td>
<td>0.164*</td>
<td>34</td>
<td>0.152</td>
<td>34</td>
<td>0.151</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BAHSC</td>
<td>0.172</td>
<td>31</td>
<td>0.149*</td>
<td>24</td>
<td>0.148*</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RELIEF</td>
<td>0.186</td>
<td>17</td>
<td>0.178</td>
<td>17</td>
<td>0.179</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LASSO</td>
<td>0.263</td>
<td>33</td>
<td>0.268</td>
<td>34</td>
<td>0.269</td>
<td>34</td>
</tr>
</tbody>
</table>

showing the superiority of LoKDR over the other feature selection methods are statistically significant at the 95% confidence level with respect to all methods, except BAHSC on the PROST and ARRHY datasets, where they perform comparably. From our results, we conclude that the LoKDR feature selection algorithm chooses features that enable outlier detection algorithms to do consistently well across all the datasets, from those which are very high-dimensional with imbalanced class labels and that suffer from the small sample space problem, to a more general dataset without these properties.
6.4.4.2 Performance Experiments

By utilizing the parallelism that exists in the LoKDR algorithm, the GPU-based implementation of our proposed feature selection method offers significant performance improvements over the serial (CPU only) implementation. To quantify this gain, we perform a series of performance experiments where we pit the serial and GPU implementations head-to-head in terms of their running time on the real-world datasets. The GPU version was written using NVIDIA’s Compute Unified Device Architecture (CUDA) [110] and the serial version was written in C. Our serial experiments were run on an Intel Xeon CPU E5405 running at 2.00 GHz. The GPU used in our experiments is the NVIDIA Tesla M2070, which is based on the NVIDIA Fermi GPU architecture. Fermi GPUs are designed for general-purpose high performance GPU computing. The Tesla M2070 modules are performance-optimized, high-end products. They offer 6 GB of GDDR5 ECC-protected memories on board with 1.566 GHz memory clock and 384-bit memory interface. Fermi architectures provide floating multiply-add (FMA) instructions for both 32-bit single-precision and 64-bit double-precision floating point numbers. The features supported by a CUDA hardware are described by the Compute Capability, and Fermi architectures have the support for CUDA Compute Capability 2.0.

In Figure 6.8, we present the performance results for the microarray and mass spectrometry datasets. Figure 6.8(a) shows the running time (in seconds) of the serial implementation of LoKDR on the datasets with increasing number of selected features. At each round of the algorithm, features that have not yet been chosen are individually added to the current feature set and evaluated; the feature that best improves the criterion function is selected for inclusion in the current feature set. Of the microarray and mass spectrometry datasets, the CNS dataset has the largest
number of data points (90) and features (7129), and consequently requires a longer running time during each round of the feature search. Conversely, the OVARY dataset which has the fewest data points (66) and features (6000) has the lowest running time at each round. The more interesting result is that of the running time of LYMPH versus PROST. While the number of features in LYMPH (7129) is greater than that of PROST (6000), there is a larger number of data points in PROST (89) compared to LYMPH (77), which leads to its slightly longer running time. This validates our analysis that the number of data points has a greater impact on the computational complexity of the serial implementation than does the number of features.

Another observation that can be made from this figure is that as we add more features to each dataset, the running times show a superlinear increase. Contrast this to Figure 6.8(b), which presents the running time of the GPU-based implementation of LoKDR and where the running times of the datasets grow in a more linear, and even somewhat sublinear, manner. An important characteristic of the GPU-time plot is the drastic improvement in the running time for all of the datasets, compared to the serial version. To see this more clearly, in Figure 6.8(c), we plot the running time of both the serial and GPU implementations on the CNS dataset. The time it takes to select 600 features goes from over a day in the serial version to about two and a half minutes on the GPU. This results in a computation time savings of over 99.8%; in other words, the GPU-based implementation is over 580× faster than its serial counterpart. Figure 6.8(d) presents this improvement in running time, or speedup, for the microarray and mass spectrometry datasets with increasing number of selected features. As can be seen in the figure, the speedup grows progressively as more features are added, exceeding 800× improvement on these datasets and only going higher from there. This figure clearly illustrates the performance gain that can be
Figure 6.8: Evaluation of serial and GPU LoKDR on CNS, LYMPH, OVARY, PROST achieved from concurrent processing on a GPU. As more features are selected, the cost of performing distance calculations (and hence $k$-NN queries) increases, so there is more performance to be gained when they are done concurrently. This, coupled with the small sample space of the microarray and mass spectrometry datasets, means that more data points can be processed and used to calculate the optimization criterion within a shorter amount of time, which is reflected in the large rise in speedup for the CNS, LYMPH, OVARY, and PROST datasets.
Figure 6.9 contains the performance results on the ARRHY dataset with varying values of $k$ (size of the local neighborhood). The serial implementation results are shown in Figure 6.9(a) and the GPU-based implementation results in Figure 6.9(b). Due to the smaller feature size compared to the microarray and mass spectrometry datasets, we can present the running time as the number of selected features increases from 1 to 276 (the total number of features). As the number of selected features grows, the set of features not yet selected shrinks, resulting in fewer feature subsets to evaluate. Consequently, the closer you get to 276 selected features, the flatter the slope of the running time curve. The GPU implementation of the algorithm provides an overall smaller slope compared to the serial version which start off with an initially steeper slope. From the figure, we also see that larger values for the size of the local neighborhood result in longer running times, which is to be expected as $k$ affects the time required for sorting the $k$-nearest neighbors, as well as the calculation of the criterion function.

