Marcus A. Maloof (Ed.)

Machine Learning and Data Mining for Computer Security

Springer

Advanced Information and Knowledge Processing

Also in this series

Gregoris Mentzas, Dimitris Apostolou, Andreas Abecker and Ron Young
Knowledge Asset Management
1-85233-583-1

Michalis Vazirgiannis, Maria Halkidi and Dimitrios Gunopulos
Uncertainty Handling and Quality Assessment in Data Mining
1-85233-655-2

Asunción Gómez-Pérez, Mariano Fernández-López and Oscar Corcho
Ontological Engineering
1-85233-551-3

Arno Scharl (Ed.)
Environmental Online Communication
1-85233-783-4

Shichao Zhang, Chengqi Zhang and Xindong Wu
Knowledge Discovery in Multiple Databases
1-85233-703-6

Jason T.L. Wang, Mohammed J. Zaki, Hannu T.T. Toivonen and Dennis Shasha (Eds)
Data Mining in Bioinformatics
1-85233-671-4

C.C. Ko, Ben M. Chen and Jianping Chen
Creating Web-based Laboratories
1-85233-837-7

Manuel Graña, Richard Duro, Alicia d'Anjou and Paul P. Wang (Eds)
Information Processing with Evolutionary Algorithms
1-85233-886-0

Colin Fyfe
Hebbian Learning and Negative Feedback Networks
1-85233-883-0

Yun-Heh Chen-Burger and Dave Robertson
Automating Business Modelling
1-85233-835-0

Dirk Husmeier, Richard Dybowski and Stephen Roberts (Eds)
Probabilistic Modeling in Bioinformatics and Medical Informatics
1-85233-778-8

Ajith Abraham, Lakhmi Jain and Robert Goldberg (Eds)
Evolutionary Multiobjective Optimization
1-85233-787-7

K.C. Tan, E.F.Khor and T.H. Lee
Multiobjective Evolutionary Algorithms and Applications
1-85233-836-9

Nikhil R. Pal and Lakhmi Jain (Eds)
Advanced Techniques in Knowledge Discovery and Data Mining
1-85233-867-9

Amit Konar and Lakhmi Jain
Cognitive Engineering
1-85233-975-6

Miroslav Kárný (Ed.)
Optimized Bayesian Dynamic Advising
1-85233-928-4

Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos and Yannis Theodoridis
R-trees: Theory and Applications
1-85233-977-2

Sanghamitra Bandyopadhyay, Ujjwal Maulik, Lawrence B. Holder and Diane J. Cook (Eds)
Advanced Methods for Knowledge Discovery from Complex Data
1-85233-989-6

Marcus A. Maloof (Ed.)

Machine Learning and Data Mining for Computer Security

Methods and Applications

With 23 Figures

Springer

Marcus A. Maloof, BS, MS, PhD
Department of Computer Science
Georgetown University
Washington DC 20057-1232
USA

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2005928487

Advanced Information and Knowledge Processing ISSN 1610-3947
ISBN-10: 1-84628-029-X
ISBN-13: 978-1-84628-029-0

Printed on acid-free paper

Printed in the United States of America (MVY)

9 8 7 6 5 4 3 2 1

Springer Science+Business Media
springeronline.com

To my mom and dad, Ann and Ferris

Foreword

When I first got into information security in the early 1970s, the little research that existed was focused on mechanisms for preventing attacks. The goal was airtight security, and much of the research by the end of decade and into the next focused on building systems that were provably secure. Although there was widespread recognition that insiders with legitimate access could always exploit their privileges to cause harm, the prevailing sentiment was that we could at least design systems that were not inherently faulty and vulnerable to trivial attacks by outsiders.

We were wrong. This became rapidly apparent to me as I witnessed the rapid evolution of information technology relative to progress in information security. The quest to design the perfect system could not keep up with market demands and developments in personal computers and computer networks. A few Herculean efforts in industry did in fact produce highly secure systems, but potential customers paid more attention to applications, performance, and price. They bought systems that were rich in functionality, but riddled with holes. The security on the Internet was aptly compared to "Swiss cheese."

Today, it is widely recognized that our computers and networks are unlikely to ever be capable of preventing all attacks. They are just way too complex. Thousands of new vulnerabilities are reported to the Computer Emergency Response Team Coordination Center (CERT/CC) annually. We might significantly reduce the security flaws through good software development practices, but we cannot expect foolproof security as technology continues to advance at breakneck speeds. Further, the problems do not reside solely with the vendors; networks must also be properly configured and managed. This can be a daunting task given the vast and growing number of products that can be networked together and interact in unpredictable ways.

In the middle 1980s, a small group of us at SRI International began investigating an alternative approach to security. Recognizing the limitations of a strategy based solely on prevention, we began to design a system that could detect intrusions and insider abuse in real time as they occurred. Our research and that of others led to the development of intrusion detection systems. Also

in the 1980s, computer viruses and worms emerged as a threat, leading to software tools for detecting their presence. These two types of detection technologies have been largely separate but complementary. Intrusion detection systems focus on detecting malicious computer and network activity, while antiviral tools focus on detecting malicious code in files and messages.

To succeed, a detection system must know what to look for. This has been easier to achieve with viral detection than intrusion detection. Most antiviral tools work off a list containing the "signatures" of known viruses, worms, and Trojan horses. If any of the signatures are detected during a scan, the file or message is flagged. The main limitation of these tools is that they cannot detect new forms of malicious code that do match the existing signatures. Vendors mitigate the exposure of their customers by frequently updating and distributing their signature files, but there remains a period of vulnerability that has yet to be closed.

With intrusion detection, it is more difficult to know what to look for, as unauthorized activity on a system can take so many forms and even resemble legitimate activity. In an attempt to not miss something that is potentially malicious, many of the existing systems sound far too many false or inconsequential alarms (often thousands per day), substantially reducing their effectiveness. Without a means of breaking through the false-alarm barrier, intrusion detection will fail to meet its promise.

This brings me to this book. The authors have made significant progress in our ability to distinguish malicious activity and code from that which is not. This progress has come from bringing machine learning and data mining to the detection task. These technologies offer a way past the false-alarm barrier and towards more effective detection systems.

The papers in this book address one of the most exciting areas of research in information security today. They make an important contribution to that area and will help pave the way towards more secure systems.

Monterey, CA
January 2005

Dorothy E. Denning

Preface

In the mid-1990s, when I was a graduate student studying machine learning, someone broke into a dean's computer account and behaved in a way that most deans never would: There was heavy use of system resources very early in the morning. I wondered why there was not some process monitoring everyone's activity and detecting abnormal behavior. At least in the case of the dean, it should not have been difficult to detect that the person using the account was probably not the dean.

About the same time, I taught a class on artificial intelligence at Georgetown University. At that time, Dorothy Denning was the chairperson. I knew she worked in security, but I knew little about the field and her research; after all, I was studying rule learning. When I told her about my idea of learning profiles of user behavior, she remarked, "Oh, there's been lots of work on that." I made copies of the papers she gave me, and I started reading.

In the meantime, I managed to convince my lab's system administrator to let me use some of our audit data for machine learning experiments. It was not a lot of data—about three weeks of activity for seven users—but it was enough for a section in my dissertation, which was not about machine learning approaches to computer security.

After graduating, I thought little about the application of machine learning to computer security until recently, when Jeremy Kolter and I began investigating approaches for detecting malicious executables. This time, I started with the literature review, and I was amazed at how widespread the research had become. (Of course, the Internet today is not the same as it was in 1994.)

Ten years ago, it seemed that most of the articles were in computer security journals and proceedings and few were in the proceedings of artificial intelligence and machine learning conferences. Today, there are many publications in all of these forums, and we now have the new field of data mining. Many interesting papers appear in its literature. There are also publications in literatures on statistics, industrial engineering, and information systems. This description does not take into account recent work on fraud detection, which is relevant to applications in computer security, even though it does

not involve network traffic or audit data. Indeed, many issues are common to both endeavors.

Perhaps I am a little better at doing literature searches, but in retrospect, this "discovery" should not have been too surprising since there is overlap among these areas and disciplines. However, what I needed and wanted was a book that brought this work together. In addition to research contributions, I also wanted chapters that described relevant concepts of computer security. Ideally, it would be part textbook, part monograph, and part special issue of a journal.

At the time, Jeremy Kolter and I were preparing a paper for the Third IEEE International Conference on Data Mining. Xindong Wu of the University of Vermont was the program co-chair, and during a visit to his Web site, I noticed that he was an editor of Springer's series on Advanced Information and Knowledge Processing. After a few e-mails and words of encouragement, I submitted a proposal for this book. After peer review, Springer accepted it.

Intended Audience

The intended audience for this book consists of three groups. The first group consists of researchers and practitioners working in this interesting intersection of machine learning, data mining, and computer security. People in this group will undoubtedly recognize the contributors and the connection of the chapters to their past work.

The second group consists of people who know about one field, but would like to learn more about the other. It is for people who know about machine learning and data mining, but would like to learn more about computer security. These people have a dual in computer security, and so the book is also for people who know this field, but would like to learn more about machine learning and data mining.

Finally, I hope graduate students, who constitute the third group, will find this volume attractive, whether they are studying machine learning, data mining, statistics, or information assurance. I would be delighted if a professor used this book for a graduate seminar on machine learning and data mining approaches to computer security.

Acknowledgements

As the editor, I would like to begin by thanking Xindong Wu for his early encouragement. Also early on, I consulted with Ryszard Michalski, Ophir Frieder, and Dorothy Denning; they, too, provided important, early encouragement and support for the project. In particular, I would like to thank Dorothy for also taking the time to write the foreword to this volume.

Obviously, the contributors played the most important role in the production of this book. I want to thank them for participating, for submitting high-quality chapters, and for making my job as editor easy.

Of the contributors, I consulted with Terran Lane and Clay Shields the most. From the beginning, Terran helped identify potential contributors, gave advice on the background chapters I should consider, and suggested that, ideally, the person writing the introductory chapter on computer security would work closely with the person writing the introductory chapter on machine learning. Clay Shields, whose office is next to mine, accepted a fairly late invitation to write an introductory chapter on information assurance. Even before he accepted, he was a valued and close source for papers, books, and ideas.

Catherine Drury, my editor at Springer, was a tremendous help. I really have appreciated her patience, advice, and quick responses to e-mails. Finally, I would like to thank the Graduate School at Georgetown University. They provided funds for production expenses associated with this project.

Bloedorn, Talbot, and DeBarr would like to thank Alan Christiansen, Bill Hill, Zohreh Nazeri, Clem Skorupka, and Jonathan Tivel for their many contributions to their work.

Early and Brodley's chapter is based upon work supported by the National Science Foundation under Grant No. 0335574, and the Air Force Research Lab under Grant No. F30602-02-2-0217.

Kolter and Maloof thank William Asmond and Thomas Ervin of the MITRE Corporation for providing their expertise, advice, and collection of malicious executables. They also thank Ophir Frieder of IIT for help with the vector space model, Abdur Chowdhury of AOL for advice on the scalability of the vector space model, Bob Wagner of the FDA for assistance with ROC analysis, Eric Bloedorn of MITRE for general guidance on our approach, and Matthew Krause of Georgetown for helpful comments on an earlier draft of the chapter. Finally, they thank Richard Squier of Georgetown for supplying much of the additional computational resources needed for this study through Grant No. DAAD19-00-1-0165 from the U.S. Army Research Office. They conducted their research in the Department of Computer Science at Georgetown University, and it was supported by the MITRE Corporation under contract 53271.

Lane would like to thank Matt Schonlau for providing the data employed in the study as well as the results of his comprehensive study of user-level anomaly detection techniques. Lane also thanks Amy McGovern and Kiri Wagstaff for their invaluable comments on draft versions of his chapter.

Washington, DC
March 2005

Mark Maloof

List of Contributors

Eric E. Bloedorn
The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102-7508, USA
bloedorn@mitre.org

Carla E. Brodley
Department of Computer Science
Tufts University
Medford, MA 02155, USA
brodley@cs.tufts.edu

Philip Chan
Department of Computer Sciences
Florida Institute of Technology
Melbourne, FL 32901, USA
pkc@cs.fit.edu

David D. DeBarr
The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102-7508, USA
debarr@mitre.org

James P. Early
CERIAS
Purdue University
West Lafayette, IN 47907-2086, USA
earlyjp@cerias.purdue.edu

Wei Fan
IBM T. J. Watson Research Center
Hawthorne, NY 10532, USA
weifan@us.ibm.com

Klaus Julisch
IBM Zurich Research Laboratory
Saeumerstrasse 4
8803 Rueschlikon, Switzerland
kju@zurich.ibm.com

Jeremy Z. Kolter
Department of Computer Science
Georgetown University
Washington, DC 20057-1232, USA
jzk@cs.georgetown.edu

Terran Lane
Department of Computer Science
The University of New Mexico
Albuquerque, NM 87131-1386, USA
terran@cs.unm.edu

Wenke Lee
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
wenke@cc.gatech.edu

Marcus A. Maloof
Department of Computer Science
Georgetown University
Washington, DC 20057-1232, USA
maloof@cs.georgetown.edu

Matthew Miller
Computer Science Department
Columbia University
New York, NY 10027, USA
mmiller@cs.columbia.edu

Debasis Mitra
Department of Computer Sciences
Florida Institute of Technology
Melbourne, FL 32901, USA
dmitra@cs.fit.edu

Clay Shields
Department of Computer Science
Georgetown University
Washington, DC 20057-1232, USA
clay@cs.georgetown.edu

Salvatore J. Stolfo
Computer Science Department
Columbia University
New York, NY 10027, USA
sal@cs.columbia.edu

Lisa M. Talbot
Simplex, LLC
410 Wingate Place, SW
Leesburg, VA 20175, USA
talbotlm@ieee.org

Gaurav Tandon
Department of Computer Sciences
Florida Institute of Technology
Melbourne, FL 32901, USA
gtandon@cs.fit.edu

Contents

1

Introduction

Marcus A. Maloof

The Internet began as a private network connecting government, military, and academic researchers. As such, there was little need for secure protocols, encrypted packets, and hardened servers. When the creation of the World Wide Web unexpectedly ushered in the age of the commercial Internet, the network's size and subsequent rapid expansion made it impossible retroactively apply secure mechanisms. The Internet's architects never coined terms such as *spam*, *phishing*, *zombies*, and *spyware*, but they are terms and phenomena we now encounter constantly.

Computer security is the use of technology, policies, and education to assure the confidentiality, integrity, and availability of data during its storage, processing, and transmission [1]. To secure data, we pursue three activities: prevention, detection, and recovery [1].

This volume is about the use of machine learning and data mining methods to secure data, and such methods are best suited for detection. *Detection* is simply the process of identifying something's true characteristic. For example, we might want to detect if a program contains malicious logic. Informally, a *detector* is a program that reports positively when it detects the characteristic of interest; otherwise, it reports negatively or nothing at all.

There are two ways to build a detector: We can build or program a detector ourselves, or we can let software build a detector from data. To build a detector ourselves, it is not enough to know *what* we want to detect, for we must also know *how* to detect what we want. The complexity of today's networked computers makes this a daunting task in all but the simplest cases.

Naturally, software can help us determine what we want to detect and how to detect it. For example, we can use software to process known benign and known malicious executables to determine sequences of byte codes unique to the malicious executables. These sequences or *signatures* could serve as the basis for a detector.

We can use software to varying degrees when building detectors, so there is a spectrum from the simple to the ideal. Simple software might calculate the mean and standard deviation of a set of numbers. (A detector might report

positively if any new number is more than three standard deviations from the mean.) The ideal might be a fully automated system that builds detectors with little interaction from users and with little information about data sources. Researchers may debate where the exact point lies, but starting somewhere on this spectrum leading to the ideal are methods of machine learning [2] and data mining [3].

For some detection problems in computer security, existing data mining and machine learning methods will suffice. It is primarily a matter of applying these methods correctly, and knowing that we can solve such problems with existing techniques is important. Alternatively, some problems in computer security are examples of a class of problems that data mining and machine learning researchers find interesting. An an example, for researchers investigating new methods of anomaly detection, computer security is an excellent context for such work. Still other detection problems unique to computer security require new and novel methods of data mining and machine learning.

This volume is divided into two parts: survey contributions and research contributions. The purpose of the survey contributions is to provide background information for readers unfamiliar with information assurance or with data mining and machine learning. In Chap. 2, Clay Shields provides an introduction to information assurance and identifies problems in computer security that could benefit from machine learning or data mining approaches. In Chap. 3, Mark Maloof similarly describes some basic concepts of machine learning and data mining, grounded in applications to computer security.

The first research contribution deals with the problem of worms, spyware, and other malicious programs that, in recent years, have ravaged the Internet. In Chap. 4, Jeremy Kolter and Mark Maloof describe an application of text-classification methods to the problem of detecting malicious executables.

One long-standing issue with detection systems is coping with a large number of false alarms. Even systems with low false-alarm rates can produce an overwhelming number of false alarms because of the amount of data they process, and commercial intrusion detection systems are not an exception. Eric Bloedorn, Lisa Talbot, and Dave DeBarr address this problem in Chap. 5, where they discuss their efforts to reduce the number of false alarms a system presents to analysts.

However, it is not only false alarms that have proven distracting to analysts. Legitimate but highly redundant alarms also contribute to the alarm flood that overloads analysts. Klaus Julisch addresses this broader problem in Chap. 6 by grouping alarms according to their root causes. The number of resulting alarm groups turns out to be much smaller than the initial number of elementary alarms, which makes them much more efficient to analyze and process.

Determining features useful for detection is a challenge in many domains. James Early and Carla Brodley describe, in Chap. 7, a method of deriving features for network intrusion detection designed expressly to determine if a protocol is being used improperly.

Once we have identified features, computing them may require differing costs or amounts of effort. There are also costs associated with operating the detection system and with detecting and failing to detect attacks. In Chap. 8, Wenke Lee, Wei Fan, Sal Stolfo, and Matthew Miller discuss their approach for taking such costs into account.

Algorithms for anomaly detection build models from normal data. If such data actually contain the anomalies we wish to detect, then it could reduce the effectiveness of the resulting detector. Gaurav Tandon, Philip Chan, and Debasis Mitra discuss, in Chap. 9, their method for cleaning training data and removing anomalous data. They also investigate a variety of representations for sequences of system calls and the effect of these representations on performance.

As one can infer from the previous discussion, the domain of intrusion detection presents many challenges. For example, there are costs, such as those associated with mistakes. New data arrives continuously, but we may be uncertain about its true nature, whether it is malicious or benign, anomalous or normal. Moreover, training data for malicious behavior may not be available. In Chap. 10, Terran Lane argues that such complexities require a decision-theoretic approach, and proposes such a framework based on partially observable Markov decision processes.

Part I

Survey Contributions

2

An Introduction to Information Assurance

Clay Shields

2.1 Introduction

The intuitive function of computer security is to limit access to a computer system. With a perfect security system, information would never be compromised because unauthorized users would never gain access to the system. Unfortunately, it seems beyond our current abilities to build a system that is both perfectly secure and useful. Instead, the security of information is often compromised through technical flaws and through user actions.

The realization that we cannot build a perfect system is important, because it shows that we need more than just protection mechanisms. We should expect the system to fail, and be prepared for failures. As described in Sect. 2.2, system designers not only use mechanisms that *protect* against policy violations, but also *detect* when violations occur, and *respond* to the violation. This response often includes analyzing why the protection mechanisms failed and improving them to prevent future failures.

It is also important to realize that security systems do not exist just to limit access to a system. The true goal of implementing security is to protect the information on the system, which can be far more valuable than the system itself or access to its computing resources. Because systems involve human users, protecting information requires more than just technical measures. It also requires that the users be aware of and follow security policies that support protection of information as needed.

This chapter provides a wider view of information security, with the goal of giving machine learning researchers and practitioners an overview of the area and suggesting new areas that might benefit from machine learning approaches. This wider view of security is called *information assurance*. It includes the technical aspects of protecting information, as well as defining policies thoroughly and correctly and ensuring proper behavior of human users and operators. I will first describe the security process. I will then explain the standard model of information assurance and its components, and, finally, will describe common attackers and the threats they pose. I will conclude

with some examples of problems that fall outside much of the normal technical considerations of computer security that may be amenable to solution by machine learning methods.

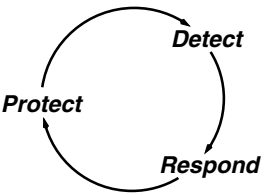


Fig. 2.1. The security cycle

2.2 The Security Process

Human beings are inherently fallible. Because we will make mistakes, our security process must reflect that fact and attempt to account for it. This recognition leads to the cycle of security shown in Fig. 2.1. This cycle is really very familiar and intuitive, and is common in everyday life, and is illustrated here with a running example of securing an automobile.

2.2.1 Protection

Protection mechanisms are used to enforce a particular policy. The goal is to prevent things that are undesirable from occurring. A familiar example is securing an automobile and its contents. A car comes with locks to prevent anyone without a key from gaining access to it, or from starting it without the key. These locks constitute the car's protection mechanisms.

2.2.2 Detection

Since we anticipate that our protection mechanisms will be imperfect, we attempt to determine when that occurs by adding detection mechanisms. These monitor the system, try to locate any policy violations that have occurred, and then provide an alert or alarm to that fact. Our familiar example is again a car. We know that a determined thief can gain entry to a car, so in many cases, cars have alarm systems that sound loudly to attract attention when they detect what might be a theft.

However, just as our protection mechanisms can fail or be defeated, so can detection mechanisms. Car alarms can operate correctly and sound the alarm when someone is breaking in. This is termed a *true positive*; the event that is looked for is detected. However, as many city residents know, car alarms can

also go off when there is no break-in in progress. This is termed a *false positive*, as the system is indicating it detected something when nothing was happening. Similarly, the alarm can fail to sound when there is an intrusion. This is termed a *false negative*, as the alarm is indicating that nothing untoward is happening when in fact it is. Finally, the system can indicate a *true negative* and avoid sounding when nothing is going on.

While these terms are certainly familiar to those in the machine learning community, it is worth emphasizing the fallibility of detection systems because the rate at which false results occur will directly impact whether the detection system is useful or not. A system that has a high false-positive rate will quickly become ignored. A system that has a high false-negative rate will be useless in its intended purpose.

2.2.3 Response

If, upon examination of an alert provided by our detection system, we find that a policy violation has occurred, we need to respond to the situation. Response varies, but it typically includes mitigating the current situation, analyzing what happened, recovering from any damage, and improving the protection and detection mechanisms to prevent similar future occurrences.

For example, if our car alarm sounds and we see someone breaking in, we might respond by summoning the police to catch or run off the thief. Some cars have devices that allow police to determine their location, so that if a car is stolen, it can be recovered. Afterwards, we might try to prevent future incidents by adding a locking device to the steering wheel or parking in a locked garage. If we find that the car was broken into and the alarm did not sound, we might choose also to improve the alarm system.

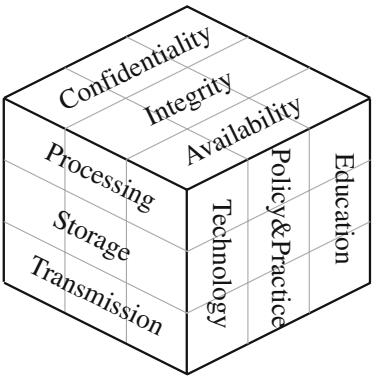


Fig. 2.2. The standard model of information assurance

2.3 Information Assurance

The standard model of information assurance is shown in Fig. 2.2 [4]. In this model, the security properties of confidentiality, integrity, and availability of information are maintained in the different locations of storage, transport, and processing by technological means, as well as through the process of educating users in the proper policies and practices. Each of these properties, location, and processes is described below.

The term *assurance* is used because we fully expect failures and errors to occur, as described above in Sect. 2.2. Recognizing this, we do not expect perfection and instead work towards a high level of confidence in the systems we build.

Though this model can apply to virtually any system which includes information flow, such as the movement of paper through an office, our discussion will naturally focus on computer systems.

2.3.1 Security Properties

The first aspects of this model we will examine are the security properties that can be maintained. The traditional properties that systems work towards are confidentiality, integrity, and availability, though other properties are sometimes included. Because different applications will have different requirements, a system may be designed to maintain all of these properties or only a chosen subset as needed, as described below.

Confidentiality

The confidentiality property specifies that only entities authorized to access some particular information are allowed to do so. This is the property that maintains the secrecy of information on a need-to-know basis, and is the most intuitive.