The speedup of the GPU implementation over the serial is shown in Figure 6.9(c). Lower values of $k$ result in the greatest gain from the GPU implementation. This can be attributed to the relatively smaller amount of time required to perform the concurrent $k$-nearest neighbor sorts. With the higher performance that comes with a smaller local neighborhood size, it would behoove us to evaluate the effect of $k$ on the outlier detection accuracy (using features selected by LoKDR). To this end, we present average AUC results with 10-fold cross validation in Figure 6.9(d). As can be seen from the figure, the size of the local neighborhood does not have a large impact on the outlier detection results and therefore, it is possible to achieve greater speedup with smaller values of $k$ without sacrificing much accuracy.
6.4.4.3 Synthetic Dataset Experiments

To evaluate the runtime of the LoKDR algorithm with respect to varying number of data points and features, we also generated and ran experiments on synthetic datasets of different sizes. In these datasets, normal points are generated from a Gaussian distribution with mean of 0 for each feature and a covariance matrix of $I$ (the identity matrix).\textsuperscript{10} Outliers, which comprise 10% of the dataset, are generated

\textsuperscript{10}This means that the features are uncorrelated and have a standard deviation of 1.
from a Gaussian distribution with mean 10 (for each feature) and covariance matrix $I$.

Figure 6.10, shows the running time results of the GPU-based implementation of LoKDR with varying number of data points and features.\textsuperscript{11} In Figure 6.10(a), we generate workloads with 300 features and varying number of data points from 100 to 1000. The running times presented are those for selecting all 300 features. From this figure we see that the running time has a slightly super-linear relationship with the number of data points. These results are very encouraging as they provide a vast improvement over the serial version which has an $O(n^2 \log n)$ relationship with the number of data points.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure610.png}
\caption{Results on synthetic workloads with varying data points and features}
\end{figure}

In Figure 6.10(b), the synthetic workloads have 100 data points and their features range from 1000 to 10000. The running times correspond to the selection of 1000 features. As the figure shows, the running time of the GPU implementation grows linearly with the number of features, with a fairly constant slope for up to 8000 features. As we saw in the computational complexity analysis, the linear relationship

\footnote{\textsuperscript{11}For the experiments on the synthetic datasets, we set $\sigma = 1$ and $k = 20$ (except for the experiment that varies $k$ to evaluate its effect on the running time).}
between the running time and the number of features is an ideal one, in terms of what can be achieved in practice. At features sizes greater than 9000, the running time increases at a higher rate. Further analysis showed that when the number of features are 8000 and fewer, the data matrix fits into the GPU’s texture memory, which is cache optimized for 2-dimensional spatial access patterns and faster to access than global memory. With feature sizes of 9000 and above, the data matrix no longer fits into texture memory and requires the use of shared memory to gain speedup. The time required to transfer data from global memory to shared memory results in an additional increase in the running time and hence, the running time is higher than when texture memory is used.

![Graph showing results on synthetic workloads with varying k](image)

**Figure 6.11:** Results on synthetic workloads with varying k

To see how the running time of the GPU implementation is affected by the size of the local neighborhood (k), we also ran experiments on a synthetic workload with 500 data points and 1000 features to evaluate this relationship. We varied the value of k from 50 to 499 (since the local neighborhood of a data point does not contain the point itself) and plotted the results in Figure 6.11. From the figure, we see that the running time has a sub-linear relationship with the size of the local neighborhood. The reason for this is because the size of the neighborhood has a relatively small
contribution to the run-time of the algorithm as a whole. In particular, the size of
the neighborhood affects the time to perform sorting of the \( k \)-nearest neighbors and
the summation over the \( k \)-nearest neighbors for the local kernel density estimation.

From all of these results, it is clearly evident that exploiting the parallelism that
exists in the LoKDR algorithm can lead to great rewards in terms of computation
time. Data mining problems where adding more features can enhance the quality of
the final solution further add to the appeal of a GPU-based implementation, with its
ability to achieve vast speedups and performance improvements. The implementation
of our proposed feature selection algorithm for outlier detection on a GPU gains from
its ability to perform processing on data points and feature subsets concurrently,
allowing the performance to scale nicely with respect to both.

### 6.5 LoKDR Feature Selection Conclusions

In this chapter, we presented a novel feature selection criterion catered to outlier
detection problems. The proposed method is non-parametric and makes no assump-
tions about the distribution of the underlying data other than the fact that outliers
are different than the normal data. It selects features that best capture the behavior
of normal data while making outliers more distinct from the normal. We applied a
forward search strategy to our feature selection criterion and compared its ability to
detect outliers with other popular feature selection methods. Experiments on real
datasets showed that our local kernel density ratio feature selection algorithm does
very well to discern features that facilitate the detection of outliers. By taking ad-
vantage of its parallel nature in terms of the number of data points and features, we
also achieved great speedups with an implementation on a graphics processing unit
(GPU).
Chapter 7

Summary

In this chapter, we summarize the contributions of this thesis and provide directions for future work.

7.1 Contributions of the Thesis

This thesis enhances the current state-of-the-art and makes key contributions to the following fields:

- **Cyber Security for Virtualization** – Sophisticated anomaly detection techniques are adapted to build an intrusion detection system for virtual execution environments. The resulting tool is the Virtual Machine Monitor Intrusion Detection System (VMM IDS).

- **Workload Characterization for Virtualization** – Powerful regression methods are extended to decompose a virtual machine workload’s behavior into primitive computing components. The resulting tool is the Virtual Machine Monitor Application Modeler.
• **Machine Learning** – A novel, effective, and efficient feature selection algorithm for outlier detection is proposed and implemented. The resulting tool is the Local Kernel Density Ratio (LoKDR) Feature Selector.

For the virtualization tools, the open source edition of VirtualBox was used to construct the front-end and extract VMM-level events. The VMM-level events were processed to produce rate and relationship features. In the back-end, outlier detection and regression algorithms were utilized to help bridge the semantic gap between the low-level architectural data and actual program behavior.

The results show that there is enough information embedded within the VMM-level data that when properly processed and mined, can be used to accurately detect an average of 94% of real-world malicious attacks on server appliances, at a cost of only 3% false alarms. In addition, the information can be used to produce a model of workload behavior in terms of CPU, memory, disk, and network activity.

We also contributed to the field of machine learning with a novel approach to selecting features that cater to outlier detection problems, which we term Local Kernel Density Ratio (LoKDR) feature selection. The idea is to seek features that maximize the density of normal points while minimizing the density of outliers. Extensive experiments on real-world data show the advantages of performing outlier detection using features chosen by LoKDR in comparison to current state-of-the-art feature selection techniques. Also, LoKDR presents several opportunities for data and task-level parallelism, making it an excellent candidate for GPU implementation. We have shown that it is possible to achieve speedups exceeding $800 \times$ when selecting up to 1000 features, with speedup further improving as more features are added. This results in a feature selection algorithm that is both fast and effective in choosing features that facilitate the detection of outliers.
7.2 Future Work

Our VMM IDS and Application Modeler offer a wealth of future research opportunities. The front-end can be extended to other VMMs such as Xen or ESX server and its performance can be significantly improved. Additional events can be extracted and used to build new features. Our back-end provides a whole new research dimension. Many data mining algorithms can be evaluated for effectiveness, as well as performance. In particular, future work can explore the use of unsupervised data mining algorithms and an incremental version of the LOF algorithm. This will provide the ability to dynamically update the model of normal behavior and build a more robust IDS with a potentially lower false alarm rate.

We currently build features based on the events that occur during the execution of various processes within the system. In future work, we can incorporate features constructed on a per-process basis. Also, in our vision for the VMM IDS, we would like to combine the results from both classifiers and produce a meta-classifier whose accuracy is higher than each of the classifiers alone. Since remediation is an essential aspect of every IDS, future research can investigate what steps need to be taken once abnormal behavior is detected, and what information should be provided to the user.