The most common mechanisms for protecting confidentiality are access control and encryption. Access control mechanisms prevent any reading of the information until the accessing entity, either a person or computer process acting on behalf of a person, prove that it is authorized to do so. Encryption does not prevent access to the information, but instead obfuscates the information so that even if it is read, it is not understandable.

The mechanisms for detecting violations of confidentiality and responding to them vary depending on the situation. In the most general case, public disclosure of the information would indicate loss of confidentiality. In an electronic system, violations might be detectable through audit and logging systems. In situations where the actions of others might be influenced by the release of confidential information, such changes in behavior might indicate a violation. For example, during World War II, an Allied effort broke the German Enigma encryption system, violating the confidentiality of German

communications. Concerned that unusual military success might indicate that Enigma had been broken, the Allies were careful to not exploit all information gained [5]. Though it will vary depending on the case, there may be learning situations that involve monitoring the actions of others to see if access to confidential information has been compromised.

There might be an additional requirement that the existence of information be kept confidential as well, in which case, encryption and access control might not be sufficient. This is a more subtle form of confidentiality.

Integrity

In the context of information assurance, *integrity* means that only authorized entities can alter information within a system. This is the property that keeps information from being changed when it should not be.

While we will use the above definition of *integrity*, it is an overloaded term and other meanings exist. *Integrity* can be used to describe the reliability of information. For example, a data source has integrity if it provides accurate data. This is sometimes referred to as *origin integrity*. *Integrity* can also be used to refer to a state that exists in several systems; if the state is consistent, then there is high integrity. If the distributed states are inconsistent, then there is low integrity.

Mechanisms exist to protect data integrity and to detect when it has been violated. In practice, protection mechanisms are similar to the access control mechanisms for confidentiality, and in implementation may share common components. Detecting integrity violations may involve comparing the data to a different copy, or the use of cryptographic hashes. Response typically involves repairing the changes by reverting to an earlier, archived copy.

Availability

Availability is the property that the information on a system is obtainable when needed. Information that is kept secret and unaltered might still be made unavailable by attackers conducting *denial-of-service* attacks.

The general approach to protecting availability is to limit the amount of system resources that can be consumed, either by rate-limiting or by requiring access control. Another common approach is to over-provision the system. Detection of availability is generally conducted by polling to see if the resources are there. It can be difficult to determine if some system is unavailable because of attack or because of some system failure. In some situations, there may be learning problems to be solved to differentiate between failure and attack conditions.

Response to availability problems generally includes reducing the system load, or adding more capacity to a system.

Other Components

The properties above are the classic components of security, and are sufficient to describe many situations. However, there has been some discussion within the security community for the need for other properties to fully capture requirements for other situations. Two of the commonly suggested additions, authentication and non-repudiation, are discussed below.

Authentication

Both the confidentiality properties and integrity properties include a notion of authorized entities. The implication is that the system can accurately identify entities in some manner and, given their identity, provide or deny access. The authentication property ensures that all entities in a system have their identities properly verified.

There are a number of ways to conduct authentication and protect against false identification. For individuals, the standard mnemonic for describing classes of authentication mechanisms is, *What you are*, *what you have*, and *what you know*.

- "What you are" refers to specific physical attributes of an individual that can serve to differentiate him or her from others. These are commonly biometric measurements of such things as fingerprints, hand size and shape, voice, or retinal patterns. Other attributes can be used as well, such as a person's weight, gait, face, or potentially DNA. It is important to realize that these systems are not perfect. They have false-positive and false-negative rates that can allow false authentication or prohibit legitimate users from accessing the system. Often the overall accuracy of a biometric system can be improved by measuring different attributes simultaneously. As an aside, many biometric systems have been shown to be susceptible to simple attacks, such as plastic bags of warm water placed on a fingerprint sensor to reactivate the prior latent print, or pictures held in front of a camera [6, 7]. Because these attacks are generally observable, it may be more appropriate for biometric authentication to take place under human observation. It might be a vision or machine learning problem to determine if this type of attack is occurring.
- "What you have" describes some token that is carried by a person that the system expects only that person to have. This token can take many forms. In a physical system, a key could be considered an access token. Most people have some form of identification, which is a token that can be used to show that the issuer of the identification has some confidence in the carrier's identity. For computer systems, there are a variety of authentication tokens. These commonly include devices that generate pass codes at set intervals. Providing the correct pass code indicates possession of the device.

- "What you know" is the most familiar form of authentication for computer users. In this form of authentication, users prove their identity by providing some information that only they would know that can be verified. The most common example of this is a password, which is a secret shared by the individual and the end system conducting the authentication. The private portion of a public/private key pair is also an example of what you know.

More recently, it has been shown that it is possible to use location as another form of authentication. With this "where you are" authentication, systems can use signals from the Global Positioning System to verify that the authentication attempt is coming from a particular location [8].

Authenticating entities across a network is a particularly subtle art. Because attackers can potentially observe, replay, and manipulate network traffic, designing protocols that are resistant to attack is very difficult to do correctly. This has been a significant area of research for some time [9].

The mechanisms outlined above provide the basis for authentication protection. Detecting authentication failures, which would be incorrectly identifying a user as a legitimate user, can often be done on the basis of behavior after authentication. There is a significant body of work addressing user profiling to detect aberrant behavior that might indicate an authentication failure. One appropriate response is to revoke the credentials gained through authentication. The intruder can also be monitored to better understand attacker behavior.

Non-repudiation

The non-repudiation property makes it difficult for any entity to deny that it performed some action. A system with non-repudiation will allow entities to be held responsible for what they do. Very few computer systems have effective non-repudiation mechanisms. In general, logging and audit data is recorded, but is often unreliable. More effective non-repudiation systems require the use of strong cryptographic mechanisms, though these require significant overhead for additional processing and key distribution.

System Security Requirements

Different systems have different security requirements, which might include some or all of the properties discussed above. A financial system might need all five: Confidentiality is required to protect the privacy of records; integrity is needed to maintain a proper balance; availability allows access to money when required; authentication keeps unauthorized users from withdrawing funds; and non-repudiation keeps users from arguing that they did not take funds out and keeps the institution from denying it received a deposit.

Other systems do not require that level of security. For example, a Web page may be publicly available and therefore not require any confidentiality.

The owner of the page might still desire that the integrity of the page be maintained and that the page be available. The owner of a wiki might allow anyone to edit the page and hence be unconcerned with integrity, but might require that users authenticate to prevent non-repudiation of what they edit.

2.3.2 Information Location

The model of information assurance makes a clear distinction about where information resides within a system. This is because the mechanisms used to protect, detect, and respond differ for each case.

Processing

While information is being processed in a computer system, it is loaded into memory, some of which may be virtual memory pages on a disk, and into the registers of the CPU. The primary protection mechanisms in this case are designed to prevent processes on the system from reading or altering each other's memory space. Modern computer systems contain a variety of hardware and software mechanisms to provide each process with a secure, independent memory space.

Confidentiality can also be lost through information leaking from a process. This can happen through a covert channel, which is a mechanism that uses shared system resources not intended for communication to transmit information between processes [10]. It is possible to prevent or rate-limit covert channels, though it can be difficult to detect them. Response varies, but includes closing the channel through system updates. Loss of confidentiality can also occur through electromagnetic radiation from system components, such as the CPU, bus, video card, and CRT. These produce identifiable signals that can be used to reconstruct information being processed on the system [11, 12]. Locations that work with highly classified information are often constructed to keep this radiation from escaping.

Storage

Information in storage resides on some media, either within the system or outside of it. The protection mechanisms for information stored on external media are primarily physical, so that the media cannot be stolen or accessed. It is also possible and frequently desirable to encrypt information that is stored externally. Detection often consists of alarm systems to detect illicit access, and inventory systems to detect missing media. To detect integrity violations, cryptographic hashes can be computed for the stored data and kept separately from the media, then periodically checked [13]. At the end of its useful lifetime, media should be destroyed instead of discarded.

Information that is stored within a system is protected by operating systems mechanisms that prevent unauthorized access to the data. These include

access control mechanisms and, increasingly, mechanisms that keep stored information encrypted. There are many methods of detecting unauthorized access. These generally fall under the classification of *intrusion detection*. Intrusion detection systems can further be classified as *signature-based*, which monitor systems for known patterns of attack, or as *anomaly detection*, which attempt to discern attacks by identifying abnormal activity.

Transport

Information can be transported either physically or electronically. While it is natural to think of transmitted data over a network, for large amounts of data it can be significantly faster to send media through the mail or via an express delivery service. Data transported in this manner can be protected using encryption or physical security, such as locked boxes.

Data being transported over the network is best protected by being encrypted, and this functionality is common in existing software. In the future, quantum cryptographic methods will increasingly be used to protect data in transmission. Using quantum cryptography, two communicating parties can agree on an encryption key in a way that inherently detects if their agreement has been eavesdropped upon [14].

2.3.3 System Processes

While most computer scientists focus on the technological processes involved in implementing security, technology alone cannot provide a complete security solution. This is because human users are integral in maintaining security. The model of information assurance recognizes this, and gives significant weight to human processes. This section provides more detail about the processes that are used to provide assurance.

Technology

Every secure information system requires some technological support to be secure. In our discussion thus far, we have mentioned a number of technological mechanisms that exist to support the protect, detect, and respond cycle. These include systems that provide authentication; access control mechanisms that limit what authenticated users can view and change; and intrusion detection systems that identify when these prior mechanisms have failed.

There are other technological controls that protect information security that are not part of computer systems, however, and which are often forgotten. The foremost of these are physical security measures. Access control on a computer system is of little use if an attacker has physical access and can simply steal the computer or its archive media and off-load the data later. Large corporations are typically more aware of this than universities, and often implement a number of controls designed to limit physical access. The efficacy

of these devices can vary, however. Some systems use cards with magnetic stripes that encode an employee number that is also shown on the front of the card, which may be worn around the neck. Anyone who is able to read this number can then duplicate the card with a $600 card writer. Radio frequency identification (RFID) tags are also becoming popular. These frequently respond to a particular radio-frequency query with a static ID. Because there is no control over who can query the tag, anyone can read and potentially duplicate the tag. Impersonation in these cases may be relatively simple for someone who feels comfortable that they might not be noticed as out of place within a secure area.

Policy and Practice

While technological controls are important, they are not sufficient simply because they are designed to allow some access to the system. If the people who are permitted to access systems do not behave properly, they can inadvertently weaken the security of the system. A common example is users who open or run attachments that they receive over e-mail. Because users are allowed to run processes on the system, the access control mechanisms prove ineffective.

Organizations that do security well therefore create policies that describe how they expect their users to act, and provide best-practice documents that detail what can be done to meet these policies. Again, these policies must go beyond the computer system. They should include physical security as well as policies that govern how to answer phones, how to potentially authenticate a caller, and what information can be provided. These policies are directed towards stopping *social engineering*, in which an outside attacker tries to manipulate people into providing sufficient information to access the system.

Education

Having defined policies and practices is not sufficient. Users must know them, accept them, and follow them. Therefore, user education is a necessity. Proper education includes new-user orientation and training, as well as recurring, periodic training to keep the material fresh. Some organizations include security awareness and practice as part of job performance evaluation.

2.4 Attackers and the Threats Posed

It is difficult to determine what security measures are needed without an understanding of what capabilities different types of attackers possess. In this section, we will examine different classes of attackers, what unique threats each might pose, and how those threats are addressed in the information assurance model.

It is important to note that attackers will generally attempt to compromise the system the easiest way that they can, given their capabilities. For example, an attacker might have access to an encrypted password file and to network traffic. In this case, it might be easier to "sniff" unencrypted passwords off the network instead of making the effort to decrypt the password file. A similar attack for someone with physical access to the system might be to place a hardware device to capture keystrokes instead of making the effort of guessing an encryption key. Other attackers might find it easier to attack the encryption; for example, government intelligence agencies might want to limit their exposure to detection. In this case, given their desire for secrecy and massive computing facilities, it might be easiest to attack the encryption.

2.4.1 Worker with a Backhoe

While they hardly seem like fearsome hackers and appear quite comical, construction workers might be one of the most damaging accidental attackers. Given the prevalence of underground power and network wiring, it is a common occurrence for lines to be severed. This can easily rob a large area of power and network access, making services unavailable. It can also take a significant amount of time to make repairs. The best defense is over-provisioning through geographically separate lines for networking or power, or possession of a separate power generator.

As a military tactic, the equivalent of an attacker with a backhoe has proven quite effective in the past, and could be again in the future. In the early days of World War I, British sailors located, raised, and severed an underwater telephone line that was used to transmit orders to the German Navy. Without the telephone line, the Germans had to transmit orders over radio, allowing the British to attack the encryption, eventually with significant success [5]. It is easy to believe that similar actions could occur today to force broadcast communication.

2.4.2 Ignorant Users

Many modern security problems are caused by otherwise well-intentioned users who make mistakes that weaken, break, or bypass security mechanisms. Users who open or run attachments received by e-mail are a clear example of this. Similarly, users who are helpful when contacted over the phone and provide confidential internal information, such as the names of employees and their phone numbers or even passwords, pose a threat. These types of employees are best prevented using proper policies, practices, and education.

2.4.3 Criminals

While most criminals lack any significant computer savvy, they are a serious threat because of the value of computer equipment. Theft of electronics is a

common occurrence, because of the potential resale value. Small items, such as laptops and external drives, are easy to steal and can contain significant amounts of information. Such information might have inherent value – passwords and account numbers are examples. Theft or misplacement could also cause financial loss as a result of legal action, especially if the lost data are like medical records, which should be kept private. Unfortunately, there is no way to know if equipment has been stolen for its value or to gain access to its information, and generally the worst case should be assumed.

2.4.4 Script Kiddies

The attackers discussed thus far have not been specifically targeting information systems. The somewhat denigrating term *script kiddie* applies to attackers who routinely attempt to remotely penetrate the security of networked computer systems, but lack the skills or knowledge to do so in a sophisticated way. Instead, they use a variety of tools that have been written by more capable and experienced people.

While they generally do not have a specific target in mind, script kiddies tend to be exceptionally persistent, and will scan hundreds of computers looking for vulnerabilities that they are able to exploit. They do not present a severe threat individually, but will eventually locate any known security hole that is presented to the network. As an analogy, imagine a group of roving youths who go from house to house trying the doors and windows. If the house is not properly secured, they will eventually find a way in. Fortunately, script kiddies are relatively easy to stop with good technological security practice.

2.4.5 Automated Agents

While script kiddies are often actively looking for security vulnerabilities, the scope of their efforts pale compared to the number of automated agents in the current Internet. These are programs, often called *malware*, that run with the sole purpose of spreading themselves to as many computers as possible. Many of these then provide their creator the ability to access information within a system, or to use its resources for other attacks. While there are many types of malware, there are a few specific types that merit mention.

Worm

A worm is a self-propagating piece of code that exploits vulnerabilities in remote systems to spread itself. Typically, a worm will infect a system and then start scanning to find other vulnerable systems and infect those. A worm might also have other functionality in its payload, including notifying its creator that it has compromised a new host and allowing access to it. It might also scan the compromised machine for interesting information and then send it to its creator.

Virus

Though the term *virus* has fallen into common use to describe any type of malware which spreads between computers, a more precise definition is that it is a piece of code which gets added to existing programs that only runs when they run. At that time, the virus adds its code to other programs on the system.

Trojan

Named after the famous Trojan horse, a *Trojan* is a piece of code that purports to do one thing but actually does another, or does what it says while also maliciously doing something else.

It should be immediately evident that a clear classification of malware into these separate categories may not be possible because one piece of malicious code may exhibit more than one of these characteristics. Many recent worms, for example, were also Trojans. They spread over the network directly, but also would search each machine compromised for e-mail addresses and then falsify e-mail that included a Trojan version of the worm. If the recipient were to open and run the attachment, the worm would continue from there.

These agents are stopped by common technological measures, the existence of which indicate how large the problem is. Unfortunately, it can be time-consuming and expensive to apply the proper patches to a large network of computers. Additionally, new malware variants are appearing that target new operating systems, like those in cellular phones, which do not have the same wealth of protection mechanisms.

2.4.6 Professional System Crackers

Unlike script kiddies, who lack the skills and experience to penetrate a specific target, professional crackers master a broad set of tools and have the intelligence and sophistication to pick and penetrate a particular target. They might do so on behalf of a government, or for financial gain, either independently or as part of an organized crime ring. While part of the attack might be conducted remotely over the network, they might also attempt to gain physical access to a particular target; to go through trash looking for useful information; or to gain the assistance of a helpful but ignorant user.

These attackers can be subtle and patient. There is no simple solution to mitigating the threat they present; instead, the full range of security measures is required.

2.4.7 Insiders

While the most popular image of a computer attacker is that of the professional cracker, they account for only a very small percentage of all attacks.

Instead, the most common attacker, and the one who is most often successful, is the insider [15]. An insider is someone who has access to some or all parts of the computer system, then misuses that access. Note that access may not be electronic; an insider may simply step over to someone else's desk while they are away and use their computer.

The insider is the most subtle and difficult attacker to identify. There is perhaps significant room for detecting insider attacks.

2.5 Opportunities for Machine Learning Approaches

It is evident from the other chapters in this book that machine learning and data mining are naturally most applicable to the detection phase of the security cycle. This section contains suggestions for other areas that might be amenable to machine learning approaches.

- When an attacker manages to acquire data without being detected, the information often ends up publicly available on the Internet. It might be possible to detect successful intrusions by making queries to search engines, such as Google. The difficulty here might not be a machine learning problem, but a data retrieval one: How is it possible to find information through queries without revealing what the information is to an attacker observing the queries?
- Many biometric authentication systems are subject to attacks that lead to false positives in identification. Most of these attacks are easily detected by human observers. A vision or machine learning problem might be to perform automated observation of biometric systems to detect these attacks.
- Education is an important part of the security process. While not all failures of proper user education will result in loss of confidentiality, integrity, or availability of data, problems short of these bad results might indicate the potential for future problems. Depending on the system, it might be possible to identify user behavior that does not result in a security violation but indicates that the user is not aware of good security practice.
- Insiders are the most insidious attackers, and the hardest to detect. One approach to detecting and identifying insiders might be to correlate user idle times between machines that are located in close proximity. A user becoming idle shortly before some other system ceases its idle time could indicate a user walking over to and using another unlocked system.
- Similarly, many companies use authentication systems that allow the physical location of employees to be known to some degree. Using data from these systems, it might be possible to identify insider attackers by finding odd movements or access patterns within a building or campus.
- Some insiders do not need to move to conduct attacks; instead, they are given broad access to a data processing system and trusted to limit the

data they examine to what they need to do their job. Without knowing what particular subset of data they should have access to, it might be possible to detect insider attackers based on the patterns of data access that are different than others who have similar responsibilities.

- Many outside attackers succeed by exploiting the trust and helpfulness of people within an organization. It might be possible to detect social engineering attacks by tracking patterns of phone calls coming into an organization. This data would likely be available in phone records.
- It can be difficult to classify availability failures as accidental or intentional. For example, a sudden increase in network consumption can indicate a denial-of-service attack, or simply a suddenly popular Web link. It might be possible to differentiate between them by examination of the network traffic.
- Automated agents, such as worms or Trojans, might be detectable based on patterns of outgoing network traffic.

2.6 Conclusion

Most machine learning work has focused on detecting technical attacks that originate from outside a particular network or system. This is actually a very small part of the security space. The ideas above touch on some aspects of security that seem to have appropriate data available, but that do not seem to have been as closely examined. There are certainly many existing and emerging areas where machine learning approaches can bring new improvements in security.

3

Some Basic Concepts of Machine Learning and Data Mining

Marcus A. Maloof

3.1 Introduction

Central to the approaches described in this volume is the use of algorithms to build models from data. Depending on the algorithm, the model, or the data, we might call such an activity pattern classification [16, 17], statistical pattern recognition [18, 19], information retrieval [20], machine learning [2, 21, 22], data mining [3, 23, 24], or statistical learning [25]. Although finding the boundaries between concepts is important in all of these endeavors, in this chapter, we will instead focus on their commonalities. Indeed, there are methods common to all of these disciplines, and methods from all have been applied to problems in computer security.

Researchers and practitioners apply such algorithms to data for two main reasons: to predict new data and to better understand existing data. Regarding the former reason, one gathers data, applies an algorithm, and uses the resulting model to predict something about new data. For instance, we may want to predict based on audit data if a user's current session is similar to old ones.

Regarding the second reason, one gathers data, applies an algorithm, and analyzes the resulting model to gain insights into the data that would be difficult to ascertain if examining only the data itself. For example, by analyzing a model derived from audit data, we might conclude that a particular person's CPU usage is far higher than that of others, which might suggest inappropriate usage of computing resources. In this scenario, the need to understand the model an algorithm produces restricts the use of some algorithms, for some algorithms produce models that are easily understood, while others do not.

The sections that follow provide an introductory overview of machine learning and data mining, especially as it relates to applications in computer security and to the chapters in this volume. In Sect. 3.2, we describe the process of transforming raw data into input suitable for learning and mining algorithms. In Sect. 3.3, we survey several such algorithms, and in Sect. 3.4, we discuss methods for evaluating the models these algorithms produce. After

this overview, we briefly examine ensemble methods and sequence learning, two important topics for research and applications. In Sect. 3.6, we note online sources of information, implementations, and data sets. Finally, in Sect. 3.7, we identify resources for further study.

3.2 From Data to Examples

Three important activities in computer security are prevention, detection, and recovery [1]. If taking a machine learning or data mining approach to computer security, then the first step is to identify a data source supporting our desired activity. Such data sources include keystroke dynamics, command sequences, audit trails, HTTP logs, packet headers, and malicious executables. For instance, one could improve preventive measures by mining logs to discover the most frequent type of attack. One could also learn profiles of user behavior from an audit trail to detect misuse of the computer system.

A raw data source is rarely suitable for learning or mining algorithms. It almost always requires processing to remove unwanted and irrelevant information, and to represent it appropriately for such algorithms. Input to learning and mining algorithms is called *cases*, *samples*, *examples*, *instances*, *events*, and *observations*. A tabular representation is common for such input, although others include relational, logical (propositional and first-order), graphical, and sequential representations.

Table 3.1. A hypothetical set of examples derived from raw audit data

login	real	system	user	chars	blocks	cpu	hog
bugs	0.17	0.02	0.02	1030	9	0.5	0.2
bugs	0.07	0.02	0.02	64	1	0.5	0.5
bugs	0.05	0.03	0	64	1	0	0.67
bugs	4.77	0.07	0.02	22144	0	0.2	0.02
bugs	0.03	0.03	0	64	0	0	1
bugs	0.03	0	0.02	839	0	1	0.5
bugs	0.02	0	0	28	0	0	1
daffy	0.08	0.03	0	419	1	0	0.4
daffy	25352.5	0.2	0.22	24856	25	0.52	0
daffy	0.1	0.03	0	419	3	0	0.33
daffy	0.08	0.02	0	419	2	0	0.2
daffy	0.07	0.02	0.02	419	1	0.5	0.5

In a tabular representation, each example consists of a set of attributes and their values. Table 3.1 shows a hypothetical set of examples for users and metrics derived from the UNIX `acctcom` command, an accounting tool. From these examples, we could build models for predicting `login` based on the audit metrics or for detecting when a user's hog factor (`hog`) is atypical.

Researchers and practitioners often give special status to the attribute they wish to predict, calling it generically the *class label*, or simply the *label*, terms typically applied to attributes with discrete values. We should also note that there are applications, especially in computer security, in which attribute values and class labels for some examples are missing or difficult to determine. Regarding class labels in particular, there is a spectrum between a fully labeled set of examples and a fully unlabeled set. In the following discussion, we will use the term *example* to mean a set of attribute values *with* a label and use the term *observation* to mean a set of attribute values *without* a class label.

To transform raw data into a set of examples, we can apply a myriad of operations. We cannot be exhaustive here, but examples of such operations include

- adding a new attribute calculated from others
- mapping values of numeric attributes to the range $[0, 1]$
- mapping values of numeric attributes to discrete values (e.g., [26])
- predicting missing attribute values based on other values (e.g., [27])
- removing attributes irrelevant for prediction (e.g., [28])
- selecting examples relevant for prediction (e.g., [29])
- relabeling mislabeled examples (e.g., [30])

The transformation of raw data into a set of examples may seem like an easy process, but it can be quite difficult and sometimes impossible, for we must give the resulting examples not only the correct form, but also the correct function. That is, we must create examples that facilitate learning and mining.