In the current work for the VMM Application Modeler, we evaluated target workloads that display a small range of the behaviors present in the canonical workload set. In future work, we can explore the behavior of more realistic workloads that exhibit diverse and time-varying behaviors. This can be extended to perform phase detection and performance optimization. Another area for future work involves studying the models produced from malicious executions and comparing them to the model of normal execution in order to detect intrusions. This can be used to enhance the VMM-based intrusion detection infrastructure.
Our novel feature selection algorithm also lends itself well to future work. For example, we can incorporate other search techniques using our novel feature selection criterion. In particular, the backward search method can reap massive rewards in terms of running time savings since it begins with all the features and removes them one by one. This necessitates a great number of computations for distance calculations and a GPU implementation can perform them concurrently, yielding significant speedups.

In the feature selection criterion, we can explore the idea of finding optimal weights for the features; specifically, we can relax the constraint that in the feature vector \( w_j \in \{0, 1\} \) and allow it to take on rational values. This will require solving an optimization problem to find the best weighting of the features that maximizes the density of normal data points while minimizing that of outliers. Furthermore, we can extend the vector \( w \) to a \( d \times q \) transformation matrix \( W \) and perform feature extraction. This would describe the optimal transformed feature space that optimizes our criterion function and can further enhance the identification of outliers.
Appendix A

Additional LoKDR Results

A.1 Sensitivity Analysis

In this section, we analyze the sensitivity of the results using our proposed criterion function to the parameters $k$ (size of the local neighborhood) and $\sigma$ (Gaussian kernel width). We present the average area under the ROC curve (AUC) outlier detection results using the LOF classifier for all the datasets.

A.1.1 Sensitivity to $k$

Figure A.1 and Figure A.2(a) present the plots of the average AUC values as a function of the selected features for varying values of $k$, the size of the local neighborhood. These results show that, except for very small values of $k$ (typically less than 20), the AUC achieved by the Local Outlier Factor (LOF) algorithm are comparable. And even in the case of small $k$ values, they tend to converge to the AUC values of the larger $k$ values for a larger number of features ($\sim 20$ for the microarray and mass spectrometry datasets and $\sim 40$ for the ARRHY dataset).
A.1.2 Sensitivity to $\sigma$

Figure A.3 and Figure A.2(b) present the plots of the average AUC values as a function of the selected features for varying values of $\sigma$, the width of the Gaussian kernel. These results show that, except for $\sigma = 1$ (and $\sigma = 5$ in the case of the LYMPH dataset), the results for the other $\sigma$ values are comparable.
Figure A.2: Average AUC on ARRHY dataset with varying $k$ and $\sigma$

A.2 Remaining AUC Results

In this section, we show the remaining average area under the curve (AUC) results using the Nearest Neighbor (NN) and One-Class Support Vector Machine (OCSVM) results in Figure A.4 through Figure A.6. Most of the results are similar to those presented in the paper using the Local Outlier Factor (LOF) classifier, except for a few. For example, on the LYMPH dataset, BAHSIC performs poorly with the NN classifier and FAST performs poorly for both NN and OCSVM.

A.3 A Note on Parameter Selection

For the BAHSIC algorithm, we present results using a linear kernel for the microarray (recommended by [126]) and mass spectrometry datasets, which was recommended by the authors in [126]. For the ARRHY dataset, we show results using the Gaussian kernel which was found to perform slightly better than the linear kernel.

For the LOF algorithm, the parameter $MinPts$ determines the size of the local
Figure A.3: Average AUC on CNS, LYMPH, OVARY, and PROST with varying $\sigma$ neighborhood. Using cross validation, we applied a wide range of $MinPts$ values in $[5, 250]$ and found that $MinPts = 20$ performed well across the microarray and mass spectrometry datasets and $MinPts = 150$ for the ARRHY dataset.
Figure A.4: Average AUC on CNS, LYMPH, OVARY, and PROST with NN classifier
Figure A.5: Average AUC on CNS, LYMPH, OVARY, and PROST with OCSVM classifier
Figure A.6: Average AUC on ARRHY dataset with NN and OCSVM classifiers
Bibliography


[127] Sourcefire. The snort network ids.


First International Workshop on Knowledge Discovery and Data Mining, pages 594–600, Washington, DC, USA, 2008. IEEE Computer Society.