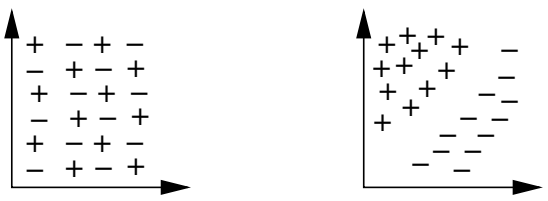


Fig. 3.1. Two representation spaces. (**a**) A space where learning is difficult for an algorithm that builds "straight-line" models. (**b**) A space where learning is easy for such an algorithm

To illustrate, assume we have an algorithm that constructs models that are lines in two-dimensional space. A good model is one that separates the positive examples and the negative examples. If the positive and negative examples we present to the algorithm are organized in a "checkerboard pattern" (see Fig. 3.1a), then it will be impossible for the algorithm to find an adequate

model. On the other hand, if the examples we present are clustered together and are linearly separable, as shown in Fig. 3.1b, then it will be easier for the algorithm to construct a good model.

Numerous factors complicate the transformation of raw data into examples: the amount of data, the potential number of attributes, the domains of those attributes, and the number of potential classes. However, a large amount of data does not always make for a difficult learning or mining problem, for the complexity of what the algorithm must learn or mine is also critical.

These difficulties of transforming a raw data source into a set of examples have prompted some researchers to investigate automated methods of finding representations for examples. Methods of *feature construction*, *feature engineering*, or *constructive induction* automatically transform raw data into examples suitable for learning. There have been proposals for general methods, which we can apply to any domain, but because of the complexity of such a task, we must often devise domain-specific or even ad hoc methods for transforming raw data into examples.

There are also costs associated with attributes, examples, and mistakes on each class. For some domains, we may know these costs; for others, we may have only anecdotal evidence that one thing is more costly than another. Some attribute values may be easy to collect or derive, while doing so for others may be difficult or costly. For example, obtaining attribute values when a connection is first established is less costly than computing such values throughout the connection. It is also less costly to extract attribute values from the packet header than from the data buffer, which could be encrypted.

The examples themselves may have different associated costs. If we are interested in building a system to identify plants, collecting examples of plants that grow locally in abundance is less costly than collecting examples of endangered plants that grow only in remote forests. Similarly, we can easily generate traces of attacks if they have been scripted. However, it is more difficult – and more costly – to obtain traces of novel unscripted attacks.

Finally, the mistakes on classes are often different. For example, a system that detects malicious executables can make two types of mistakes: It may identify a benign program as being malicious, or it may identify a malicious program as being benign. These mistakes do not have the same cost. If the system informs a user that a word processor is malicious, the user will probably just ignore the alert. However, if a new worm goes undetected, it could erase important information and render the networked computers useless.

The number of examples we collect of each class is also important. As an illustration, assume that we want to build a model to classify behavior as either acceptable or malicious. Now assume, somewhat absurdly, that we cannot collect any examples of malicious behavior. Obviously, no algorithm can build

a model of malicious behavior without examples.[1] So how many examples of malicious behavior should we gather? One or two will not be sufficient, but the point is that too few examples can impede learning and mining algorithms from building adequate models. It is often the case with *skewed data sets*, in which we have many examples of one class and few examples of another, that an algorithm builds a model that almost always predicts the class with the most examples (i.e., the majority class). It is also often the case that the class with the fewer examples (i.e., the minority class) is the most important for prediction.

There are methods of handling both costs and skew. For instance, if we know the cost of mistakes for a domain, then there are techniques for generating a model that minimizes such costs [31]. The difficulty arises when we do not know the costs, or do not know them precisely [32]. A complete discussion of these issues is beyond the scope of this chapter, although there are examples of approaches in other chapters of this volume. So, in the next section, we will proceed by examining how algorithms build models from examples.

3.3 Representations, Models, and Algorithms

Learning and mining algorithms have three components: the representation, the learning element, and the performance element. The *representation*, *hypothesis language*, or *concept description language* is a formalism used for building models. A *model*, *hypothesis*, or *concept description* is a particular instantiation of a representation. Crucially, the representation determines what models can be built.

The learning element builds a model from a set of examples. The performance element applies the model to new observations. In most cases, the model is a compact summary of the examples, and it is often easier to analyze the model than the examples themselves. This also means that we can archive or discard the examples. In most cases, the model generalizes the examples, which is a desirable property, for we need not collect all possible examples to produce a useful model. Moreover, we can use the model to make predictions about examples absent from the original set. Researchers have used a variety of representations for such models, including trees, rules, graphs, probabilities, first-order logic, the examples themselves, and coefficients of linear and nonlinear equations.

Once we have formed a set of examples, then learning and mining algorithms can support a variety of analysis tasks. For instance, we may build a model from all examples and detect anomalous events or observations (i.e., anomaly detection). We may divide the examples into two or more classes,

[1] Note that algorithms for anomaly detection build models of normal behavior and then infer that anomalous behavior is malicious. This is a slightly different issue than the one considered here.

build a model, and classify new observations as being of one of the classes. Two-class problems are often referred to as *detection tasks*, with class labels of *positive* and *negative*. Instead of predicting one of a set of values, we may want to predict a numeric value (i.e., regression). We also may want to examine the associations between sets of attributes. Finally, we may want to examine the models themselves to gain insights into the data from which they were derived.

Previously, we noted the spectrum between a fully labeled set of examples and a fully unlabeled set. *Supervised learning* is learning from a fully labeled set of examples, whereas *unsupervised learning* is learning with a fully unlabeled set. Researchers also use the terms *discovery*, *mining*, and *clustering* to describe this activity. Recent work along this spectrum has given rise to *semi-supervised learning*, where we have a partially labeled set of examples.

When applying algorithms to examples, if we can gather a sufficient number of examples in advance, then we can process them in a single batch. However, for many applications, examples are distributed over time and arrive as a stream, in which case, the algorithm must process them online. We can use a batch algorithm to process examples online if we simply store all available examples and reapply the algorithm when new ones arrive. For large data sets and complex algorithms, this is impractical in both time and space, so in such situations, we can use an incremental algorithm that uses new examples to modify the existing model.

An important phenomenon for applications for computer security is *concept drift* [33]. Put simply, this occurs when examples have certain labels for periods of time, and then have different labels for other periods of time. For instance, when running experiments, a researcher's normal behavior might be characterized by multiple jobs requiring massive amounts of CPU time and disk access. However, when the same researcher is writing a paper describing those experiments and results, then normal behavior might be defined by relatively light usage. An example of low usage would be abnormal for the researcher during the experiment phase, but would be considered normal during the writing phase.

Concept drift can occur quickly or gradually and on different time scales. Such change could be apparent in combinations of data sources: in the network traffic, in the machine's audit metrics, in the commands users execute, or in the dynamics of their keystrokes.

Researchers have developed several algorithms for tracking concept drift [33–38], but only a few methods have been developed or evaluated for problems in computer security (e.g., [39]). All of the methods have been based to some extent on traditional or classical algorithms, so in the sections that follow, we describe a representative set of these algorithms. The contributors to this volume use some of these algorithms, and they describe these and others in their respective chapters.

3.3.1 Instance-Based Learning

Instance-based learners [40] store examples as their concept description. Learning is simply storing these examples. When classifying an observation, in the simplest case, the performance element computes the distance between the observation and every stored example. It returns as its decision the class label of the closest example (i.e., the nearest neighbor). Implementations often use Euclidean distance for numeric attributes and tally mismatches for symbolic attributes. Variants of this method find the k closest instances (IBk) or the k nearest neighbors (k-NN), returning the class with the majority vote as the decision. For large sets of examples, performance can be expensive, but there are methods of indexing examples for faster matching [41]. Instance-based learning can be sensitive to irrelevant attributes, so one should apply a method for feature selection. Good practice also dictates mapping numeric attributes to the range $[0, 1]$ so attributes with large values do not dominate the distance calculation.

3.3.2 Naive Bayes

Naive Bayes stores as its concept description the prior probability of each class and the conditional probability of each attribute value given the class. The learning element estimates these probabilities from examples by simply counting frequencies of occurrence. The prior probability is the portion of examples from each class. The conditional probability is the frequency that attribute values occur given the class. Given an observation, the performance element operates under the assumption that attributes are conditionally independent and uses Bayes' rule to calculate the posterior probability of each class, returning as the decision the class label with the highest probability. For continuous attributes, implementations typically compute the mean and variance of such values for each class, and during performance, compute the probability of an attribute's value as given by the normal distribution [42]. However, researchers have shown that mapping continuous attributes to discrete intervals can improve performance. In spite of violations of the independence assumption [43] and sensitivity to irrelevant attributes [44], naive Bayes performs well on many domains [45].

3.3.3 Kernel Density Estimation

A kernel density estimator is similar to naive Bayes. (See [19] for a general discussion.) It stores the prior probabilities of each class, like naive Bayes, but also stores each example. During performance, assuming attributes are conditionally independent and normally distributed, it estimates the probability of a value given the class by averaging over Gaussian kernels centered on each stored example. Learning is therefore a simple matter of storing each example and computing the probability of each class. Experimental results

suggest that a kernel density estimator is better than a single Gaussian for handling continuous attributes [42].

3.3.4 Learning Coefficients of a Linear Function

For binary or numeric attributes, we can use as a model a linear function of the attribute values with the form

$$y = w_1 x_1 + \cdots + w_n x_n + b \,,$$

where $\mathbf{x}$ is the vector of attribute values, $\mathbf{w}$ is the vector of *weights* or *coefficients*, b is the *intercept*, and y is the output. Sometimes b is written as w_0 and called the *threshold* or *bias* [17]. This is similar to the "straight-line" model discussed in Sect. 3.2. Given an observation, $\mathbf{x}$, performance involves determining the sign of y. If y is positive, then the method predicts the positive class, and predicts the negative class otherwise. Learning is then a process of finding an appropriate intercept and set of weights.

Linear classifiers and linear discriminants have a long history. Fisher's linear discriminant [46] is a special case of regression with two regressors [19]. The perceptron algorithm is restricted to binary inputs and two-class problems, but is guaranteed to converge to a solution if one exists [47] (i.e., if the positive and negative examples are *linearly separable*; see Fig. 3.1b as an example). Such methods find a solution, but do not necessarily find the optimal solution, defined as being the function that, in an informal sense, perfectly divides or separates the positive and negative examples. Under some circumstances (e.g., inputs are numeric rather than binary), no algorithm can converge to an exact solution, but we can use methods, such as gradient descent, to find an approximate solution. Solving both these problems, support vector machines map examples to a (higher-dimensional) space where examples are linearly separable and then find the optimal solution [48]. See [17] for a more detailed discussion of linear classifiers.

3.3.5 Learning Decision Rules

A decision rule consists of an antecedent and a consequent. The antecedent is simply a conjunction of attribute tests, and the consequent consists of a class label. Performance entails determining if an observation's attribute values satisfy all of the rule's conditions. If so, then the decision is the class label in the rule's consequent. An observation may satisfy no rule, but if necessary, we can match flexibly by selecting the rule with the greatest number of conditions satisfied. One method of learning rules is to select an example from a class and generalize it as much as possible without intersecting examples from the other classes. The algorithm removes any examples from the class that satisfy the rule and continues until no examples remain. It repeats with each class in the set of examples. AQ19 [49] is an implementation of this specific-to-general

algorithm, while CN2 [50] is similar, but implements a general-to-specific algorithm. RIPPER [51] also forms rules, but by growing them by repeatedly adding conditions. Once formed, RIPPER prunes the rules to remove ineffective sequences of conditions. OneR forms a model using the single, most predictive attribute [52].

3.3.6 Learning Decision Trees

A decision tree is a rooted tree with internal nodes corresponding to attributes and leaf nodes corresponding to class labels. Internal nodes have a child node for each value its associated attribute takes. The learning element generates a tree recursively by selecting the attribute that best splits the examples into their proper classes, creating child nodes for each value of the selected attribute, and distributing the examples to these child nodes based on the values of the selected attribute. The algorithm then removes the selected attribute from further consideration and repeats for each child node until producing nodes containing examples of the same class. These methods handle continuous attributes by finding a threshold that best splits the examples into their respective classes. Overtraining can occur, meaning that trees perform well on training examples but perform poorly on unseen data, so as a post-processing step, implementations may apply a pruning algorithm to remove nodes that will likely lead to higher error. Performance is simply a traversal of the tree from the root to a leaf node guided by the attribute values present in an observation. The class label of that leaf node is the prediction for the observation. It is also possible to further process trees into decision rules [53]. C4.5 is the quintessential program for learning decision trees [53]. The release comes with C4.5-rules, which converts trees to rules. C5 and C5-rules are their respective commercial successors. J48 is an implementation of the C4.5 algorithm present in WEKA [24]. Decision stumps are one-level decision trees [54]. ID4 [33], ID5 [55], and ITI [56] are incremental algorithms for tree induction. VFDT is a very fast algorithm for inducing decision trees that is particularly suitable for data mining tasks, for it always grows trees from the leaf nodes and requires no tree restructuring [57].

3.3.7 Mining Association Rules

Motivated by market-basket analysis, where we must discover patterns in baskets of purchased items, association rules represent associations between sets of attribute values [58]. They have an antecedent and a consequent, like decision rules. However, algorithms forming association rules return an exhaustive set of rules. In contrast, algorithms for inducing decision rules typically produce a minimal set of rules. Indeed, as Witten and Frank describe, it is possible to generate association rules using an algorithm for inducing decision rules by running the algorithm on all combinations of attribute values [24, p. 104].

The Apriori algorithm [59] generates all association rules that satisfy three constraints: the number of attribute values to be used to form the rule,[2] the level of *support* the rule must have, and the level of *confidence* the rule must have. Support is the percentage of the examples the rule matches. Confidence is the percentage of the examples matching the antecedent that also match the consequent. For example, we may use the Apriori algorithm to generate all association rules consisting of five attribute values, 66% support, and 50% confidence. Because association rules can potentially consider all possible combinations of attribute values, the method is well suited for gaining insights into a set of examples.

3.4 Evaluating Models

When a learning or mining algorithm builds a model from a set of examples, the algorithm carrys out an inductive inference. That is, it reasons from examples to hypotheses, and such inference is uncertain.[3] At issue then is how to evaluate the "quality" of the models built by such algorithms.

We have only a finite set of examples, and we have presumably gathered as many examples as possible. Furthermore, some unknown process generated the examples we have. Let us call this unknown process the *true* or *target function*, *model*, or *concept*. The model generated from a set of examples is then an approximation to the true model. Ultimately, we want to determine how well the induced model approximates the true model.

Since we have only a finite set of examples, evaluation involves splitting the available examples into a *training set* and a *testing set*. Generally, we apply an algorithm to the examples in the training set and evaluate the resulting model using the examples in the test set. Since we know the labels of the examples in the test set, we can compute a variety of performance metrics. One such metric is simply the percentage of testing examples the model predicts correctly. This process estimates the true error of the model, which is the error we can expect when new observations arrive.

Researchers have devised a number of evaluation schemes, including hold-out, leave-one-out, cross-validation, and bootstrap. The scheme we choose is governed several factors including the number of examples available, the number and complexity of the algorithms we intend to evaluate, the amount of computational resources available, and how well the evaluation scheme approximates the true error. This last consideration is the subject of investigation (e.g., [62]).

[2] The set of such values is called an *item set*.

[3] Some forms of induction are certain. Michalski's *conclusive induction* [60] and Cohen's *summative induction* [61] are examples.

Hold-out

The hold-out method randomly divides a data set once into training and testing sets. We, of course, apply an algorithm to the training set and evaluate the resulting model on the testing set, computing a performance metric, such as percent correct. The proportion of examples in the training set varies, but proportions of 50%, 66%, 75%, and 90% are common.

Leave-one-out

The leave-one-out method involves leaving one example out of the set of examples, building a model with those remaining, and evaluating it using the held-out example. If classifying or detecting, the prediction will be either right or wrong. We then repeat the process for each example in the set. The overall accuracy is the accuracy averaged over the total number of runs or applications of the algorithm, which is equal to the number of examples in the data set.

Cross-validation

The cross-validation method involves partioning the examples randomly into n folds. (Ten is a fairly popular choice for n, but much depends on the number of examples available.) We use one partition as a testing set and use the remaining partitions to form a training set. As before, we apply an algorithm to the training set and evaluate the resulting model on the testing set, calculating percent correct. We repeat this process using each of the partitions as the testing set and using the remaining partitions to form a training set. The overall accuracy is the accuracy averaged over the number of runs, which is equivalent to the number of partitions. *Stratified* cross-validation involves creating partitions so that the number of examples of each class is proportional to the number in the original set of examples.

Bootstrap

The bootstrap method is a resampling technique that involves sampling with replacement from a set of n examples to produce n sampled sets, each of n examples [63]. We then use the n bootstrap samples to estimate some statistic, such as the population mean. We cannot use this method directly for evaluating learning and mining algorithms, for we must produce training and testing sets. The .632 bootstrap [64] and the .632+ bootstrap [62] are methods appropriate for evaluating learning and mining algorithms. These methods are too involved to discuss here, and they are often too computationally expensive to apply to large data sets. Nonetheless, research suggests that they are superior to other evaluation methods, such as cross-validation, for estimating the true error of a classifier [62, 65].

Common Measures of Performance

Once we have selected an algorithm and an evaluation methodology, we need to select a performance metric. In previous sections, we have used accuracy or percent correct. As we describe, it is not the only one.

Table 3.2. Quantities computed from a test set for a two-class problem

		Predicted	
		+	−
Actual	+	a	b
	−	c	d

For two-class problems, a test case will be either positive or negative. The performance element, when given a test case, will predict either correctly or incorrectly. This yields four quantities that we can compute by applying a model to a set of test cases, as shown in Table 3.2. For a set of test cases, let

- a be the number of times the model predicts positive when the example's label is positive,
- b be the number of times the model predicts negative when the example's label is positive,
- c be the number of times the model predicts positive when the example's label is negative,
- d be the number of times the model predicts negative when the example's label is negative.

Given these counts, we can define a variety of common performance metrics. For example, accuracy is the portion of the test examples that the model correctly predicts:

$$\frac{a+d}{a+b+c+d}.$$

Percent correct is simply accuracy expressed as a percentage. Other performance metrics include:

- Error rate, being the portion of the examples in the test set the model predicts incorrectly: $(b+c)/(a+b+c+d)$.
- True-positive rate (tp), hit rate, detect rate, or sensitivity, being the portion of the positive examples the model predicts correctly: $a/(a+b)$.
- True-negative rate (tn), correct-reject rate, or specificity, being the portion of the negative examples the model predicts correctly: $d/(c+d)$.
- False-negative rate or miss rate, being the portion of the positive examples the classifier predicts falsely as negative: $b/(a+b)$. Also $1-tp$.
- False-positive rate or false-alarm rate, being the portion of the negative examples the classifier predicted falsely as positive: $c/(c+d)$. Also $1-tn$.

Metrics common to document-retrieval tasks include:

- Recall (R), which is equivalent to the true-positive rate: $a/(a+b)$.
- Precision (P): $a/(a+c)$.
- F-measure:

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P + R} ,$$

  where β is a parameter that adjusts the relative importance of precision and recall.

Performance metrics not based on accuracy include the time an algorithm requires to produce a model from a set of examples, the time an algorithm requires to apply a model to a test set, and the size or complexity of the model an algorithm produces.

For numeric prediction tasks, a common performance measure is the mean squared error. Given n testing examples, for the ith example, let o_i be the model's prediction for the actual value, y_i. The mean squared error for the test sample is

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - o_i)^2 . \tag{3.1}$$

It is also common to take the square root of (3.1), which is called the root mean squared (RMS) error. Instead of the squared error, researchers also use the absolute error: $|y_i - o_i|$.

3.4.1 Problems with Simple Performance Measures

Researchers have made the case that evaluations using accuracy (or percent correct or error) are problematic [66]. One problem is that accuracy does not show how well a model predicts examples by class. For instance, if a testing set contains many more negative examples than positive examples, high accuracy could be due to the model's exceptional performance on the majority (i.e., negative) class. Indeed, examining the true-positive rate may reveal that the model performs quite poorly on positive cases.

Using the true-positive and true-negative rates (or the class accuracies) as performance measures is a simple solution, but simple measures of performance, such as accuracy and the true-positive rate, are problematic for other reasons. These simple measures are appropriate only when certain conditions hold, such as there being an equal number of examples in each class, the cost of those examples being the same, and the cost of making mistakes on each class being the same. For real-world problems, we will rarely be able to satisfy all of these constraints.

3.4.2 ROC Analysis

Researchers have used receiver operating characteristic (ROC) analysis [67] to mitigate some of the problems with simple measures of performance, discussed in the previous section. Briefly, ROC analysis provides a means of evaluating an algorithm over a range of possible operating scenarios. This is in contrast to traditional methods of evaluating performance in which we measure an algorithm's performance for only one operating scenario. Such scenarios stem from a combination of factors, including the number of examples of each class, the costs of mistakes being unequal, but unknown, and the examples not being a truly representative sample.

An ROC curve is simply a plot of a model's true-positive rate against its false-positive rate, as shown in Fig. 3.2. With a true-positive rate of unity and a false-positive rate of zero being perfect performance, we prefer curves that "push" toward the upper left of the ROC graph. We often use the area under the ROC curve (AUC) as a single measure of performance, a number that ranges between 0.5 and 1.0, with larger areas preferred.

We will review three methods for evaluating an algorithm under various operating scenarios. The first involves varying the decision threshold of the performance element, and the exact way of doing this is usually specific to the learning or mining method. For naive Bayes, for example, the performance element returns the class with the maximum posterior probability, which, for a two-class problem, is equivalent to a decision threshold of 0.5 (since probabilities must sum to unity). So, we can change the decision threshold of naive Bayes by predicting the positive class if its posterior probability is greater than or equal to, say, 0.4. By evaluating performance at a variety of thresholds, we generate a set of true-positive and false-positive rates, which we can then plot as an ROC curve, similar to the one in Fig. 3.2a. To compute the area under the curve, we can use the trapezoid rule, which entails summing the areas of the trapezoids formed by two adjacent points on the curve.

The second method entails using the model to *rate* the test cases, rather than to classify them. Let us again consider naive Bayes. For a two-class problem, we can use the posterior probability of the negative class as a case rating. Instead of indicating positive or negative, the rating indicates the probability that a case is negative. We can then use software, such as `labroc4` [68], to estimate an ROC curve from the case ratings. The software computes operating points consisting of true-positive and false-positive rates, which we can plot as an ROC curve, similar to the one in Fig. 3.2a. The program also estimates parameters of a parametric ROC curve, which is based on normal distributions, and we can use these parameters to plot a continuous ROC curve, similar to the one pictured in Fig. 3.2b. `labroc4` also computes the areas and standard errors of both the empirical and parametric curves.

The third method uses case ratings and the predicted labels directly. We begin by computing Δx by dividing one by the number of positive examples and Δy by dividing one by the number of negative examples. After sorting the

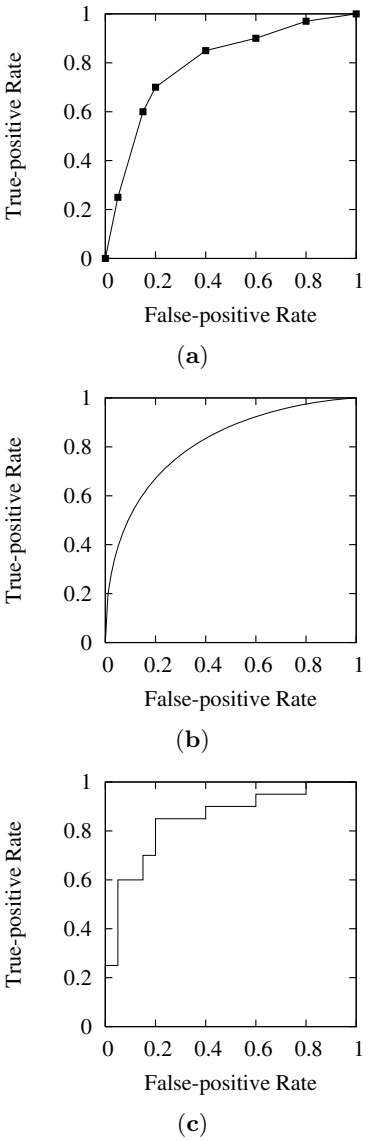


Fig. 3.2. Hypothetical ROC curves. (**a**) Generated from specific true-positive and false-positive rates. (**b**) Generated from parameters of a parametric curve. (**c**) Generated from sorted ratings and labels

predicted labels by their case rating, we start drawing an ROC curve from the lower-left corner of the ROC space, where the true-positive and false-positive rates are both zero. We process the sorted labels, and when we encounter a negative label, we draw a vertical line of length Δy, and when we encounter a positive label, we draw a horizontal line of length Δx. After processing all of the predicted labels in this fashion, we will produce an ROC curve similar to that pictured in Fig. 3.2c. We can compute the area under this curve by summing the areas of the rectangles forming the curve.

Most evaluations of an algorithm will yield a set of ROC curves. For example, ten-fold cross-validation will produce ten ROC curves, one for each fold. To obtain a single ROC for the experiment, we can average the true-positive and false-positive rates at each decision threshold and plot averaged rates. We can also sort all of the ratings and predicted labels from the folds, and plot the resulting curve. We can then compute the area under the final ROC curve directly from the final ROC curve or by averaging the areas from individual curves.

3.4.3 Principled Evaluations and Their Importance

Learning and mining algorithms carry out inductive inference on data sets to produce models. These data sets, regardless of their size, are often small samples of the space of possible examples. As a result, the models such algorithms produce are approximations. We often use such models to make critical decisions. Consequently, the importance of conducting principled evaluations of such algorithms cannot be overstated.

We described a few of the pre-processing operations that researchers and practitioners apply when transforming raw data into examples (see Sect. 3.2). When evaluating algorithms, it is critically important to apply these operations only to the training examples and not to the testing examples. We must apply such operations after creating training and testing sets. For example, selecting the most relevant attributes *before* dividing a set of examples into training and testing sets will invariably bias the evaluation of the resulting model. The outcome of such an evaluation will probably be that the model's accuracy is higher than is the case. We will not discover this fact until after we deploy the model and apply it to new observations. Depending on the domain, this could be catastrophic.

It is impossible to determine a priori which algorithm will perform the best on a given data set. Some algorithms work well across a large range of data sets, problems, or tasks, but we cannot conclude that they will perform well for any and all tasks. As a consequence, we must always evaluate as many algorithms as possible. We should select a representative set of algorithms that differ in their learning and performance elements, and their model representations. For instance, it would be better to evaluate three algorithms that produce different models than to evaluate three that all produce decision trees.

When evaluating several algorithms using, say, ten-fold cross-validation, there are many sources of variability. Each of the ten training sets is slightly different, so a given algorithm will produce slightly different models for each. Some algorithms are *unstable*, meaning that a slight change in the training set leads to a considerably different model [69]. Quantifying this variability is as important as quantifying accuracy. One can choose from a number of measures, such as variance, standard deviation, standard error, and confidence intervals. Ideally, we would also apply a hypothesis test to determine if results are significantly different [70, 71].

Because of such sources of variability, it is important to design an experiment to eliminate the sources we can, thereby reducing the error variance [72]. For example, whenever possible, it is best to apply all of the algorithms to the same training sets and evaluate all of the resulting models to the same testing sets. That is, whenever possible, it is better to use a *within-subjects design* rather than a *between-subjects design* [72].

Similarly, it is important to evaluate algorithms under the same experimental conditions. For instance, if we have many algorithms to evaluate, we might be tempted to evaluate one algorithm using cross-validation on one machine and evaluate another algorithm in the same manner on some other machine, but this should be avoided. Potentially, each algorithm will learn from different training examples, which introduces an unnecessary source of variability. It would be better to distribute the training and testing sets across the two machines and run both algorithms on each machine.

When evaluating a new algorithm, it is critically important to evaluate it across a range of appropriate tasks, and to compare, either directly or indirectly, to other algorithms designed for the task. Direct comparisons are best, because we evaluate the competing algorithm under exactly the same conditions as the new algorithm. When implementations are unavailable and too complex to implement, then we can compare the new method to published reports.

When evaluating an extension to an existing algorithm, it is common to conduct a *lesion study* by comparing the original algorithm to the enhanced version. Such an evaluation will show the effect of the modification and any trade-offs in performance. Furthermore, comparing the enhanced algorithm to other algorithms across a variety of appropriate problems will strengthen the evaluation.

3.5 Ensemble Methods and Sequence Learning

The preceding sections provide a survey of some of the fundamental topics in machine learning and data mining. Two topics that build upon these ideas are *ensemble methods* and *sequence learning*. Both are important for applications in computer security, and we briefly review them in the next two sections.

3.5.1 Ensemble Methods

Researchers have recently begun investigating *ensemble methods*, which use multiple models for prediction, rather than a single model. Critical to ensemble methods is a scheme for producing a set of different models. Indeed, virtually all algorithms will produce the same model when given the same training data. While there are many techniques for creating multiple models, such as stacking [73], arcing [69], and weighted majority [74], we will survey two popular methods: bagging and boosting.

Bagging involves producing a set of bootstrap samples from a training set, and then building a model from each sample [75]. To obtain a bootstrap sample, we simply pick examples randomly with replacement (meaning that a single example could be picked multiple times) until the bootstrap sample has the same number of examples as the training set. We then use an algorithm to build a model for each bootstrap sample. To classify an observation, we obtain a prediction from each model, and the final prediction is the class label with the majority of the predictions. For example, if we have eleven models and six predict malicious and five predict benign, then the final prediction is the majority: malicious. There have been several empirical studies suggesting that bagging improves the performance of a single model on a variety of tasks [76–79].

Boosting [80], which is a bit more complicated than bagging, also improves the performance of single-model methods [76–79, 81]. We will give an informal account of boosting here, but, for more detail, see Witten and Frank's description [24] or Freund and Schapire's original paper [82].

When applying an algorithm to a set of examples, some examples will be easy for it to learn, and others will be difficult. After building a model from a set of examples, because the model will perform well on some and not so well on others, we will have some insights into which examples were easy to learn and which were hard. This gives a way to weight the examples based on their degree of difficulty. Algorithms, such as C4.5 [53], can use these weights to produce a weighted model, which effectively focuses the algorithm on the hard examples (i.e., those with high weights).

Thus, boosting entails iteratively weighting examples and building a desired number of models. This produces a set of models, each with a weight based on the model's estimated accuracy. To classify an observation, boosting applies each of the models to the observation, yielding a set of predictions. Rather than returning the majority vote as the prediction, boosting predicts the weighted majority vote.

3.5.2 Sequence Learning

For many applications, including some in computer security, a single event or action is insufficient to identify some phenomena. For example, A occurring by itself is not anomalous and B occurring by itself is, likewise, not anomalous.

However, if A occurs and then B immediately occurs, then this sequence of actions might suggest malicious behavior. Individual commands or system calls may not be sufficient for predicting, say, an intrusion. Detecting an intrusion may require examining sequences of commands or sequences of system calls.

There are several approaches to *supervised sequence learning*; we will examine two: sliding windows and hidden Markov models. (See [83] for a review.) With the first method, we use a moving window to transform a sequence into a set of examples. Assume we have the malicious sequence `abcb` and a window of size two. We then create an example for each subsequence of size two, giving the three examples: ⟨ `a`, `b`, `malicious` ⟩, ⟨ `b`, `c`, `malicious` ⟩, and ⟨ `c`, `b`, `malicious` ⟩. After doing the same to benign sequences, we can apply many of the methods discussed in Sect. 3.3.

Another way to approach this problem is using a hidden Markov model. A *Markov model* consists of a set of states and a set of transition probabilities [17]. As time advances, the model transitions from state to state, as dictated by the transition probabilities. If the current state is always sufficient to determine the next state, then the model is a *first-order* Markov model, and this is known as the *Markov assumption*, thereby defining a *memoryless process*.

All of the states in this model are *observable*, but for some systems we need to model, all could be *hidden*, or only some states could be hidden (i.e., *partially observable*). If all of the states are hidden, then the only way to observe a system's behavior is through its outputs. And so a first-order hidden Markov model (HMM) consists of a set of states, a set of transition probabilities, a set of output symbols, and a set of output probabilities. Now, as the model transitions between states, we cannot observe the actual states, because they are hidden, but we can observe the output produced by it being in these states.

There are three ways we can use HMMs [17]. First, given training data, we can learn or estimate the transition and output probabilities. Second, given a sequence of output symbols, we can calculate its probability. Third, given a sequence of output symbols, we can calculate the probability of the model being in each of the hidden states.

Consider the following simplified scenario: We want to build a model to classify a user's computing activity based on command sequences. A sequence of UNIX commands such as ⟨`vi`, `make`, `gcc`, `a.out`, `gdb`, `vi`⟩ might imply compiling activity, whereas ⟨`cp`, `tar`, `rm`⟩ might imply archiving activity. In this scenario, the HMM's output symbols are the commands, and the activities are the hidden states.

One way to proceed is to use the HMM to infer whether a sequence of commands is compiling, archiving, or malicious activity. This entails collecting sequences of commands, labeling them, and using a training algorithm, such as the forward-backward algorithm [84], to estimate the HMM's probabilities. During performance, given a sequence of commands, we can use the HMM to infer the most likely hidden state, which corresponds to the user's activity.

As stated previously, this scenario is a simplification, but it does illustrate one way of applying HMMs to a problem in computer security. In reality, it is often difficult to collect malicious sequences of commands, but we can also use HMMs to detect anomalous sequences of commands [85].

3.6 Implementations and Data Sets

Over the years, companies and researchers have developed numerous implementations of learning and mining algorithms. It is easy to find on the Internet implementations of both individual algorithms and collections of algorithms. Furthermore, the research community has diligently maintained publicly accessible data sets for a variety of domains.

KDnuggets™ (`http://www.kdnuggets.com`) is a Web portal for data mining, knowledge discovery, and related activities. There are links from this site to software (both commercial and free), data sets, courses, conferences, and other Web sites.

At the time of this writing, WEKA [24] is the most comprehensive, free collection of tools for machine learning and data mining (`http://www.cs.waikato.ac.nz/ml/weka/`). It is written in the Java programming language, so it is portable to most platforms. It is distributed as open-source software, so we can view the source code to learn about these methods, or we can modify the source code for our own special purposes.

The most comprehensive repositories for a variety of data sets are the Machine Learning Database Repository (`http://www.ics.uci.edu/~mlearn/MLRepository.html`) [86] and the Knowledge Discovery in Databases Archive (`http://kdd.ics.uci.edu/`) [87], both maintained by the Department of Information and Computer Science at the University of California, Irvine.

The most widely used computer security data set was collected as part of the DARPA Intrusion Detection Evaluation [88]. MIT Lincoln Labs collected and distributes a variety of data sets based on network traffic and audit logs (`http://www.ll.mit.edu/IST/ideval`). There are also data sets for specific scenarios, such as a distributed denial of service attack by a novice.

Portions of this data collection have been used to support numerous empirical studies. It was also used for the 1999 KDD Cup Competition, where contestants had to build the best performing model for classifying connections as either normal or one of four types of attack [89].

3.7 Further Reading

This chapter provides an overview of some of the basic concepts of machine learning and data mining. Readers may be able to proceed to the other chapters of this volume, or they may want to consult some of the cited books and articles.

For readers in this latter category, Witten and Frank's book [24] is an excellent next step, mainly because readers can download WEKA, which consists of free, open-source, high-quality Java implementations of an amazing number of algorithms for building models, selecting attributes, and other such operations. Examining and experimenting with this software is a great way to gain a better understanding of the practicalities of machine learning and data mining.

There are two books about data mining approaches to computer security complementary to this one. Mena [90] surveys several machine learning and data mining technologies and describes companies, software, and case studies of such technologies applied to a wide range of applications, including intrusion detection, fraud detection, and criminal profiling. Reference [91] is a collection of research articles, similar to the second part of this volume.

Readers with a concrete or practical understanding of such topics may want to investigate sources that discuss these issues more generally and more formally. Such references include those on pattern classification [17], statistical pattern recognition [18, 19], information retrieval [20], machine learning [2, 21, 22], data mining [3, 23], and statistical learning [25]. Books dealing solely with kernel methods and support vector machines include [92] and [93]. In this chapter, there was little mention of neural networks, which certainly have a role to play. Linear classifiers and the perceptron algorithm form the basis for more powerful models, such as feed-forward neural networks trained with the back-propagation algorithm. Reference [2] contains a chapter on the subject, while [94] is a more thorough and formal treatment.

3.8 Concluding Remarks

In this chapter, we surveyed some of the basic concepts of machine learning and data mining. It was an attempt to introduce some of the topics discussed in the research contributions, and as such, it – hopefully – will give readers new to these techniques a better ability to read the research contributions of this volume. If not, then perhaps it has provided references for further study.

Part II

Research Contributions

4

Learning to Detect Malicious Executables*

Jeremy Z. Kolter and Marcus A. Maloof

4.1 Introduction

Malicious code is "any code added, changed, or removed from a software system to intentionally cause harm or subvert the system's intended function" [95, p. 33]. Such software has been used to compromise computer systems, to destroy their information, and to render them useless. It has also been used to gather information, such as passwords and credit card numbers, and to distribute information, such as pornography, all without the knowledge of the system's users. As more novice users obtain sophisticated computers with high-speed connections to the Internet, the potential for further abuse is great.

Malicious executables generally fall into three categories based on their transport mechanism: viruses, worms, and Trojan horses. Viruses inject malicious code into existing programs, which become "infected" and, in turn, propagate the virus to other programs when executed. Viruses come in two forms, either as an infected executable or as a virus loader, a small program that only inserts viral code. Worms, in contrast, are self-contained programs that spread over a network, usually by exploiting vulnerabilities in the software running on the networked computers. Finally, Trojan horses masquerade as benign programs, but perform malicious functions. Malicious executables

do not always fit neatly into these categories and can exhibit combinations of behaviors.

Excellent technology exists for detecting known malicious executables. Software for virus detection has been quite successful, and programs such as McAfee Virus Scan and Norton AntiVirus are ubiquitous. Indeed, Dell recommends Norton AntiVirus for all of its new systems. Although these products use the word *virus* in their names, they also detect worms and Trojan horses.

These programs search executable code for known patterns, and this method is problematic. One shortcoming is that we must obtain a copy of a malicious program before extracting the pattern necessary for its detection. Obtaining copies of new or unknown malicious programs usually entails them infecting or attacking a computer system.

To complicate matters, writing malicious programs has become easier: There are virus kits freely available on the Internet. Individuals who write viruses have become more sophisticated, often using mechanisms to change or obfuscate their code to produce so-called *polymorphic viruses* [96, p. 339]. Indeed, researchers have recently discovered that simple obfuscation techniques foil commercial programs for virus detection [97]. These challenges have prompted some researchers to investigate learning methods for detecting new or unknown viruses, and more generally, malicious code.

Our efforts to address this problem have resulted in a fielded application, built using techniques from machine learning [2] and data mining [3]. The Malicious Executable Classification System (MECS) currently detects unknown malicious executables "in the wild," that is, without removing any obfuscation. To date, we have gathered 1,971 system and non-system executables, which we will refer to as "benign" executables, and 1,651 malicious executables with a variety of transport mechanisms and payloads (e.g., key-loggers and backdoors). Although all were for the Windows operating system, it is important to note that our approach is not restricted to this operating system.

We extracted byte sequences from the executables, converted these into n-grams, and constructed several classifiers: IBk, TFIDF, naive Bayes, support vector machines (SVMs), decision trees, boosted naive Bayes, boosted SVMs, and boosted decision trees. In this domain, there is an issue of unequal but unknown costs of misclassification error, so we evaluated the methods using receiver operating characteristic (ROC) analysis [67], using area under the ROC curve as the performance metric. Ultimately, boosted decision trees outperformed all other methods with an area under the curve of 0.996.

We delivered MECS to the MITRE Corporation, the sponsors of this project, as a research prototype. Users interact with MECS through a command line. They can add new executables to the collection, update learned models, display ROC curves, and produce a single classifier at a specific operating point on a selected ROC curve.

With this chapter, we make three main contributions. We show how established methods for text classification apply to executables. We present empirical results from an extensive study of inductive methods for detecting

malicious executables in the wild. We report on a fielded application developed using machine learning and data mining.

In the three sections that follow, we describe related work, our data collection, and the methods we applied. Then, in Sect. 4.6, we present empirical results, and in Sect. 4.7, we discuss these results and other approaches.

4.2 Related Work

There have been few attempts to use machine learning and data mining for the purpose of identifying new or unknown malicious code. These have concentrated mostly on PC viruses, thereby limiting the utility of such approaches to a particular type of malicious code and to computer systems running Microsoft's Windows operating system. Such efforts are of little direct use for computers running the UNIX operating system, for which viruses pose little threat. However, the methods proposed are general, meaning that they could be applied to malicious code for any platform, and presently, malicious code for the Windows operating system poses the greatest threat.

In an early attempt, Lo et al. [98] conducted an analysis of several programs – evidently by hand – and identified *telltale signs*, which they subsequently used to filter new programs. While we appreciate their attempt to extract patterns or signatures for identifying any class of malicious code, they presented no experimental results suggesting how general or extensible their approach might be. Researchers at IBM's T.J. Watson Research Center have investigated neural networks for virus detection [99], and have incorporated a similar approach for detecting boot-sector viruses into IBM's Anti-Virus software [100].

Table 4.1. Results from the study conducted by Schultz et al. [101]

Method	TP Rate	FP Rate	Accuracy (%)
Signature + `hexdump`	0.34	0.00	49.31
RIPPER + DLLs used	0.58	0.09	83.61
RIPPER + DLL function used	0.71	0.08	89.36
RIPPER + DLL function counts	0.53	0.05	89.07
Naive Bayes + `strings`	0.97	0.04	97.11
Voting Naive Bayes + `hexdump`	0.98	0.06	96.88

More recently, instead of focusing on boot-sector viruses, Schultz et al. [101] used data mining methods, such as naive Bayes, to detect malicious code. The authors collected 4,301 programs for the Windows operating system and used McAfee Virus Scan to label each as either malicious or benign. There were 3,301 programs in the former category and 1,000 in the latter. Of the malicious programs, 95% were viruses and 5% were Trojan horses.

Furthermore, 38 of the malicious programs and 206 of the benign programs were in the Windows Portable Executable (PE) format.

For feature extraction, the authors used three methods: binary profiling, string sequences, and so-called *hex dumps*. The authors applied the first method to the smaller collection of 244 executables in the Windows PE format and applied the second and third methods to the full collection.

The first method extracted three types of resource information from the Windows executables: (1) a list of Dynamically Linked Libraries (DLLs), (2) function calls from the DLLs, and (3) the number of different system calls from within each DLL. For each resource type, the authors constructed binary feature vectors based on the presence or absence of each in the executable. For example, if the collection of executables used ten DLLs, then they would characterize each as a binary vector of size ten. If a given executable used a DLL, then they would set the entry in the executable's vector corresponding to that DLL to one. This processing resulted in $2,229$ binary features, and in a similar manner, they encoded function calls and their number, resulting in 30 integer features.

The second method of feature extraction used the UNIX `strings` command, which shows the printable strings in an object or binary file. The authors formed training examples by treating the strings as binary attributes that were either present in or absent from a given executable.

The third method used the `hexdump` utility [102], which is similar to the UNIX octal dump (`od -x`) command. This printed the contents of the executable file as a sequence of hexadecimal numbers. As with the printable strings, the authors used two-byte words as binary attributes that were either present or absent.

After processing the executables using these three methods, the authors paired each extraction method with a single learning algorithm. Using five-fold cross-validation, they used RIPPER [51] to learn rules from the training set produced by binary profiling. They used naive Bayes to estimate probabilities from the training set produced by the `strings` command. Finally, they used an ensemble of six naive-Bayesian classifiers on the `hexdump` data by training each on one-sixth of the lines in the output file. The first learned from lines 1, 6, 12, ...; the second, from lines 2, 7, 13, ...; and so on. As a baseline method, the authors implemented a signature-based scanner by using byte sequences unique to the malicious executables.

The authors concluded, based on true-positive (TP) rates, that the voting naive Bayesian classifier outperformed all other methods, which appear with false-positive (FP) rates and accuracies in Table 4.1. The authors also presented ROC curves [67], but did not report the areas under these curves. Nonetheless, the curve for the single naive Bayesian classifier appears to dominate that of the voting naive Bayesian classifier in most of the ROC space, suggesting that the best performing method was actually naive Bayes trained with strings.

However, as the authors discuss, one must question the stability of DLL names, function names, and string features. For instance, one may be able to compile a source program using another compiler to produce an executable different enough to avoid detection. Programmers often use methods to obfuscate their code, so a list of DLLs or function names may not be available.

The authors paired each feature extraction method with a learning method, and as a result, RIPPER was trained on a much smaller collection of executables than were naive Bayes and the ensemble of naive-Bayesian classifiers. Although results were generally good, it would have been interesting to know how the learning methods performed on all data sets. It would have also been interesting to know if combining all features (i.e., strings, bytes, functions) into a single training example and then selecting the most relevant would have improved the performance of the methods.

There are other methods of guarding against malicious code, such as *object reconciliation* [96, p. 370], which involves comparing current files and directories to past copies; one can also compare cryptographic hashes. One can also audit running programs [103] and statically analyze executables using predefined malicious patterns [97]. These approaches are not based on data mining, although one could imagine the role such techniques might play.

Researchers have also investigated classification methods for the determination of software authorship. Most notorious in the field of authorship are the efforts to determine whether Sir Frances Bacon wrote works attributed to Shakespeare [104], or who wrote the twelve disputed Federalist Papers, Hamilton or Madison [105]. Recently, similar techniques have been used in the relatively new field of *software forensics* to determine program authorship [106]. Gray et al. [107] wrote a position paper on the subject of authorship, whereas Krsul [108] conducted an empirical study by gathering code from programmers of varying skill, extracting software metrics, and determining authorship using discriminant analysis. There are also relevant results published in the literature pertaining to the plagiarism of programs [109, 110], which we will not survey here.

Krsul [108] collected 88 programs written in the C programming language from 29 programmers at the undergraduate, graduate, and faculty levels. He then extracted 18 layout metrics (e.g., indentation of closing curly brackets), 15 style metrics (e.g., mean line length), and 19 structure metrics (e.g., percentage of `int` function definitions). On average, Krsul determined correct authorship 73% of the time. Interestingly, of the 17 most experienced programmers, he was able to determine authorship 100% of the time. The least experienced programmers were the most difficult to classify, presumably because they had not settled into a consistent style. Indeed, they "were surprised to find that one [programmer] had varied his programming style considerably from program to program in a period of only two months" [111, §5.1].

While interesting, it is unclear how much confidence we should have in these results. Krsul [108] used 52 features and only one or two examples for each of the 20 classes (i.e., the authors). This seems underconstrained, espe-

cially when rules of thumb suggest that one needs ten times more examples than features [112]. On the other hand, it may also suggest that one simply needs to be clever about what constitutes an example. For instance, one could presumably use functions as examples rather than programs, but for the task of determining authorship of malicious programs, it is unclear whether such data would be possible to collect or if it even exists. Fortunately, as we discuss in the next section, a lack of data was not a problem for our project.

4.3 Data Collection

As stated previously, the data for our study consisted of 1,971 benign executables and 1,651 malicious executables. All were in the Windows PE format. We obtained benign executables from all folders of machines running the Windows 2000 and XP operating systems. We gathered additional applications from SourceForge (`http://sourceforge.net`).

We obtained viruses, worms, and Trojan horses from the Web site VX Heavens (`http://vx.netlux.org`) and from computer-forensic experts at the MITRE Corporation, the sponsors of this project. Some executables were obfuscated with compression, encryption, or both; some were not, but we were not informed which were and which were not. For one collection, a commercial product for detecting viruses failed to identify 18 of the 114 malicious executables. Note that for viruses, we examined only the loader programs; we did not include infected executables in our study.

We used the `hexdump` utility [102] to convert each executable to hexadecimal codes in an ASCII format. We then produced n-grams, by combining each four-byte sequence into a single term. For instance, for the byte sequence `ff 00 ab 3e 12 b3`, the corresponding n-grams would be `ff00ab3e`, `00ab3e12`, and `ab3e12b3`. This processing resulted in 255,904,403 distinct n-grams. One could also compute n-grams from words, something we explored and discuss further in Sect. 4.6.1. Using the n-grams from all of the executables, we applied techniques from information retrieval and text classification, which we discuss further in the next section.

4.4 Classification Methodology

Our overall approach drew techniques from information retrieval (e.g., [20]) and from text classification (e.g., [113, 114]). We used the n-grams extracted from the executables to form training examples by viewing each n-gram as a binary attribute that is either present in (i.e., 1) or absent from (i.e., 0) the executable. We selected the most relevant attributes (i.e., n-grams) by computing the *information gain* (IG) for each:

$$IG(j) = \sum_{v_j \in \{0,1\}} \sum_{C \in \{C_i\}} P(v_j, C) \log \frac{P(v_j, C)}{P(v_j)P(C)} ,$$

where C is the class, v_j is the value of the jth attribute, $P(v_j, C)$ is the proportion that the jth attribute has the value v_j in the class C_i, $P(v_j)$ is the proportion that the jth n-gram takes the value v_j in the training data, and $P(C)$ is the proportion of the training data belonging to the class C. This measure is also called *average mutual information* [115].

We then selected the top 500 n-grams, a quantity we determined through pilot studies (see Sect. 4.6.1), and applied several learning methods, most of which are implemented in WEKA [24]: IBk, TFIDF, naive Bayes, a support vector machine (SVM), and a decision tree. We also "boosted" the last three of these learners, and we discuss each of these methods in the following sections.

4.4.1 Instance-Based Learner

One of the simplest learning methods is the instance-based (IB) learner [40]. Its concept description is a collection of training examples or instances. Learning, therefore, is the addition of new examples to the collection. To classify an unknown instance, the performance element finds the example in the collection most similar to the unknown and returns the example's class label as its prediction for the unknown. For binary attributes, such as ours, a convenient measure of similarity is the number of values two instances have in common. Variants of this method, such as IBk, find the k most similar instances and return the majority vote of their class labels as the prediction. Values for k are typically odd to prevent ties. Such methods are also known as *nearest neighbor* and k-*nearest neighbors*.

4.4.2 The TFIDF Classifier

For the TFIDF classifier, we followed a classical approach from information retrieval [20]. We used the *vector space model*, which entails assigning to each executable (i.e., document) a vector of size equal to the total number of distinct n-grams (i.e., terms) in the collection. The components of each vector were weights of the top n-grams present in the executable. For the jth n-gram of the ith executable, the method computes the weight w_{ij}, defined as

$$w_{ij} = tf_{ij} \times idf_j \ ,$$

where tf_{ij} (i.e., term frequency) is the number of times the ith n-gram appears in the jth executable and $idf_j = \log \frac{d}{df_j}$ (i.e., the inverse document frequency), where d is the total number of executables and df_j is the number of executables that contain the jth n-gram. It is important to note that this classifier was the only one that used continuous attribute values; all others used binary attribute values.

To classify an unknown instance, the method uses the top n-grams from the executable, as described previously, to form a vector, $\mathbf{u}$, the components of which are each n-gram's inverse document frequency (i.e., $u_j = idf_j$).

Once formed, the classifier computes a similarity coefficient (SC) between the vector for the unknown executable and each vector for the executables in the collection using the *cosine similarity measure*:

$$SC(\mathbf{u}, \mathbf{w}_i) = \frac{\sum_{j=1}^{k} u_j w_{ij}}{\sqrt{\sum_{j=1}^{k} u_j^2 \cdot \sum_{j=1}^{k} w_{ij}^2}} ,$$

where $\mathbf{u}$ is the vector for the unknown executable, $\mathbf{w}_i$ is the vector for the ith executable, and k is the number of distinct n-grams in the collection.

After selecting the top five closest matches to the unknown, the method takes a weighted majority vote of the executable labels, and returns the class with the least weight as the prediction. It uses the cosine measure as the weight. Since we evaluated the methods using ROC analysis [67], which requires *case ratings*, we summed the cosine measures of the negative executables in the top five, subtracted the sum of the cosine measures of the positive executables, and used the resulting value as the rating. In the following discussion, we will refer to this method as the TFIDF classifier.

4.4.3 Naive Bayes

Naive Bayes is a probabilistic method that has a long history in information retrieval and text classification [116]. It stores as its concept description the prior probability of each class, $P(C_i)$, and the conditional probability of each attribute value given the class, $P(v_j|C_i)$. It estimates these quantities by counting in training data the frequency of occurrence of the classes and of the attribute values for each class. Then, assuming conditional independence of the attributes, it uses Bayes' rule to compute the posterior probability of each class given an unknown instance, returning as its prediction the class with the highest such value:

$$C = \operatorname*{argmax}_{C_i} P(C_i) \prod_j P(v_j|C_i) .$$

For ROC analysis, we used the posterior probability of the negative class as the case rating.

4.4.4 Support Vector Machines

Support vector machines (SVMs) [48] have performed well on traditional text classification tasks [113, 114, 117], and performed well on ours. The method produces a linear classifier, so its concept description is a vector of weights, $\mathbf{w}$, and an intercept or a threshold, b. However, unlike other linear classifiers, such as Fisher's, SVMs use a kernel function to map training data into a higher dimensioned space so that the problem is linearly separable. It then uses quadratic programming to set $\mathbf{w}$ and b such that the hyperplane's margin

is optimal, meaning that the distance is maximal from the hyperplane to the closest examples of the positive and negative classes. During performance, the method predicts the positive class if $\langle \mathbf{w} \cdot \mathbf{x} \rangle - b > 0$ and predicts the negative class otherwise. Quadratic programming can be expensive for large problems, but sequential minimal optimization (SMO) is a fast, efficient algorithm for training SVMs [118] and is the one implemented in WEKA [24]. During performance, this implementation computes the probability of each class [119], and for ROC analysis, we used probability of the negative class as the rating.

4.4.5 Decision Trees

A decision tree is a tree with internal nodes corresponding to attributes and leaf nodes corresponding to class labels. For symbolic attributes, branches leading to children correspond to the attribute's values. The performance element uses the attributes and their values of an instance to traverse the tree from the root to a leaf. It predicts the class label of the leaf node. The learning element builds such a tree by selecting the attribute that best splits the training examples into their proper classes. It creates a node, branches, and children for the attribute and its values, removes the attribute from further consideration, and distributes the examples to the appropriate child node. This process repeats recursively until a node contains examples of the same class, at which point, it stores the class label. Most implementations use the *gain ratio* for attribute selection [53], a measure based on the information gain. In an effort to reduce overtraining, most implementations also prune induced decision trees by removing subtrees that are likely to perform poorly on test data. WEKA's J48 [24] is an implementation of the ubiquitous C4.5 algorithm [53]. During performance, J48 assigns weights to each class, and we used the weight of the negative class as the case rating.

4.4.6 Boosted Classifiers

Boosting [82] is a method for combining multiple classifiers. Researchers have shown that *ensemble methods* often improve performance over single classifiers [79, 120]. Boosting produces a set of weighted models by iteratively learning a model from a weighted data set, evaluating it, and reweighting the data set based on the model's performance. During performance, the method uses the set of models and their weights to predict the class with the highest weight. We used the AdaBoost.M1 algorithm [82] implemented in WEKA [24] to boost SVMs, J48, and naive Bayes. As the case rating, we used the weight of the negative class. Note that we did not apply AdaBoost.M1 to IBk because of the high computational expense.

4.5 Experimental Design

To evaluate the approaches and methods, we used stratified ten-fold cross-validation. That is, we randomly partitioned the executables into ten disjoint sets of equal size, selected one as a testing set, and combined the remaining nine to form a training set. We conducted ten such runs using each partition as the testing set.

For each run, we extracted n-grams from the executables in the training and testing sets. We selected the most relevant features from the training data, applied each classification method, and used the resulting classifier to rate the examples in the test set.

To conduct ROC analysis [67], for each method, we pooled the ratings from the iterations of cross-validation, and used `labroc4` [68] to produce an empirical ROC curve and to compute its area and the standard error of the area. With the standard error, we computed 95% confidence intervals [67]. We present and discuss these results in the next section.

4.6 Experimental Results

We conducted three experimental studies using our data collection and experimental methodology, described previously. We first conducted pilot studies to determine the size of words and n-grams, and the number of n-grams relevant for prediction. Once determined, we applied all of the classification methods to a small collection of executables. We then applied the methodology to a larger collection of executables, all of which we describe in the next three sections.

4.6.1 Pilot Studies

We conducted pilot studies to determine three quantities: the size of n-grams, the size of words, and the number of selected features. Unfortunately, due to computational overhead, we were unable to evaluate exhaustively all methods for all settings of these parameters, so we assumed that the number of features would most affect performance, and began our investigation accordingly.

Using the experimental methodology described previously, we extracted bytes from 476 malicious executables and 561 benign executables and produced n-grams, for $n = 4$. (This smaller set of executables constituted our initial collection, which we later supplemented.) We then selected the best 10, 20, ..., 100, 200, ..., 1,000, 2,000, ..., 10,000 n-grams, and evaluated the performance of an SVM, boosted SVMs, naive Bayes, J48, and boosted J48. Selecting 500 n-grams produced the best results.

We fixed the number of n-grams at 500, and varied n, the size of the n-grams. We evaluated the same methods for $n = 1, 2, \ldots, 10$, and $n = 4$ produced the best results. We also varied the size of the words (one byte, two

bytes, etc.), and results suggested that single bytes produced better results than did multiple bytes.

And so by selecting the top 500 n-grams of size four produced from single bytes, we evaluated all of the classification methods on this small collection of executables. We describe the results of this experiment in the next section.

4.6.2 Experiment with a Small Collection

Processing the small collection of executables produced 68,744,909 distinct n-grams. Following our experimental methodology, we used ten-fold cross-validation, selected the 500 best n-grams, and applied all of the classification methods. The ROC curves for these methods are in Fig. 4.1, while the areas under these curves with 95% confidence intervals are in Table 4.2.

Table 4.2. Results for detecting malicious executables in the small collection. Areas under the ROC curve (AUC) with 95% confidence intervals

Method	AUC
Naive Bayes	0.8850±0.0247
J48	0.9235±0.0204
Boosted Naive Bayes	0.9461±0.0170
TFIDF	0.9666±0.0133
SVM	0.9671±0.0133
IBk, $k = 5$	0.9695±0.0129
Boosted SVM	0.9744±0.0118
Boosted J48	0.9836±0.0095

As one can see, the boosted methods performed well, as did the instance-based learner and the support vector machine. Naive Bayes did not perform as well, and we discuss this further in Sect. 4.7.

4.6.3 Experiment with a Larger Collection

With success on a small collection, we turned our attention to evaluating the text-classification methods on a larger collection of executables. As mentioned previously, this collection consisted of 1,971 benign executables and 1,651 malicious executables, while processing resulted in over 255 million distinct n-grams of size four. We followed the same experimental methodology – selecting the 500 top n-grams for each run of ten-fold cross-validation, applying the classification methods, and plotting ROC curves.

Figure 4.2 shows the ROC curves for the various methods, while Table 4.3 presents the areas under these curves (AUC) with 95% confidence intervals. As one can see, boosted J48 outperformed all other methods. Other methods,

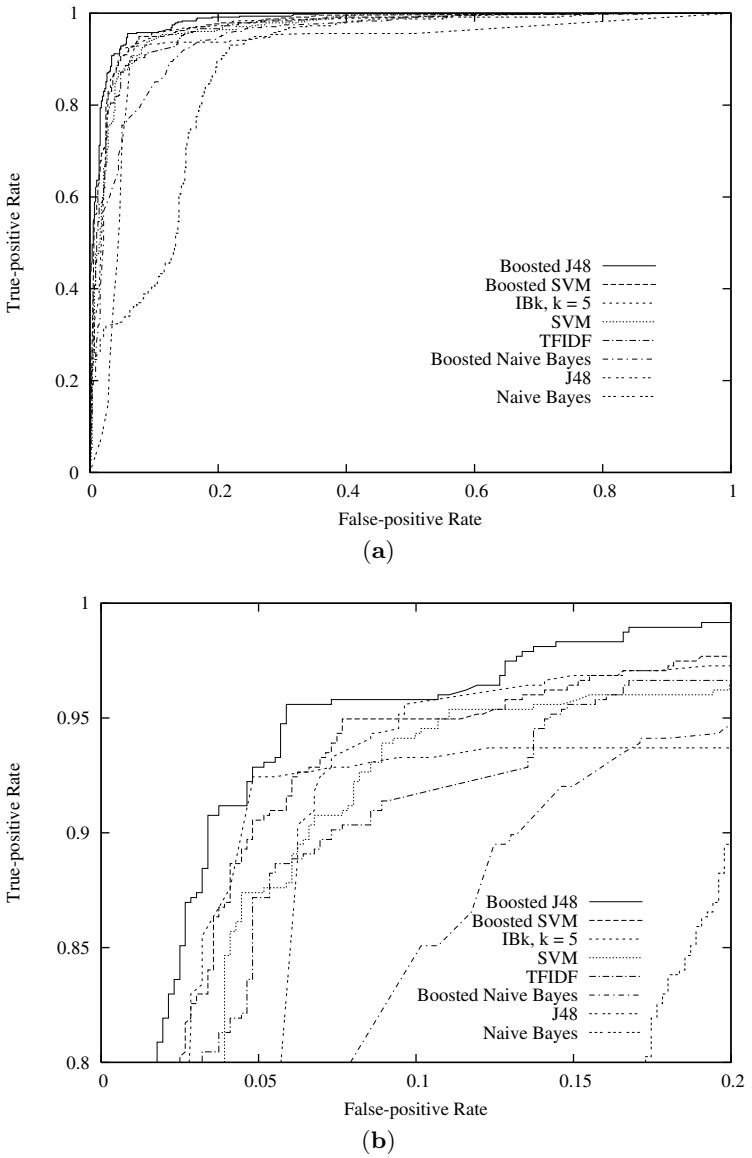


Fig. 4.1. ROC curves for detecting malicious executables in the small collection. (**a**) The entire ROC graph. (**b**) A magnification

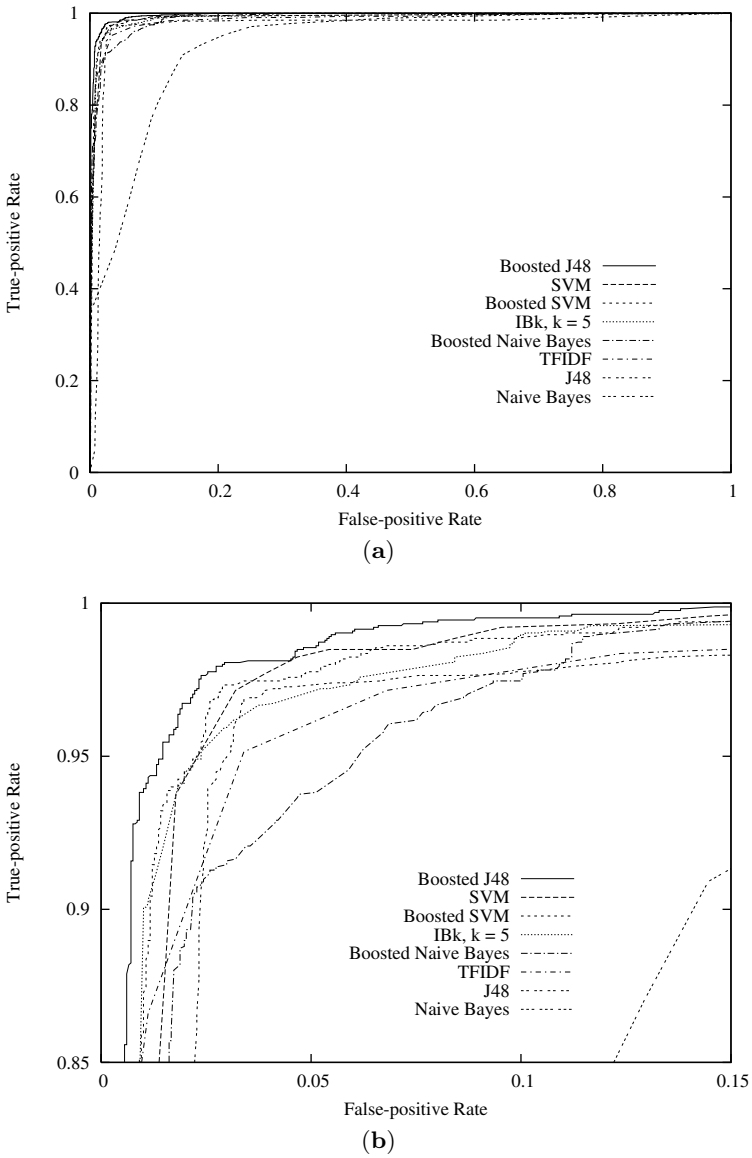


Fig. 4.2. ROC curves for detecting malicious executables in the larger collection. (**a**) The entire ROC graph. (**b**) A magnification

Table 4.3. Results for detecting malicious executables in the larger collection. Areas under the ROC curve (AUC) with 95% confidence intervals

Method	AUC
Naive Bayes	0.9366±0.0099
J48	0.9712±0.0067
TFIDF	0.9868±0.0045
Boosted Naive Bayes	0.9887±0.0042
IBk, $k = 5$	0.9899±0.0038
Boosted SVM	0.9903±0.0038
SVM	0.9925±0.0033
Boosted J48	0.9958±0.0024

such as IBk and boosted SVMs, performed comparably, but the ROC curve for boosted J48 dominated all others.

4.7 Discussion

To date, our results suggest that methods of text classification are appropriate for detecting malicious executables in the wild. Boosted classifiers, IBk, and a support vector machine performed exceptionally well given our current data collection. That the boosted classifiers generally outperformed single classifiers echos the conclusion of several empirical studies of boosting [69, 78, 79, 82], which suggest that boosting improves the performance of unstable classifiers, such as J48, by reducing their bias and variance [69, 78]. Boosting can adversely affect stable classifiers [78], such as naive Bayes, although in our study, boosting naive Bayes improved performance. Stability may also explain why the benefit of boosting SVMs was inconclusive in our study [69].

Our experimental results suggest that the methodology will scale to larger collections of executables. The larger collection in our study contained more than three times the number of executables in the smaller collection. Yet, as one can see in Tables 4.2 and 4.3, the absolute performance of all of the methods was better for the larger collection than for the smaller. The relative performance of the methods changed somewhat. For example, the SVM moved from fourth to second, displacing the boosted SVMs and IBk.

Visual inspection of the concept descriptions yielded interesting insights, but further work is required before these descriptions will be directly useful for computer-forensic experts. For instance, one short branch of a decision tree indicated that any executable with two PE headers is malicious. After analysis of our collection of malicious executables, we discovered two executables that contained another executable. While this was an interesting find, it represented an insignificantly small portion of the malicious programs.

Leaf nodes covering many executables were often at the end of long branches where one set of n-grams (i.e., byte codes) had to be present and another set had to be absent. Understanding why the absence of byte codes was important for an executable being malicious proved to be a difficult and often impossible task. It was fairly easy to establish that some n-grams in the decision tree were from string sequences and that some were from code sequences, but some were incomprehensible. For example, one n-gram appeared in 75% of the malicious executables, but it was not part of the executable format, it was not a string sequence, and it was not a code sequence. We have yet to determine its purpose.

Nonetheless, for the large collection of executables, the size of the decision trees averaged over 10 runs was about 90 nodes. No tree exceeded 103 nodes. The heights of the trees never exceeded 13 nodes, and subtrees of heights of 9 or less covered roughly 99.3% of the training examples. While these trees did not support a thorough forensic analysis, they did compactly encode a large number of benign and malicious executables.

To place our results in context with the study of Schultz et al. [101], they reported that the best performing approaches were naive Bayes trained on the printable strings from the program and an ensemble of naive-Bayesian classifiers trained on byte sequences. They did not report areas under their ROC curves, but visual inspection of these curves suggests that with the exception of naive Bayes, all of our methods outperformed their ensemble of naive-Bayesian classifiers. It also appears that our best performing methods, such as boosted J48, outperformed their naive Bayesian classifier trained with strings.

These differences in performance could be due to several factors. We analyzed different types of executables: Their collection consisted mostly of viruses, whereas ours contained viruses, worms, and Trojan horses. Ours consisted of executables in the Windows PE format; about 5.6% of theirs was in this format.

Our better results could be due to how we processed byte sequences. Schultz et al. [101] used non-overlapping two-byte sequences, whereas we used overlapping sequences of four bytes. With their approach it is possible that a useful feature (i.e., a predictive sequence of bytes) would be split across a boundary. This could explain why in their study string features appeared to be better than byte sequences, since extracted strings would not be broken apart. Their approach produced much less training data than did ours, but our application of feature selection reduced the original set of more than 255 million n-grams to a manageable 500.

Our results for naive Bayes were poor in comparison to theirs. We again attribute this to the differences in data extraction methods. Naive Bayes is well known to be sensitive to conditionally dependent attributes [45]. We used overlapping byte sequences as attributes, so there were many that were conditionally dependent. Indeed, after analyzing decision trees produced by J48, we found evidence that overlapping sequences were important for detection.

Specifically, some subpaths of these decision trees consisted of sequentially overlapping terms that together formed byte sequences relevant for prediction. Schultz et al.'s [101] extraction methods would not have produced conditionally dependent attributes to the same degree, if at all, since they used strings and non-overlapping byte sequences.

Regarding our experimental design, we decided to pool a method's ratings and produce a single ROC curve (see Sect. 4.5) because `labroc4` [68] occasionally could not fit an ROC curve to a method's ratings from a single fold of cross-validation (i.e., the ratings were degenerate). We also considered producing ROC convex hulls [32] and cost curves [121], but determined that traditional ROC analysis was appropriate for our results (e.g., the curve for boosted J48 dominated all other curves).

In our study, there was an issue of high computational overhead. Selecting features was expensive, and we had to resort to a disk-based implementation for computing information gain, which required a great deal of time and space to execute. However, once selected, WEKA's [24] Java implementations executed quickly on the training examples with their 500 binary attributes.

In terms of our approach, it is important to note that we have investigated other methods of feature extraction. For instance, we examined whether printable strings from the executable might be useful, but reasoned that subsets of n-grams would capture the same information. Indeed, after inspecting some of the decision trees that J48 produced, we found evidence suggesting that n-grams formed from strings were being used for detection. Nonetheless, if we later determine that explicitly representing printable strings is important, we can easily extend our representation to encode their presence or absence. On the other hand, as we stated previously, one must question the use of printable strings or DLL information since compression and other forms of obfuscation can mask this information.

We also considered using disassembled code as training data. For malicious executables using compression, being able to obtain a disassembly of critical sections of code may be a questionable assumption. Moreover, in pilot studies, a commercial product failed to disassemble some of our malicious executables.

We considered an approach that runs malicious executables in a sandbox and produces an audit of the machine instructions. Naturally, we would not be able to completely execute the program, but $10,000$ instructions may be sufficient to differentiate benign and malicious behavior. We have not pursued this idea because of a lack of auditing tools, the difficulty of handling large numbers of interactive programs, and the inability of detecting malicious behavior occurring near the end of sufficiently long programs.

There are at least two immediate commercial applications of our work. The first is a system, similar to MECS, for detecting malicious executables. Server software would need to store all known malicious executables and a comparably large set of benign executables. Due to the computational overhead of producing classifiers from such data, algorithms for computing information gain and for evaluating classification methods would have to be executed in

parallel. Client software would need to extract only the top n-grams from a given executable, apply a classifier, and predict. Updates to the classifier could be made remotely over the Internet. Since the best performing method may change with new training data, it will be critical for the server to evaluate a variety of methods and for the client to accommodate any of the potential classifiers. Used in conjunction with standard signature methods, these methods could provide better detection of malicious executables than is currently possible.

The second is a system oriented more toward computer-forensic experts. Even though work remains before decision trees could be used to analyze malicious executables, one could use IBk or the TFIDF classifier to retrieve known malicious executables similar to a newly discovered malicious executable. Based on the properties of the retrieved executables, such a system could give investigators insights into the new executable's function. However, it remains an open issue whether an executable's statistical properties are predictive of its functional characteristics, an issue we are currently investigating and one we discuss briefly in the concluding section.

4.8 Concluding Remarks

We considered the application of techniques from information retrieval and text classification to the problem of detecting unknown malicious executables in the wild. After evaluating a variety of classification methods, results suggest that boosted J48 produced the best classifier with an area under the ROC curve of 0.996. Our methodology resulted in a fielded application called MECS, the Malicious Executable Classification System, which we have delivered to the MITRE Corporation.

In future work, we plan to investigate a classification task in which methods determine the *functional characteristics* of malicious executables. Detecting malicious executables is important, but after detection, computer-forensic experts must determine the program's functional characteristics: Does it mass-mail? Does it modify system files? Does it open a backdoor? This will entail removing obfuscation, such as compression, if possible. Furthermore, most malicious executables perform multiple functions, so each training example will have multiple class labels, a problem that arises in bioinformatics and in document classification.

We anticipate that MECS, the Malicious Executable Classification System, is but one step in an overall scheme for detecting and classifying "malware." When combined with approaches that search for known signatures, we hope that such a strategy for detecting and classifying malicious executables will improve the security of computers. Indeed, the delivery of MECS to MITRE has provided computer-forensic experts with a valuable tool. We anticipate that pursuing the classification of executables into functional categories will provide another.

5

Data Mining Applied to Intrusion Detection: MITRE Experiences

Eric E. Bloedorn, Lisa M. Talbot, and David D. DeBarr

5.1 Introduction

As computers and the networks that connect them become increasingly important for the storage and retrieval of vital information, efforts to protect them become even more important. As illustrated by the Internet Storm Center (`http://isc.sans.org/`), systems connected to the Internet are subject to frequent probes and intrusion attempts; and organizations now deploy an array of measures to counteract these attacks. Intrusion detection serves a significant role in network defense; and a number of intrusion detection systems (IDSs) are now available, including Snort, RealSecure, and Dragon. Most IDSs rely on static signatures of attacks to separate them from normal traffic. However, new types of attacks are constantly being developed, and system administrators spend increasing effort to keep these signatures up to date.

MITRE's network security attempts to detect malicious intrusions by having all alerts reviewed by human analysts. Without automated support, however, this task has become increasingly difficult due to the volume of alerts. In one day, sensors can generate about 850,000 alerts, of which about 18,000 are labeled priority 1, the most severe. Attacks and probes can be frequent and noisy, generating thousands of alerts in a day. This can create a burden on the network security analyst, who must perform a kind of triage on the enormous flood of alerts.

Data mining can play a useful role in intrusion detection. In our research, we have pursued both classification and clustering approaches to see how they can be used to improve MITRE's network defense. Our goal in applying these methods is to reduce the number of false alarms presented to network analysts while maintaining the ability to detect unusual alert events. Toward this goal, we developed and continue to refine an alert aggregation procedure, a classifier for identifying mapping episodes, and a ranking process to identify those episodes that are more than simple probes and may disguise more malicious activity. To reduce false alarms, we developed a novel classification method by incrementally learning decision trees and decision rules. To detect anoma-

lous activity, we also developed a clustering approach based on the k-means algorithm.

With this combination of approaches (aggregation, false-alarm filtering, and then cluster-based anomaly detection), we have been able to significantly reduce the number of alerts that need to be reviewed by human analysts. For example, in one 5-day period, 71,094 priority 1 alerts were reduced to 1,011 (over a 98% reduction). This lets analysts spend more time reviewing unusual alerts and potentially more stealthy attacks, and thus increase system security.

In the process of this development, we addressed a number of problems, including a lack of labeled data for supervised classification, incidents with many alerts that drown out smaller classes, the need to identify behavior at an incident level whereas data is represented at an individual alert level, and difficulty of detecting both interesting anomalies as well as known attacks.

We have evaluated components of our approach on operational data collected from the MITRE network and on data used for the 1999 KDD Cup competition [89].

This paper gives a description of how MITRE is exploring use of data mining methods to improve the processing of network alerts. We present background on MITRE's network intrusion detection and related work. We then describe initial feature selection, aggregation, classification, and ranking processes. We then present a decision-tree and clustering approach for removing false alarms and identifying anomalies.

5.1.1 Related Work

Classification has been repeatedly applied to the problem of intrusion detection either to classify events into separate attack categories (e.g., the 1999 KDD Cup Competition) [89] or to characterize normal use of a network service [122]. In our application, the greatest need was to reduce the amount of false alarms requiring human review. The ability to further label events into specific attack categories is useful, but not essential. The problem of coming up with attack categories that have clear distinctions (and thus consistent labels) for a team of human analysts is nontrivial. For this reason, we built a false-alarm classifier rather than an n-way attack classifier, as has been done in previous applications of data mining to intrusion detection [123].

We are primarily seeking to improve the performance of an existing network defense system rather than trying to replace current intrusion detection methods with a data mining approach. While current signature-based commercial intrusion detection methods have limitations, they do still provide important services and represent significant investments by an organization. This required us to determine how data mining could be used in a complementary way to existing measures and led to our use of network alert data rather than raw connection data. In this sense, our work is similar to that of

Lee et al. [124] and Manganaris et al. [125]. Lee et al. used a classification algorithm called RIPPER to update the rules used by Network Flight Recorder (NFR), a commercial real-time network monitoring tool. Manganaris et al. used association rules from Intelligent Miner to reduce false alarms generated by NetRanger's sensors.

5.1.2 MITRE Intrusion Detection

MITRE's network infrastructure supports several thousand internal users, many customer collaborative networks, public Web servers, and multiple high-speed connections to the Internet. To protect these systems, MITRE incorporates a layered defense posture [126], which includes the deployment of filtering routers, state-aware firewalls, dedicated proxy servers, and intrusion detection, coupled with regular vulnerability assessments.

MITRE's approach to developing data mining processes to support intrusion detection is documented by Bloedorn et al. [127] and Skorupka et al. [128]. A general architecture illustrating our use of data mining is shown in Fig. 5.1.[1] Network traffic is analyzed by a variety of available signature-based intrusion detection sensors. All this sensor data is pulled periodically to a central server for processing, and loaded into a relational database. Analysts review incident data and individual alerts through the analyst interface. This interface is a Web server front end to the database that allows analysts to launch a number of predefined queries as well as free-form SQL queries.

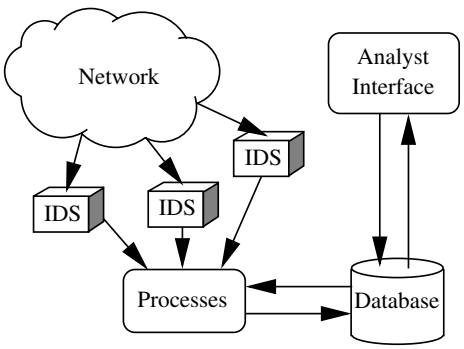


Fig. 5.1. MITRE sensors feed into overall intrusion detection system

The nerve center of MITRE's intrusion detection system is the processing. This processing includes data loading, aggregation, filtering, classification, and ranking to support data analysis. Details of some of the processes that are part

[1] For security reasons, we cannot reveal details of the MITRE network architecture. But we do employ a variety of sensor types located throughout the network both in front of and behind firewalls.

of MITRE's intrusion detection system are shown in Fig. 5.2. Events from the sensor data are filtered by a Heuristic for Obvious Mapping Episode Recognition (HOMER) before being passed on to the Back-End Analysis and Review of Tagging (BART), and then on to the classifier and clustering analyses. These data mining tools filter false alarms and identify anomalous behavior in the large amounts of remaining data and are described in more detail in later sections. All but the clustering tools are in daily operation.

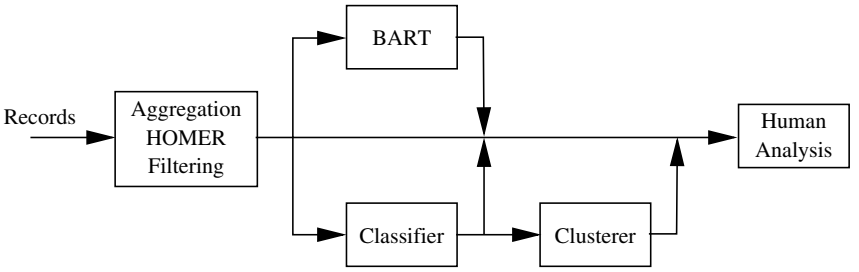


Fig. 5.2. Data mining processes in MITRE's intrusion detection system

The goal of this operational model is to detect malicious intrusions by presenting alerts to analysts for review. In order to make the best use of limited analyst time, we want to present the most likely alerts, aggregated by incident, and ranked according to potential severity.

5.2 Initial Feature Selection, Aggregation, Classification, and Ranking

Automated classification of false alarms would be of great value in the network security domain, but immediate application of decision-tree or rule-based methods was not possible. In order to use these methods, we needed labeled examples of true attacks and false alarms. But to get these labels would mean even more work for human analysts. Our objective was to classify with minimal human intervention. To address this problem, we developed a Web-based feedback system that made the labeling and aggregation of alerts into incidents as easy as possible. Using this feedback, we were able to collect over $10,000$ labeled examples per month of 7 different classes of incidents.

The Web-based analyst interface is useful, but it does not completely solve the problem of obtaining labeled examples because analysts were being flooded with alerts from only one type of network event. We discovered that around 90% of the alerts (around $600,000$ in June 2000) sent to the analysts were being generated by a small number of mapping attacks. In mapping attacks, an external host machine sends requests to multiple IP addresses in order to identify those machines running certain services. Obvious mapping episodes

can involve hundreds of machines over a period of just a few minutes. These mapping incidents were clogging our analysts' queues and preventing us from getting more labeled examples. Before advanced data mining approaches could be considered, this problem of mapping episodes (data aggregation) needed to be addressed.

Figure 5.3 shows how we use aggregation, mapping episode classification (HOMER), and tagging review (BART) in MITRE network operations. First, millions of alerts are aggregated by source IP address into thousands of episodes. HOMER classifies these episodes as a mapping if the number of destination IP addresses is greater than a threshold and then BART ranks the mapping episodes according to a computed severity. Those alerts deemed not to be part of a mapping episode are sent to a classifier to filter false alarms. Those records found to be benign by the classifier are sent to a cluster-based anomaly detector for additional testing. The following sections describe each of these processes in more detail.

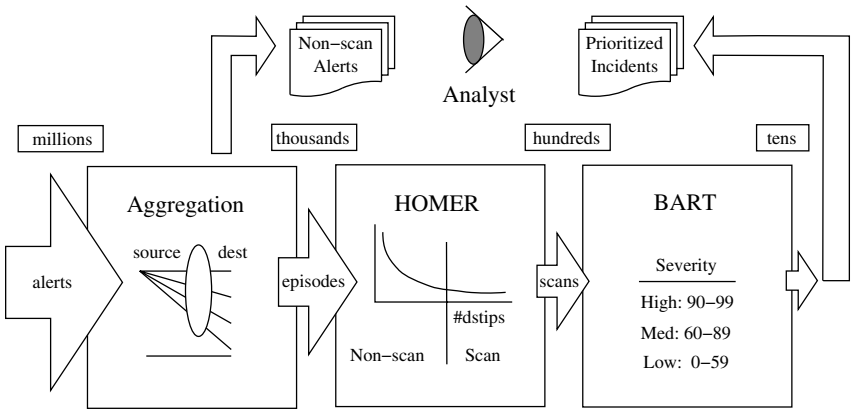


Fig. 5.3. Aggregation, mapping episode classifier (HOMER), and tagging review (BART) in MITRE's intrusion detection system

5.2.1 Feature Selection and Aggregation

A subtle obstacle to learning in the network security domain is the problem of learning descriptions of incidents from examples of alert events. Our data source for this work is a collection of reports (stored in the central server in Fig. 5.1) collected from a variety of network sensors. Each entry in that table is a record of a single alert from a single sensor and has attributes such as source IP address, destination IP address, and time. (A complete list of attributes appears in Tables 5.5–5.11 at the end of this chapter.) One difficulty with using this data directly for data mining is that the data are at the wrong level

of granularity. It would be more appropriate to learn at the level of an incident attack (i.e., a collection of alerts). But we cannot aggregate alerts into attacks until we know what kind of attack it is. An important question is, Which do we do first, the classification or the aggregation? We attempt to overcome this problem by providing context for each individual alert. To do this requires a number of additional attributes. We currently have 97 attributes beyond those given directly by the sensors. Examples of the additional attributes include the number of records with the same source IP and destination IP in the last day, and the number of records with the same source IP and destination port in the last day. Inclusion of these context features appears to allow us to use simpler approaches to classification and clustering and still achieve good performance.

The aggregation process operates on a batch of alert records from multiple sensors in the database. It aggregates records generated during a time window according to a simple aggregation criterion, such as by common source IP address.

Aggregates that pass a filter are tagged by HOMER as mapping incidents and all associated records are then removed from analyst view. Thus, HOMER tags obvious mapping episodes in order to prevent mapping episode alerts from flooding analysts' work queues.

5.2.2 HOMER

HOMER is a simple filter classifier, described in more detail by Skorupka et al. [128], that tags aggregates as mapping episodes according to a simple heuristic. If the number of destination addresses in the aggregate incident exceeds a threshold, set by domain experts to the 99.5th percentile, then it is classified as a mapping incident. This threshold was chosen based on a review of a random sample of incidents for the 99th percentile. All alerts associated with an incident are summarized so that analysts do not have to review them individually. Thus, if a source IP communicates with an unusually large number of destination machines in a short period of time, the associated records are summarized and removed from analyst review.

The reduction obtained by filtering events with HOMER is significant. For example, for the period of Sep. 18 to Sep. 23, 2000, MITRE network sensors generated $4,707,323$ alerts, with $71,094$ labeled priority 1. After HOMER, there were $2,824,559$ with $3,690$ priority 1. This is a reduction of 40% overall and 94% for priority 1.

5.2.3 BART Algorithm and Implementation

BART is a process that generates a score for aggregated network sensor alerts that have been tagged as mapping episodes by HOMER. The score indicates the potential that the mapping episode is more than a benign mapping and can help analysts focus attention on critical alerts. Based on Axelsson's [129]

taxonomy, BART is a member of the programmed class of signature-based intrusion detection systems. It looks for clues of unusual behavior in a collection of alerts that make up a mapping episode.

BART computes scores based on domain-specific metrics to identify specific characteristics. The current implementation of BART contains three metrics. These metrics are combined as a scaled product to generate the severity ranking that is scaled to a range $[1, 99]$, with 99 being the most severe. This scale was chosen to suit the Web-based analyst interface. BART also produces a summary report with details from each metric that helps analysts in their investigation of the incident.

BART Metrics

BART currently considers three metrics for detecting anomalous incidents: coverage, popularity, and uniqueness. Additional metrics could be easily added to the process. Here, we define an *event* as the label assigned by the sensor to the individual network alert.

Coverage

Each of the alerts in a mapping episode targets a set of destination IPs. If each event in a mapping episode covers all of the destination IPs targeted in the episode, then we expect that the mapping episode is a simple mapping. On the other hand, if a particular event is directed toward a small subset of destination IPs, it might indicate that a vulnerability has been detected and further exploits have been attempted against a host. This metric indicates a more focused attack toward a subset of targets.

We compute coverage as

$$c = 99\left(1 - \frac{\min\limits_{E}(n_{\text{dstip}})}{N_{\text{dstip}}}\right), \tag{5.1}$$

where E is the set of unique events in the incident, $\min\limits_{E}(n_{\text{dstip}})$ is the minimum number of distinct destination IPs for any unique event in the incident, and N_{dstip} is the total number of unique destination IP addresses in the incident.

The coverage metric detects events that target few destination IPs with events that are not part of the mapping episode. Coverage is inversely proportional to the percentage of destination addresses associated with the least widely distributed event in the incident.

Suppose, for example, that there are $1,000$ events labeled "Scan Proxy" to $1,000$ unique destination IPs and two "WEB MISC" events to two of the destination IPs, all from the same source IP. Then $\min\limits_{E}(n_{\text{dstip}}) = 2$ and $N_{\text{dstip}} = 1,000$, so $c = 98.8$. The "WEB MISC" events are suspicious because they did not cover the IP space covered by the "Scan Proxy" probe.

Popularity

For a benign mapping episode, we expect that the number of alerts will be uniformly distributed over all of the destination IPs. If a particular target is significantly "popular" (targeted with a larger percentage of the records in the incident), the additional attention toward one host might indicate that it has been attacked.

We compute popularity as

$$p = \begin{cases} 5\nu - 1 & \text{if } \nu < 20 \ ; \\ 99 & \text{otherwise} \ , \end{cases} \tag{5.2}$$

where

$$\nu = \frac{\max\limits_{D}(n_r)}{\bar{n}_d} \ , \tag{5.3}$$

and D is the set of unique destination IP addresses in the incident, $\max\limits_{D}(n_r)$ is the maximum number of records associated with any destination IP in the incident, and $\bar{n}_d$ is the average number of records per destination IP for the incident.

The popularity metric detects incidents that target destination IPs with a disproportionately large number of records. Popularity is proportional to the ratio of the maximum number of events per destination address to the average number of events per destination address.

For example, consider an incident that has $1,000$ events labeled "Scan Proxy" to $1,000$ unique destination IPs and 100 additional events of various types to one destination IP in the set, all from the same source IP. Then $\max\limits_{D}(n_r) = 101$ and $\bar{n}_d = 1.1$, so $\nu = 91.8$ and $p = 99$. The additional events directed toward the single IP are suspicious because they target one particular host, indicating higher interest by the source in that host.

Uniqueness

There are a number of records associated with each of the unique events in the incident. For a benign mapping episode, we expect that the number of records for each of the events would be approximately the same. If a particular event has significantly fewer records than the average number of records per event in the incident, it might indicate that a unique event, or an additional exploit, is included with the mapping.

Uniqueness is computed as

$$u = 99 \left(1 - \frac{\min\limits_{E}(n_r)}{N_r} \right) \ , \tag{5.4}$$

where E is the set of unique events in the incident, $\min\limits_{E}(n_r)$ is the minimum number of records for a unique alert in the incident, and N_r is the total number of records in the incident.

The uniqueness metric detects incidents for which there are few records associated with any given event associated with the incident. Uniqueness is inversely proportional to the percentage of records associated with the least used event in the incident.

Consider an example with $1,000$ events labeled "Scan Proxy" to $1,000$ unique destination IPs and one "WEB MISC" alert to one destination IP. Then $\min_E(n_r) = 1$ and $N_r = 1,001$, so $u = 98.9$. The "WEB MISC" event is suspicious because there is only one alert of that type, whereas events in simple mappings are usually directed to most or all of the destination IPs in the incident.

BART Severity Ranking

The BART severity ranking combines coverage, popularity, and uniqueness metrics into a single score of incident severity, s. The combination could be done in many ways. We use the simple formula

$$s = 100(1 - c'p'u') \ , \tag{5.5}$$

where c', p', and u' are scaled functions of the coverage, popularity, and uniqueness metrics. We take one minus the product so that any of the scaled metrics with a low value will result in a low product that will give a high severity. Thus, if any of the metrics are strongly triggered, the severity will be high.

A scaled function of coverage, popularity, and uniqueness metrics allows us to control the number of cases that are given a high severity value. The metric scaling can be done in many ways. We want to invert the value and also have a sharp increase as the metric decreases, so we chose to use the function

$$m' = 1 - \alpha \ln(1 - m/100) \ , \tag{5.6}$$

where m is the metric (i.e., c, p, or u), m' is the scaled metric, and α is a tunable weight factor.

In our implementation, we weighted popularity and uniqueness by $\alpha \approx 0.1$ and coverage by $\alpha \approx 0.2$. This scaling emphasizes the coverage metric.

5.2.4 Other Anomaly Detection Efforts

Although not always the case, a possible attack strategy might include using separate clients to find computers on the network, to find vulnerable services on a target, and to exploit those vulnerabilities. Additional anomaly detection strategies have been designed to include the identification of port scanning incidents and vulnerability scanning incidents.

Port scanning is a form of probe where a potential intruder checks to see what well-known services are being offered by a particular host. An unusually

large number of "common" ports targeted on any single host by a single client is a good indication of a port scan. Common ports to be monitored include those listed in Appendix A of the SANS Top 20 Most Critical Vulnerabilities document [130]. These ports include the File Transfer Protocol (FTP), the Simple Mail Transfer Protocol (SMTP), the Domain Name Service (DNS), the Hypertext Transfer Protocol (HTTP), the Post Office Protocol (POP3), the Internet Message Access Protocol (IMAP), the Simple Network Management Protocol (SNMP), the Network Basic Input Output System (NetBIOS), and other ports. Although it might be common for networks to offer a few of these protocols as external services, it would be unusual to find many of these services provided by a single machine.

Vulnerability scanning is a form of probe where a potential intruder launches a battery of exploits against a particular service. For example, the Nessus Security Scanner could be used to probe an HTTP server for vulnerabilities. This is considerably more aggressive than a simple network mapping probe. An unusually large number of IDS signatures generated by a single client for a single target is a good indication of a vulnerability scan.

A connection logger is the best sensor to monitor for port scanning and host mapping incidents. With the ever-increasing power of today's personal computers, it is relatively easy to periodically check for incidents using a 24-hour monitoring window. The distribution of the number of common ports per source/destination pair, and the distribution of the number of IDS signatures per source/destination pair, appear to follow an exponential decay model. There are many small values and very few large values. A value from the tail of each distribution can be used as a threshold to identify port scans and vulnerability scans, based on a review of a random sample of possible incidents from the 99th percentile. Prioritization of these alerts should account for the fact that scans from internal hosts may be considered to be more significant than scans from external hosts.

5.3 Classifier to Reduce False Alarms

With initial feature selection, aggregation, classification, and ranking in place, analysts were able to detect and label more events (about $4,000$ per week), but there was still a need to reduce the total number of alerts. To do this, we needed to build a classifier that could learn from large numbers of examples, provide comprehensible rules for filtering false alarms, and support incremental updates as new types of attacks and false alarms were identified over time. Our approach was to construct an incremental learning algorithm based on C4.5 [53], a decision-tree/rule generator.

5.3.1 Incremental Classifier Algorithm

The incremental classifier algorithm we developed uses domain-knowledge (DK) rules in addition to labeled training examples. These DK rules can be

directly provided by some expert or, in the case of an incremental learning application, provided by an earlier iteration of the algorithm. The goal of these rules is to modify the attribute selection process during tree construction so that lower scoring attributes can be chosen if the knowledge encoded in the rules indicates such a preference.

The challenge is to balance these preferences against the data. If we have only rules, we learn the same classifier as in a previous iteration. With only data, we learn a classifier fit to the current set of data, but are forgetful of the past. With both rules and data, it is hoped the classifier is informed, but not overwhelmed by historic and current data. An initial study in this area using a modification of C4.5 on ten data sets with four different metrics for attribute closeness confirmed that it is possible to consistently choose an alternative attribute (other than the top-scoring one) without, on average, significantly degrading classification accuracy.

We have developed a grammar for representing domain-knowledge rules. This grammar is shown in Fig. 5.4. Each `<rule>` starts with a sign or a signed number. A positive sign indicates the rule is encouraging the selection of an attribute. A negative sign indicates the rule is discouraging the selection of an attribute by the decision tree. The number at the start of a rule is used for comparison of scores, explained later.

```
<theory>                    ::= <pragma> <rule list> EOF
<pragma>                    ::= rank | percent
<rule list>                 ::= { <rule> }*
<rule>                      ::= <rating> : <simple rule>;
<rating>                    ::= { - | + } <number>
<simple rule>               ::=
    <attribute value pair> { , <attribute value pair> }*
<attribute value pair>      ::=
    <optional name> <attribute value operator> <optional name>
<optional name>             ::= <name> | *
<attribute value operator> ::= = | > | >= | < | <=
```

Fig. 5.4. Grammar for knowledge representation

Each `<simple rule>` includes attribute names and their values. In each `<simple rule>`, the presence of at least one `<attribute value pair>` is required; multiple pairs are separated by commas. In each `<attribute value pair>`, the presence of at least one name is required and the other name could be an asterisk (*), meaning all values.

We modified the original decision-tree procedure, as indicated in the following algorithm:

Given a set of vectors E described by attributes A:

1. **Find** the best attribute a given examples E
2. **Apply Domain Knowledge** rules to adjust attribute ordering
3. **Split** the set of examples E into subsets $E_1, \ldots, E_n$ such that all examples in E_i have $a = v_i$, for $i = 1, \ldots, n$.
4. **Check** for each E_i
   a) If all examples belong to the same class then build leaf node and stop
   b) Else goto Step 1 with examples in E_i

Step 2, Apply Domain Knowledge, applies the provided positive and negative preferences. If a knowledge fragment shows that there is a preference for splitting a certain attribute a, while the data-driven algorithm has selected another attribute, b, for splitting, then the two attributes, a and b, are compared for their rank or percentage. In the percentage method, an applicable preference rule with rating r applies its information gain score to its chosen attribute adjusted by $r \times 100\%$. For example, if $r = +0.1$, the information gain score is increased by 10% (and attribute a may now have a higher score than b). Then, the split method continues as usual in the algorithm. For rank, we extend the split method in a similar way. Instead of finding the single best candidate, the split method finds all applicable tests and sorts them by their gain ratio score. If an attribute preferred in a DK rule is in the top N scores it becomes the top attribute for tree construction regardless of the value of its information gain score.

To use DK from previous iterations, the user selects beforehand a preference method (rank or percentage) and a threshold. We have found empirically that a rank > 3 or a percentage threshold $\leq 10\%$ works best. Because we feel it is more important for old attacks to be remembered, the sign is always set to positive for old rules. With these defaults previously learned rules (e.g., `connection = outbound` $\rightarrow$ `class False_Alarm`) are interpreted as "percent +10 connection = outbound."

5.3.2 Classifier Experiments

Our experiments in false-alarm classification used MITRE operational data labeled by analysts for the time period from Aug. 7, 2000, to Jan. 22, 2001. We then experimented with a number of different incremental approaches. We found methods that forget old examples based on a window of time resulted in slightly worse performance than one that did not forget (i.e., a full-memory approach). Therefore, we need a system that can remember relevant examples for long periods of time, but which will not grow the training data so fast that training times become excessive. To do this, we use a variant of a full-memory approach that stores only the unique examples in the training data. In the first trial, we trained on data from Aug. 7 to Aug. 20 (inclusive), and then tested using data from Aug. 21 to Aug. 27. Trial 2 trained on data from Aug. 7

to Aug. 27 and then tested on data from the week starting Aug. 28. Following this pattern, we obtained 23 trials (1 week had no data). The trials were found to be a very difficult environment for data mining: Only 17 weeks had any false alarms, the percentage of false-alarm events in a single trial week varied from zero to 79%, and the total number of examples in a trial week varied from a low of 5 to a high of 6,159. Furthermore, the data was labeled by six different operators, but no one labeled data for more than 15 of the 24 weeks. These large fluctuations in the makeup of the data did have some effect on the performance of the classifier. The worst performance was on the week with only 5 test examples labeled by an operator who was just starting.

Over the 23 trials, the learned models averaged a predictive accuracy of 88%, a detection rate of 91%, and a false-alarm rate of 12.61%, as shown in Table 5.1. Predictive accuracy is the proportion of correctly classified samples of all samples. Detection rate is also called the true-positive rate or sensitivity, and is the proportion of correctly classified positive samples of all the true-positive samples. False-alarm rate is also called the false-positive rate, and is the proportion of incorrectly classified negative samples of all the true-negative samples.

Another useful metric for measuring the effectiveness of these models is to compare them to baseline constant models that always predict attack or false alarm. Compared to a model that always predicts attack, the learned models have 10% better accuracy and remove 7,876 more true false alarms. Compared to a model that always predicts false alarm, the learned models are 69% more accurate and correctly identify 36,976 more true attacks.

Table 5.1. Classification results for false-alarm reduction on MITRE data

Labeled Type	Detected Type				
	Total	False Alarm		Anomaly	
	Number	Number	Percent	Number	Percent
False Alarm	8,653	7,876	91.02%	777	8.98%
Attack	42,312	5,336	12.61%	36,976	87.39%

The high detection rate of false alarms is operationally useful because it reduces the workload on human analysts, but the 5,336 false positives is problematic because these represent attacks mistakenly labeled as benign.

We then tried the incremental learning algorithm described above, which accepted both training examples and rules. For the first week, we ran C4.5 on a set of data, originally without any domain knowledge. Then we used the results of this first iteration as knowledge fragments for running C4.5 on a new set of data. The results were used in turn as domain knowledge for the third iteration and so on. The goal was to build accurate, simple classifiers. But it is also interesting to note how rules persist in progressive data sets. We do not have exhaustive results, but anecdotal evidence shows that concepts do drift and return in this application. For example, we observed the following

behavior for the discovered rules over 25 intrusion detection data sets collected over a progressive period of time. The following two rules first appeared in iteration 1, and made it to iterations 2 and 3:

```
Rule 2: connection = outbound → class False_Alarm [99.9%]
Rule 5: connection = mitre → class False_Alarm [99.2%]
```

The following rule first appeared in iteration 5, then in iteration 7, then iterations 10 through 13, then iteration 18, and finally in iterations 24 and 25:

```
Rule 5: srcipzone = boundary → class False_Alarm [85.7%]
```

Table 5.2 compares the error rates for running C4.5 without our modifications against the new algorithm (C4.5 with our modifications to use domain knowledge). The error rate of the DK tree approach has about half the average error rate when no DK is used (7.0% vs. 13.5%) and has a much shorter average running time (114.8 sec vs. 1.4 sec), much fewer rules (41.6 rules vs. 4.6 rules), and improved stability over time (17.3 vs. 11.3 max error rate).

The next section describes a second stage of analysis we perform on this data in order to reduce the number of missed attacks.

5.4 Clustering to Detect Anomalies

The previously described classifier has reasonable precision, but attacks may still slip by, especially if they are new. Human review is difficult, again, due to the volume of alerts. An unsupervised learning approach, specifically clustering, can help with this overload to prioritize the false alarms based on a likelihood of being an anomaly [131, 132]. Unsupervised anomaly detection may also identify new attacks that are not explicitly characterized by the classifier. If the identified anomalies correspond to attacks, the clustering approach will be beneficial to the intrusion detection process. In addition to the benefits of prioritizing alerts and potentially identifying new attacks, the resulting cluster model can also give insight into the characteristics of the unique and dynamic nature of alerts for a particular network.

Clustering was evaluated in two experiments. The first evaluation, using data from the 1999 KDD Cup Competition, provided evidence of how well clustering might work for network attacks. The second experiment on MITRE operational data evaluated how well such an approach will work for our needs. In the KDD Cup experiments, we examined clustering for anomaly detection with a reference model. In this scenario, we generated a cluster model from data considered to be normal network traffic and used this model to identify potential anomalies in a batch of alert data different from the training data. A collection of normal alert data does not yet exist in the MITRE data, so a slightly different approach was needed for this data. In this experiment, we trained a cluster model from all alert data classified as false alarm by

Table 5.2. Error rates, run time, and number of rules for two versions of false-alarm classifier on 25 weeks of data

Run	With DK			No DK		
	Error	Time	#	Error	Time	#
1	6.5	0.9	6	6.5	0.8	6
2	0.3	1.0	8	24.2	3.6	11
3	1.3	0.8	5	11.3	3.7	13
4	0.5	1.1	4	0.3	4.6	15
5	1.1	1.1	4	3.7	6.9	17
6	17.6	0.6	0	9.2	7.9	18
7	12.8	2.0	8	51.9	11.5	19
8	17.6	3.0	5	1.4	44.4	37
9	3.4	0.8	5	24.6	26.4	23
10	1.6	1.3	5	17.3	51.9	40
11	0.9	10.0	13	2.3	78.9	42
12	2.9	2.2	5	6.7	59.4	33
13	41.4	1.5	9	50.7	117.7	53
14	0	0.1	0	60.0	127.9	49
15	0	0	0	0.1	131.5	51
16	0	0.2	0	2.5	137.2	51
17	0.5	0.4	0	0.8	134.1	43
18	22.0	0.6	2	20.6	177.0	46
19	33.5	0.8	12	11.2	163.3	62
20	0	0.3	7	0	213.8	61
21	0.2	0.5	0	21.1	218.4	63
22	0	0.5	3	1.1	253.3	73
23	0.8	1.3	0	1.2	286.1	69
24	11.1	1.9	5	7.6	365.8	74
25	0	2.3	10	0.1	244.0	72
Avg.	7.0	1.4	4.6	13.5	114.8	41.6

the previously described classifier. The goal was to identify anomalies in this clean data. The generated model is then used to prioritize the same data set for human review.

5.4.1 Clustering with a Reference Model on KDD Cup Data

Anomalies can be more easily identified with a reference model built from clean, non-anomalous data. With this approach, we first generate a reference model from previously identified, normal data. Then, in a separate process, we assign new data to the nearest cluster in the reference model and apply an outlier criterion to determine those records that are different from normal. We tested this approach on the 1999 KDD Cup data set with its large volume of labeled normal training data. In practice, the explicitly labeled data may not be available, but a body of mostly normal training data may be identified.

In generating the reference model, we selected only the normal training data and then subsampled 20 to 1 to reduce data volume from 972,522 to 48,626 records. We excluded the service field because it takes on too many values. We then generated multiple k-means models using SPSS Clementine [133], an interactive data mining tool, with several arbitrarily chosen numbers of clusters: $k = 3$, 5, 10, 20, and 40. The k-means algorithm uses a Euclidean distance measure. The distance between an individual alert and an alert cluster is the distance between the alert and the cluster centroid. The reason for multiple models is that there may be patterns in the data at multiple levels. A different value of k will yield a model that captures different underlying patterns in the data. Ideally, a validity measure could be computed for each of the different values of k to help identify those that best fit the data. In this work, we chose the k-values arbitrarily.

Once the models are generated, they can be used to identify anomalies in the 311,029 test records. For each of the cluster models, the record is assigned to its closest cluster, and a distance to that cluster centroid is computed. We determined the minimum distance over all models as the simple anomaly metric. This metric is based on the argument that a data record with a very small minimum distance must be well-represented by at least one cluster model and is therefore likely to be a normal record. Other outlier metrics could be computed as well. For instance, a cluster validity measure could be used to weight each distance.

A threshold for determining whether the records are normal or anomalous can be determined from the outlier metrics for each record. In this example, we sorted the distances in descending order and found the point with the sharpest decrease, indicating a transition from anomalous data that is not well-fitted by the reference model and normal data that is similar to the model. To determine this transition, we computed the point of maximum slope as the change in distance over a 1% increment of records. A more robust method that takes into account previous performance could be explored in future work.

With labeled data, we can evaluate the performance of this approach. Table 5.3 shows that at a threshold of 74% of the records, the false-alarm rate is 1.81%, and the detection rate is 91.47%.

Table 5.3. Clustering results for 1999 KDD Cup data with threshold equal to 74% of records

Labeled Type		Detected Type			
	Total	False Alarm		Anomaly	
	Number	Number	Percent	Number	Percent
Normal	60,593	59,497	98.19%	1,096	1.81%
Attack	250,436	21,371	8.53%	229,065	91.47%

5.4.2 Clustering without a Reference Model on MITRE Data

Clustering without a reference model means that we do not have a separate set of training data. We use a batch of network data to generate a cluster model and then use the model to identify anomalies on the same data set. We tested this approach on false-alarm output of a classifier for MITRE network data.

In these experiments, we used the k-means algorithm implemented in SPSS Clementine. For this example, we first used the classification model trained on data from Sep. 18, 2000, to Sep. 22, 2000, to filter known false alarms (Sect. 5.2). When we applied this false-alarm classifier to data from the period, Sep. 18, 2000, to Sep. 22, 2000, 2,880 events were labeled as false alarms and 810 were labeled as attacks. Of the 2,880 records, 8 were known anomalies. These false alarms were input to the clustering algorithm for anomaly detection. We did not have a measure of cluster validity to determine the best number of clusters, k, for the model. Instead, we evaluated arbitrarily chosen values of k for performance on this data and found that $k = 3$ worked well. Intuitively, a small number of clusters would not over-fit the data, allowing outliers to be detected. We currently use a simple outlier criterion, the distance to the nearest cluster centroid. Other outlier criteria are possible [132]. The data is ranked in order of distance, with records of highest distance considered most likely to be anomalous.

For evaluation purposes, we applied a simple threshold algorithm to identify the sharpest drop in distances as the break between anomalous and normal. As with the KDD Cup data, we chose the threshold to be the point of maximum slope computed as change in distance over 1% increments of records. In practice, this threshold would more likely be determined by the number of records that human reviewers could evaluate or by the number of false alarms that could be tolerated in the system. Here, the data is partially labeled by human reviewers. The labels are not used in the clustering process but only to estimate performance of the approach. Performance is only an estimate because the data is incompletely labeled.

Although these results are preliminary, using this approach we found some of the rare anomalous alerts. Given 2,880 alert events containing 8 known anomalies, 7 of the 8 were ranked in the top 20 events. All 8 were found in the top 225. The threshold based on maximum slope is at 201 (7%). With this threshold, the false-alarm rate is 6.75% and detection rate is 87.5% as shown in Table 5.4.

5.5 Conclusion

This paper gives a description of how MITRE is using data mining to improve the processing of network alerts. Our goals in this work are to reduce

Table 5.4. Clustering results for classifier false-alarm output on MITRE data with a threshold of 201 records

Labeled Type	Detected Type				
	Total	False Alarm		Anomaly	
	Number	Number	Percent	Number	Percent
Normal	2,872	2,678	93.25%	194	6.75%
Attack	8	1	12.50%	7	87.50%

the number of false alarms generated by existing network sensors and to detect unusual alert events that may correspond to new attacks. To this end, we use a combination of decision trees and k-means clustering. When tested in 23 trials using MITRE data from Aug. 7, 2000, to Jan. 22, 2001, the classification approach had an overall detection rate of 91% and a false-alarm rate of less than 13%. The approach also significantly reduced the number of alerts requiring human review. Preliminary clustering results with both KDD Cup and MITRE operational data show that this approach has potential for accurate anomaly detection. With the KDD Cup data, the clustering model has a false-alarm rate of only 1.8% and a detection rate of 91.5%. Based on an initial experiment using MITRE data, the false-alarm rate is 6.75% and detection rate is 87.5%.

Table 5.5. Incident Label, pertaining to the record's incident, if one exists

Name	Type	Description / Sample Value
INCIDENTCLASSLABELID	Short	The classification ID of the associated incident, if one exists

Table 5.6. Base Features, obtained directly from sensorlog

Name	Type	Description / Sample Value
JULDATE	Long Integer	A unique integer for each day
RECID	Long Integer	Unique key in sensorlog
SENSORNAME	CHAR(18)	Sensor host / location
DATATYPE	CHAR(5)	Type of sensor (FW1, JID, LOG, etc.)
STARTDATE	Date / Time	Start date and time of event. YYYY/MM/DD HH:MM:SS
ENDDATE	Date / Time	End date and time of event (often missing). YYYY/MM/DD HH:MM:SS
PRIORITY	Short Integer	1, 2, or 3. 1 = highest priority. Highly dependent on sensor
EVENT	CHAR(50)	Description or Name of Event – as provided by the sensor
PROTOCOL	CHAR(10)	TCP, UDP, ICMP
SRCPORT	Unsigned Integer	$0 \leq x \leq 65535$
DSTPORT	Unsigned Integer	$0 \leq x \leq 65535$
SRCIP	CHAR(15)	x.x.x.x : $0 \leq x \leq 255$
DSTIP	CHAR(15)	x.x.x.x : $0 \leq x \leq 255$
DSTDNS	CHAR(255)	The domain name of the destination (often missing)
EVENTINFO	CHAR(255)	Information such as TCP flags
EVENTINFO2	CHAR(255)	More information about the event.

Table 5.7. Record Features, calculated from an individual record

Name	Type	Description / Sample Value
DURATION	Long	Duration in seconds (only filled if the record has an endtime)
STARTDAYOFWEEK	CHAR(3)	Mon, Tue, Wed, etc.
WEEKDAY	Boolean	1: Mon–Fri; 0: Sat, Sun
STARTTIMEBLOCK	CHAR(3)	day: 7am–4:59pm; eve: 5pm–10:59pm; night: 11pm–6:59am
STARTYEAR	Integer	4-digit year
STARTMONTH	Short	1–12
STARTDAY	Short	1–31
STARTHOUR	Short	0–23
STARTMINUTE	Short	0–59
STARTSECOND	Short	0–59
ENDYEAR	Integer	4-digit year
ENDMONTH	Short	1–12
ENDDAY	Short	1–31
ENDHOUR	Short	0–23
ENDMINUTE	Short	0–59
ENDSECOND	Short	0–59
DISCRETEDURATION	CHAR(9)	‘neg’, ‘zero’, ‘1min’, ‘2–5min’, ‘6–20min’, ‘21–40min’, ‘over 40min’
SRCTIMEBETWEEN	Long	number of seconds since last connection from this src IP
DSTTIMEBETWEEN	Long	number of seconds since last connection to this dst IP
SRCSUBNET	CHAR(11)	x.x.x : $0 \leq x \leq 255$
DSTSUBNET	CHAR(11)	x.x.x : $0 \leq x \leq 255$

Table 5.8. Record Features, calculated from an individual record

Name	Type	Description / Sample Value
BROADCAST	Boolean	Is the last IP octet set to 255 (indicates a broadcast)? 0, 1
SAMEPORT	Boolean	Are the src port and dst port the same? 0, 1
LOWSRCPORT	Boolean	src port < 20? 0, 1
LOWDSTPORT	Boolean	dst port < 20? 0, 1
HIGHDSTPORT	Boolean	dst port > 1024? 0, 1
WELLKNOWNDSTPORT	Boolean	Is the dst port a well-known port? 0, 1
SYNFLAG	Boolean	TCP protocol, Checks eventinfo for Syn flag. 0, 1
SAMESUBNET	Boolean	Are the src and dst subnet's the same? 0, 1
HOTSRCIP	Boolean	Is the src IP in the list of hot IPs? 0, 1
HOTSRCSUBNET	Boolean	Does the source subnet contain a hot IP? 0, 1
HOTDSTIP	Boolean	Is the dst IP in the list of hot IPs? 0, 1
HOTDSTSUBNET	Boolean	Does the destination subnet contain a hot IP? 0, 1
SRCZONE	CHAR(8)	internal, dmz, boundary, external, nonmitre
DSTZONE	CHAR(8)	internal, dmz, boundary, external, nonmitre
CONNECTION	CHAR(8)	inbound, outbound, mitre, nonmitre
SRCTLD	CHAR(16)	Last token of domain name (e.g., .org, .com, etc).
DSTTLD	CHAR(16)	Last token of domain name (e.g., .org, .com, etc).
SRCSLD	CHAR(64)	Last two tokens of domain name (e.g., .mitre.org).
DSTSLD	CHAR(64)	Last two tokens of domain name (e.g., .mitre.org).
SRCDOMAINTOKENS	Short	Number of src domain tokens (strings between periods)
DSTDOMAINTOKENS	Short	Number of dst domain tokens (strings between periods)
IPMAPHEUR	Boolean	Was this record caught by our IP mapping heuristic? 0, 1

Table 5.9. Time Window Features (Group 1), calculated using a time window on records sorted by SRCIP, DSTIP, and STARTTIME

Name	Type	Description / Sample Value
SAMECON2	Long	Number of events with same SRCIP and DSTIP in a 2 second window
SAMECON86400	Long	Number of events with same SRCIP and DSTIP in a 1 day window
SAMECON1209600	Long	Number of events with same SRCIP and DSTIP in a 2 week window
RECIPCON2	Long	Number of events with reciprocal SRCIP and DSTIP in a 2 second window
RECIPCON600	Long	Number of events with reciprocal SRCIP and DSTIP in a 10 minute window
RECIPCON1209600	Long	Number of events with reciprocal SRCIP and DSTIP in a 2 week window

Table 5.10. Time Window Features (Group 2), calculated using a time window on records sorted by SRCIP and STARTTIME

Name	Type	Description / Sample Value
SRCAVGBETWEEN3600	Float	Average time between connections from same SRCIP in 1 hour window
SRCSTDBETWEEN3600	Float	Standard deviation of time between connections from same SRCIP in 1 hour window
SRCZBETWEEN3600	Float	z-score of time between connections from same SRCIP in 1 hour window
SRCAVGDUR3600	Float	Average connection duration from same SRCIP in 1 hour window
SRCSTDDUR3600	Float	Standard deviation of connection duration from same SRCIP in 1 hour window
SRCZDUR3600	Float	z-score of duration of connections from same SRCIP in 1 hour window
SRCAVGBETWEEN1209600	Float	Average time between connections from same SRCIP in 2 week window
SRCSTDBETWEEN1209600	Float	Standard deviation of time between connections from same SRCIP in 2 week window
SRCZBETWEEN1209600	Float	z-score of time between connections from same SRCIP in 2 week window
SRCAVGDUR1209600	Float	Average connection duration from same SRCIP in 2 week window
SRCSTDDUR1209600	Float	Standard deviation of connection duration from same SRCIP in 2 week window
SRCZDUR1209600	Float	z-score of duration of connections from same SRCIP in 2 week window
SRCHIPRI	Long	Number of high priority records for this SRCIP in 1 day window

Table 5.11. Time Window Features (Group 3), calculated using a time window on records sorted by DSTIP and STARTTIME

Name	Type	Description / Sample Value
DSTAVGBETWEEN3600	Float	Average time between connections to same DSTIP within past hour
DSTSTDBETWEEN3600	Float	Standard deviation of time between connections to same DSTIP within past hour
DSTZBETWEEN3600	Float	z-score of time between connections to same DSTIP within past hour
DSTAVGDUR3600	Float	Average connection duration to same DSTIP within past hour
DSTSTDDUR3600	Float	Standard deviation of connection duration to same DSTIP within past hour
DSTZDUR3600	Float	z-score of duration of connections to same DSTIP within past hour
DSTAVGBETWEEN1209600	Float	Average time between connections to same DSTIP within past 2 weeks
DSTSTDBETWEEN1209600	Float	Standard deviation of time between connections to same DSTIP within past 2 weeks
DSTZBETWEEN1209600	Float	z-score of time between connections to same DSTIP within past 2 weeks
DSTAVGDUR1209600	Float	Average connection duration to same DSTIP within past 2 weeks
DSTSTDDUR1209600	Float	Standard deviation of connection duration to same DSTIP within past 2 weeks
DSTZDUR1209600	Float	z-score of duration of connections to same DSTIP within past 2 weeks
DSTHIPRI	Long	Number of high priority records for this DSTIP within past day

6

Intrusion Detection Alarm Clustering

Klaus Julisch

6.1 Introduction

Over the past 10 years, the number as well as the severity of network-based computer attacks have significantly increased [134]. As a consequence, classic information security technologies, such as authentication and cryptography, have gained in importance. Simultaneously, intrusion detection has emerged as a new and potent approach to protect information systems [135, 136]. In this approach, so-called *intrusion detection systems* (IDSs) are used to monitor information systems for signs of security violations. Having detected such signs, IDSs trigger alarms to report them. These alarms are presented to a human operator who evaluates them and initiates an adequate response. Examples of possible responses include law suits, firewall reconfigurations, and the repairing of discovered vulnerabilities.

Evaluating intrusion detection alarms and conceiving an appropriate response was found to be a challenging task. In fact, practitioners [125, 137] as well as researchers [138, 139] have observed that IDSs can easily trigger thousands of alarms per day, up to 99% of which are *false positives* (i.e., alarms that were mistakenly triggered by benign events). This flood of mostly false alarms makes it very difficult to identify the hidden *true positives* (i.e., those alarms that correctly flag attacks). For example, the manual investigation of alarms has been found to be labor-intensive and error-prone [125, 137, 140]. Tools to automate alarm investigation are being developed [140–142], but there is currently no silver-bullet solution to this problem.

In a series of past publications [139, 143–145], we have introduced a novel semiautomatic approach for handling intrusion detection alarms efficiently. Central to this approach is the notion of alarm root causes. Intuitively, the *root cause* of an alarm is the reason for which it occurs. We have made the key observation that in most environments, a few dozens of highly persistent root causes account for over 90% of all alarms. Moreover, given the persistence of these root causes, they do not disappear until someone removes them. As a consequence, they trigger immense amounts of alarms, which distract

the intrusion detection analyst from spotting the real (generally more subtle) attacks. To free the intrusion detection analyst from these highly persistent root causes and their associated alarms, we have developed the following three-step process:

Clustering: First, we use CLARAty[1] – a clustering algorithm specifically developed for this purpose – to cluster and summarize alarms that are likely to share the same root cause.

Root Cause Analysis: Next, we interpret the clusters produced by CLARAty. The goal of this step is to identify for each alarm cluster the root causes that triggered its constituent alarms.

Taking Action: Finally, we act upon the clusters. For example, very often it is possible to eliminate the identified root causes so they can no longer trigger alarms. Other times, the processing of alarms triggered by well-known root causes can be automated. Either way, our experience has shown that the number of alarms that the intrusion detection analyst must henceforth handle drops by 70% on the average.

For the sake of completeness, we will briefly review this three-step process, but refer the reader to the original publications for details and a treatment of related work [139, 143–146]. More specifically, Sect. 6.2 defines the notion of root causes more precisely, and Sect. 6.3 describes the CLARAty clustering algorithm. The novel contribution of this chapter lies in Sects. 6.4 and 6.5, which address the difficult questions of cluster validation (does CLARAty really find clusters whose constituent alarms share the same root cause?) and cluster tendency (do intrusion detection alarms naturally form clusters, or does CLARAty impose an artificial structure, which the data does not support?). Section 6.6 concludes and summarizes this chapter.

6.2 Root Causes and Root Cause Analysis

This section defines the concepts of root causes and root cause analysis. The *root cause* of an alarm, as mentioned in the introduction, is the reason for which the alarm is triggered. Another useful mental construct is that root causes are problems that affect components and cause them to trigger alarms. For example, a failure can affect the implementation of a TCP/IP stack, and cause it to fragment all outbound IP traffic, which triggers "Fragmented IP" alarms. Similarly, a worm is a root cause that affects a set of hosts and causes them to trigger alarms when the worm spreads. *Root cause analysis* is the task of identifying root causes as well as the components they affect.

We do not attempt to formally define root causes or root cause analysis, because such an attempt seems fruitless. In fact, in the dependability field, the term *fault* denotes a concept that is very similar to a root cause, and the

[1] **CL**ustering **A**larms for **R**oot cause **A**nalysis (CLARAty).

dependability notion of *fault diagnosis* corresponds to root cause analysis [147, 148]. Neither faults nor fault diagnosis have been formally defined. Therefore, we consider it unlikely that the intrusion detection equivalents of these terms possess formal definitions.

By way of illustration, we next list some typical root causes that CLARAty revealed, as well as the alarms that these root causes triggered. Note that we observed all of the below root causes in our practical work with real-world intrusion detection systems:

1. A popular HTTP server with a broken TCP/IP stack that fragments outgoing traffic. "Fragmented IP" alarms ensue when the server responds to client requests.
2. At one site, a misconfigured secondary DNS server performed half-hourly DNS zone transfers from the primary DNS server. The resulting "DNS Zone Transfer" alarms are no surprise.
3. A Real Audio server whose traffic remotely resembles TCP hijacking attacks. This caused the deployed IDS to trigger countless "TCP Hijacking" alarms.
4. A firewall that has Network Address Translation (NAT) enabled funnels the traffic of many users and thereby occasionally seems to perform host scans. In detail, a NAT-enabled firewall acts as proxy for its users. When these users *simultaneously* request external services, then the firewall will proxy these requests and the resulting SYN packets resemble SYN host sweeps.
5. A load balancing reverse proxy, such as Cisco LocalDirector, that dispatches Web client requests to the least busy server. The resulting traffic patterns resemble host scans that trigger alarms on most IDSs.
6. Many network management tools query sensitive variables in the management information base (MIB) and thereby trigger IDS alarms. (Other network management tasks such as vulnerability scanning or network mapping offer further examples of root causes.)
7. Macintosh FTP clients, which issue the SYST command on every FTP connection, trigger an abundance of "FTP SYST command" alarms. The FTP SYST command is reported by some IDSs because it provides reconnaissance information about the FTP server.
8. A distributed denial-of-service (DDoS) attack being launched from an external network against a Web hosting site triggered "SYN Flooding" alarms.
9. External Code Red infected machines [149] scanning the internal network for vulnerable servers. Most IDSs trigger alarms to report traffic resulting from Code Red.

6.3 The CLARAty Alarm Clustering Method

This section summarizes the CLARAty alarm clustering method in three steps: Section 6.3.1 describes the intuition behind CLARAty, Sect. 6.3.2 presents how it actually works, and Sect. 6.3.3 contains a use case. First, however, a few introductory definitions are in order.

Intrusion detection systems trigger *alarms* to report presumed security violations. This chapter models alarms as tuples over the Cartesian product $\mathrm{dom}(A_1) \times \cdots \times \mathrm{dom}(A_n)$ attributes, and $\mathrm{dom}(A_i)$ is the domain (i.e., the range of possible values) of attribute A_i. The *alarm attributes* (*attributes* for short) capture intrinsic alarm properties, such as the source IP address of an alarm, its destination IP address, its alarm type (which encodes the observed attack), and its time stamp. The value that attribute A_i assumes in alarm $\mathbf{a}$ is denoted by $\mathbf{a}[A_i]$. This article models *alarm log* $\mathcal{L}$ as sets of alarms. This model is correct because alarms are implicitly ordered by virtue of their time stamps; moreover, unique alarm-identifiers can be used to guarantee that all alarms are pairwise distinct.

6.3.1 Motivation

To get an intuitive understanding of how CLARAty works, let us reconsider the sample root causes presented in Sect. 6.2. In particular, for the first root cause (i.e., the HTTP server whose broken TCP/IP stack fragments outgoing packets), a network-based IDS will trigger alarms of type "Fragmented IP". Moreover, all of these alarms will originate from source port 80 of the HTTP server. The alarms are targeted at HTTP clients on non-privileged ports. Furthermore, under the assumption that the HTTP server is mainly used between Monday and Friday, one will observe that the bulk of alarms occurs on workdays. Finally, a "Fragmented IP" alarm is triggered each time that the HTTP server responds to a client request. For a popular HTTP server this results literally in a flood of "Fragmented IP" alarms. The key observation to be made is that all alarms caused by the broken TCP/IP stack can be summarized by the following generalized alarm:

> On workdays, there is a large number of "Fragmented IP" alarms originating from port 80 of the HTTP server, targeted against non-privileged ports of HTTP clients.

We call this a *generalized alarm* as it contains generalized attribute values such as "non-privileged port", "HTTP clients", and "workdays", which represent sets of elementary attribute values. The first row of Table 6.1 shows this generalized alarm in a more schematic manner. Similarly, the second row of the table shows that the second root cause can be modeled by the generalized alarm of "'DNS zone transfer' alarms being triggered from a non-privileged port of the secondary DNS server against port 53 of the primary DNS server". Analogously, the remaining rows of Table 6.1 show the generalized alarms that

Table 6.1. The generalized alarms induced by nine sample root causes

RC	Source-IP	Src-Port	Destination-IP	Dst-Port	Alarm Type
1	HTTP server	80	HTTP clients	Non-priv.	Fragmented IP
2	Sec. DNS server	Non-priv.	Prim. DNS server	53	DNS zone transfer
3	Real Audit server	7070	Real Audio clients	Non-priv.	TCP hijacking
4	Firewall	Non-priv.	External network	Privileged	Host scan
5	Reverse proxy	Non-priv.	HTTP servers	80	Host scan
6	Mgmt. console	Non-priv.	SNMP clients	161	Suspicious GET
7	Mac FTP clients	Non-priv.	FTP server	21	FTP SYST
8	External network	Non-priv.	HTTP servers	80	SYN flood
9	External network	Non-priv.	Internal network	80	Code Red

the other root causes induce. The "RC" (root cause) column of the table refers to the item numbers in the enumeration of root causes in Sect. 6.2; the entry "Non-priv." denotes the set $\{1025, \ldots, 65535\}$ of non-privileged ports, and the entry "Privileged" stands for the set of privileged ports below 1025.

From the above examples we observe that a significant class of root causes manifest themselves in large alarm clusters that are adequately modeled by generalized alarms. By *adequately*, we mean that generalized alarms are capable of capturing and representing the main features of alarm clusters without losing much essential information. The CLARAty algorithm will search for such large alarm clusters that are adequately modeled by generalized alarms. According to the logic of abduction [150–152], it is then likely that the alarms of such clusters share the same root cause.

6.3.2 The CLARAty Algorithm

The CLARAty algorithm heuristically searches for large alarm clusters that are adequately modeled by generalized alarms. An alarm cluster is large if it comprises more than *min_size* alarms, with *min_size* being a user-defined integer parameter. Moreover, the user must also define the space of possible generalized alarms that CLARAty searches. To this end, she defines a separate generalization hierarchy $\mathcal{G}_i$ for each alarm attribute A_i. A *generalization hierarchy* (also known as *is-a* hierarchy) is a directed tree[2] whose leafs are the elementary values in $\mathrm{dom}(A_i)$, and whose internal nodes are so-called *generalized attribute values*, which represent sets of values in $\mathrm{dom}(A_i)$. The edges of the tree show how lower-level concepts can be generalized into higher-level ones.

[2] More generally, a generalization hierarchy can be defined as a single-rooted, connected, directed acyclic graph (DAG). This generalization is further discussed in [144, 145].

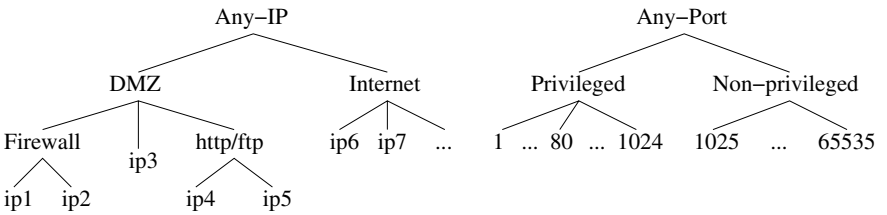


Fig. 6.1. Sample generalization hierarchies

By way of illustration, Fig. 6.1 shows sample generalization hierarchies for IP addresses and port numbers. As can be seen, the IP value *ip3* can be generalized to the generalized attribute value *DMZ*, which in turn represents the set $\{ip1, ip2, ip3, ip4, ip5\}$ of elementary IP values. Similarly, Fig. 6.1 shows that the port number 1 can be generalized to *Privileged*, which in turn can be generalized to *Any-Port*. Again, *Privileged* is a generalized attribute value and represents the set $\{1, \ldots, 1024\}$ of elementary values.

We are now ready to describe the CLARAty clustering algorithm: Given an alarm log $\mathcal{L}$, a minimum cluster size *min_size*, and generalization hierarchies $\mathcal{G}_i$ for all attributes A_i, $i = 1, \ldots, n$, CLARAty repeatedly generalizes the alarms in $\mathcal{L}$. Generalizing alarms is done by heuristically choosing an attribute A_i and replacing the A_i values of all alarms in $\mathcal{L}$ by their parent values in $\mathcal{G}_i$. This process continues until an alarm has been found to which at least *min_size* of the original alarms can be generalized. This alarm constitutes the output of the algorithm. As the reader might have noted correctly, CLARAty is a variant of the well-known technique Attribute-Oriented Induction (AOI) [153, 154].

Figure 6.2 shows the pseudo-code of the alarm clustering method. Line 1 copies the alarm log $\mathcal{L}$ into a relational database table T. This table has one attribute (or column) for each alarm attribute A_i. In addition, the table T possesses the integer-valued attribute *count*, which is used for bookkeeping, only. Line 2 sets the count attributes of all alarms to 1. In lines 3 to 9, the algorithm loops until an alarm $\mathbf{a}$ has been found, whose count value is at least *min_size*. Line 4 selects an alarm attribute A_i according to a heuristic, which we will describe in a moment. Lines 5 and 6 replace the A_i values of all alarms in T by their parent values in $\mathcal{G}_i$. By doing so, previously distinct alarms can become identical. Two alarms $\mathbf{a}$ and $\mathbf{a}'$ are *identical* if $\mathbf{a}[A_i] = \mathbf{a}'[A_i]$ holds for all attributes A_i, while $\mathbf{a}[count]$ and $\mathbf{a}'[count]$ are allowed to differ. Steps 7 and 8 merge identical alarms into a single generalized alarm whose count value equals the sum of individual counts. In this way, the count attribute always reflects the number of original alarms that are summarized by a given generalized alarm. Moreover, each generalized alarm $\mathbf{a}$ represents an alarm cluster of size $\mathbf{a}[count]$. To conclude the discussion, we now specify the heuristic that we use in line 4:

Input: An alarm clustering problem $(\mathcal{L}, min\_size, \mathcal{G}_1, \ldots, \mathcal{G}_n)$
Output: A heuristic solution for $(\mathcal{L}, min\_size, \mathcal{G}_1, \ldots, \mathcal{G}_n)$
Algorithm:

```
1:  T := L;                                          // Store log L in table T.
2:  for all alarms a in T do a[count] := 1;          // Initialize counts.
3:  while ∀a ∈ T : a[count] < min_size do {
4:      Use heuristics to select an attribute A_i, i ∈ {1, ..., n};
5:      for all alarms a in T do                     // Generalize attribute A_i
6:          a[A_i] := father of a[A_i] in G_i;
7:      while identical alarms a, a' exist do // Merge identical alarms.
8:          Set a[count] := a[count] + a'[count] and delete a' from T;
9:  }
10: Output all generalized alarms a ∈ T with a[count] ≥ min_size;
```

Fig. 6.2. Heuristic alarm clustering algorithm

Definition 1 (Attribute Selection Heuristic). *For each attribute A_i, let $F_i := \max\{f_i(v) \mid v \in \mathcal{G}_i\}$ be the maximum of the function*

$$f_i(v) := \textit{SELECT } sum(count) \textit{ FROM } T \textit{ WHERE } A_i = v \ ,$$

which counts the number of original alarms that had their A_i-attribute generalized to the value v. Line 4 of Fig. 6.2 selects any attribute A_i whose F_i value is minimal, i.e., the F_i value must satisfy $\forall j : F_i \leq F_j$. □

The intuition behind this heuristic is that if there is an alarm $\mathbf{a}$ that satisfies $\mathbf{a}[count] \geq min\_size$, then $F_i \geq f_i(\mathbf{a}[A_i]) \geq min\_size$ holds for all alarm attributes A_i, $i = 1, \ldots, n$. In other words, an alarm $\mathbf{a}$ with $\mathbf{a}[count] \geq min\_size$ cannot exist (and the algorithm cannot terminate) unless $F_i \geq min\_size$ holds for all attributes A_i. We therefore use it as a heuristic to increase any of the smallest F_i values by generalizing its corresponding attribute A_i. Other heuristics are clearly possible, but the one above performs favorably in our experience. Moreover, rather than merely varying the heuristic, one could even conceive a completely different clustering algorithm, for example one that is based on partitioning or hierarchical clustering [23, 155].

6.3.3 CLARAty Use Case

We have successfully used CLARAty on millions of intrusion detection alarms to identify their root causes [143, 145]. In typical cases, CLARAty yields generalized alarms such as the one shown in Table 6.2.

An analysis of this generalized alarm has revealed the following root cause: The deployed network-based IDS triggers "Cisco Catalyst 3500 XL Bug" alarms because it observes URLs containing the substring "/exec/". This

Table 6.2. A sample generalized alarm

Attribute	Attribute Value
Source IP	Office PC users
Source port	Non-privileged, i.e., ≥ 1025
Destination IP	www.amazon.com
Destination port	80
Alarm type	Cisco Catalyst 3500 XL Bug
Time stamp	Workdays, office hours

substring is flagged because it has been associated with a vulnerability in the Cisco Catalyst 3500 XL switch [156]. In the present case, however, the "Office PC users" were eagerly shopping on "workdays" during "office hours" at "www.amazon.com". However, virtually all URLs at amazon.com contain the substring "/exec/", which results in the above cluster of alarms. Clearly, all alarms in this cluster are false positives. This concludes our use case of how CLARAty facilitates root cause analysis.

6.4 Cluster Validation

Section 6.4.1 explains why clustering results are a priori suspect and need to be validated. Section 6.4.2 gives a brief introduction to cluster validation, and Sect. 6.4.3 presents the form of validation we use for CLARAty.

6.4.1 The Validation Dilemma

Clustering has traditionally been used to explore new data sets whose properties were poorly understood. Therefore, the principal goal of clustering was to uncover the hidden structures in a data set, and to stimulate theories that explain them. Accordingly, there were claims that the main criterion for assessing a clustering result was its interpretability and usefulness. However, clustering methods have important limitations that make such an ad hoc approach to cluster validation a dangerous endeavor:

The random data phenomenon: Clustering methods *always* find clusters, no matter whether they exist or not. In fact, even when applied to random data, they mechanically group it into clusters. Clearly, such clusters are meaningless at best, and misleading at worst.

Imposed structures: Clustering methods are not "neutral", but have a tendency to *impose* a structure on the data set [157–160]. For example, the k-means method is known to favor spherical (a.k.a. globular) clusters of approximately equal sizes [23, 159, 160]. When a data set contains clusters of a different kind (e.g., clusters that form parallel lines in the plane), then the k-means method is nonetheless inclined to impose "synthetic"

clusters of globular shapes and similar sizes. Clearly, these clusters are an artifact of the k-means method, and do not reflect the true structure of the data set.

Method-dependent results: A corollary to the last point is that markedly different results can be obtained when a given data set is analyzed by different clustering methods [157–159, 161].

Given these drawbacks, and given that the human mind is quite capable of providing post hoc justification for clusters of dubious quality, there are clear dangers in a purely manual approach to cluster validation. Specifically, the risk of inadvertently justifying and approving meaningless clusters can be high [155, 159, 161]. Clearly, these reservations also apply to the alarm clusters found by CLARAty. We would therefore like to prove that the clusters found by CLARAty are *correct* in the sense that the alarms in a given cluster share the same root cause. If this form of correctness is satisfied, then CLARAty eliminates redundancy by grouping alarms that share the same root cause. Moreover, if one understands the root cause of any alarm in an alarm cluster, then one understands the root causes of all alarms in the cluster. This fact reduces the risk of misinterpreting alarm clusters.

Unfortunately, correctness of clusters cannot be guaranteed, neither by CLARAty nor by any other clustering algorithm. In fact, root causes are a model-world concept, which exist only in the world of our thinking. Computer programs are not aware of root causes and can therefore not enforce the requirement that all alarms of an alarm cluster must share the same root cause. To illustrate this, let us consider a machine whose broken TCP/IP stack fragments most IP traffic. Suppose that this machine is behind a router that itself fragments a substantial fraction of the traffic passing through it. Now, let an IDS in front of the router trigger a "Fragmented IP" alarm for a packet from said machine. Unless substantial background knowledge about the state of the system is available, there is no way of deciding if the alarm's root cause is the broken TCP/IP stack or the fragmenting router. More generally, if only an alarm log is given, it is not possible to decide whether two or more alarms have the same root cause. We therefore know that CLARAty cannot be correct in the above sense.

6.4.2 Cluster Validation in Brief

Despite the negative result of the last section, the question about the quality of clusters remains: How do we know that the clusters found by CLARAty are any good, rather than being mere random groupings? This question is addressed by the field of cluster validation, according to which a set of clusters are *valid* if they cannot reasonably be explained by chance or by the idiosyncrasies of a clustering method [155, 159, 162, 163]. In practice, a set of clusters is declared valid, if there is evidence that it correctly captures the natural groups of the underlying data set.

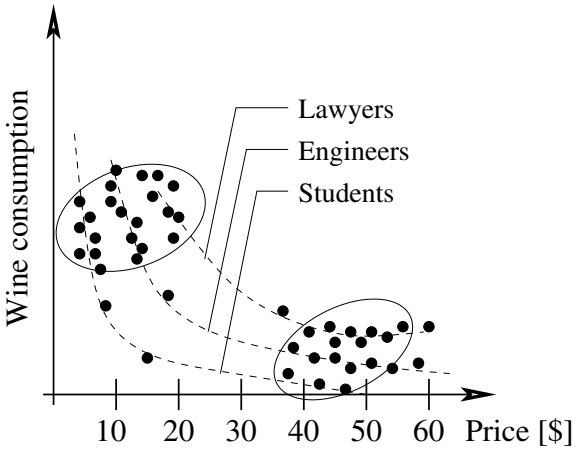


Fig. 6.3. Example of valid clusters that have no intuitive interpretation

It is important to note that cluster validation is a completely formal endeavor, that is rid of any notions of semantics. This can lead to the situation where a cluster structure is declared "valid" even though it has no meaningful interpretation. For example, Fig. 6.3 plots for lawyers, engineers, and students their respective wine consumptions at various wine prices. Note that only a few data points are available in the \$20 to \$40 price range. Clearly, the two elliptic clusters capture the structure of the data set rather well. Therefore, these clusters are valid by most validity measures, even though they have no obvious interpretations (at least not in terms of people's professions). This illustrates that cluster validation can at best increase our confidence in a cluster structure, but it cannot prove the structure to be correct or meaningful [157, 163–165].

In [145], we made a very detailed and critical analysis of cluster validation methods. In summary, we found that the published validation methods are of little practical value to us. Specifically, the most significant limitations we encountered are as follows:

- High computational costs.
- The assumption that the "correct" clustering result is known, which generally is not the case.
- Crucial assumptions (e.g., about what a random data set looks like) whose impact on the validation result is insufficiently understood.
- Simplifying assumptions (e.g., about the characteristics of the data set to be clustered) that are generally not satisfied in reality.
- Contradictory results obtained by different validation methods.

6.4.3 Validation of Alarm Clusters

Replication analysis [166–168] is the validation technique that we found most useful for our purposes. Replication analysis is based on the logic of cross-validation [169] as used in regression studies. Specifically, a cluster structure is declared valid if it is consistently discovered across different samples that are independently drawn from the same general population. Replication analyses comprise five major steps:

1. Two samples are required to conduct a replication analysis. Perhaps the most direct way to accomplish this is to randomly divide one larger data set into two samples.
2. A clustering method is chosen, and the first sample is partitioned using this method.
3. The second sample is partitioned using the same clustering method as in step 2.
4. The second sample is repartitioned by assigning each of its data objects to the most similar (according to some measure) cluster in the partition of the first sample. This yields another clustering of the second sample.
5. A numeric measure of partition agreement is used to compare how well the two partitions of the second sample match. The greater the level of agreement between the two partitions, the more confidence one may have in their validity.

Hence, replication analysis considers a clustering result more trustworthy if it is not affected by sampling or other spurious/arbitrary decisions that might have been involved in its generation. However, summarizing the results of a replication analysis in a single number as done in step 5 seems overly formal given that this number is interpreted in a qualitative way, where "larger is better". Moreover, the fourth step in a replication analysis partitions the second sample according to its resemblance to the clusters in the first sample. The details of this step are open to interpretation, and few guidelines exist. However, the choices made here are known to influence the final results [166]. For all of these reasons, we modified the "classic" replication analysis presented above to address these concerns.

Let $\mathcal{L}$ be an alarm log and let $\mathcal{L}_0 \subseteq \mathcal{L}$ be a randomly chosen sub-log of $\mathcal{L}$. We randomly partition the alarm log $\mathcal{L} \setminus \mathcal{L}_0$ into two disjoint equal-sized sub-logs $\mathcal{L}_1$ and $\mathcal{L}_2$ (i.e., $|\mathcal{L}_1| = |\mathcal{L}_2|$ and $\mathcal{L}_1 \uplus \mathcal{L}_2 = \mathcal{L} \setminus \mathcal{L}_0$). Then, we apply CLARAty separately to the logs $\mathcal{L}_0 \cup \mathcal{L}_1$ and $\mathcal{L}_0 \cup \mathcal{L}_2$ and compare the resulting generalized alarms. One could mark generalized alarms that are not discovered in both alarm logs as questionable, so that they are investigated more carefully. However, generalized alarms that do not replicate are very rare so that we decided to discard them without mention. The advantage of this approach is that it improves the robustness of the alarm clustering method while being completely transparent to the user. The user only sees "robust"

alarm clusters, i.e., alarm clusters that are not affected by slight noise or sampling.

Determining the size of the alarm log $\mathcal{L}_0$ is an important decision. The larger $\mathcal{L}_0$ is, the more alarms the two alarm logs $\mathcal{L}_0 \cup \mathcal{L}_1$ and $\mathcal{L}_0 \cup \mathcal{L}_2$ share, and the weaker the form of robustness that we enforce by means of replication. In our practical work, we chose $\mathcal{L}_0$ to contain 60% of the alarms in $\mathcal{L}$. This value is based on informal experiments with different sizes of $\mathcal{L}_0$.

6.5 Cluster Tendency

Clustering algorithms (including CLARAty) will create clusters whether a data set contains clusters or it is purely random. In Sect. 6.4.2, we have seen that it is very difficult to validate clusters after the fact. It is therefore advisable to verify the existence of clusters before any clustering algorithm is applied to a data set. In that way, it is possible to prevent the inappropriate application of clustering methods. Acknowledging this best practice, we show in this section that intrusion detection alarms have – indeed – a natural tendency to form clusters. That does not imply that CLARAty discovers these clusters correctly, but it does imply that the application of clustering methods to intrusion detection alarms is justified.

The tool used to establish the existence of clusters is known as a *test of cluster tendency* [155, 161, 163, 170]. Such tests decide if a given data set $\mathcal{D}$ can reasonably be assumed to contain clusters, even though the clusters themselves are not identified. In a nutshell, tests of cluster tendency use a *test statistic* that measures in a single number the degree to which a data set contains clusters. Moreover, they determine the probability that a random data set scores the same or a better value in the test statistic. If this probability is negligible (say, smaller than 0.001), then the data set $\mathcal{D}$ is assumed to contain clusters. This, however, is no proof for the existence of clusters. In fact, there are limits to how well a single number can measure the existence of clusters. Moreover, even when the data set $\mathcal{D}$ scores a value in the test statistic that is highly unlikely for random data sets, this does not exclude the possibility that $\mathcal{D}$ is actually random and rid of any clusters. Thus, tests of cluster tendency are plausibility checks that offer corroborating evidence for the existence of clusters.

In Sect. 6.5.1, we describe the test of cluster tendency that we have designed to test the existence of clusters in intrusion detection alarms. In Sect. 6.5.2, we apply this test of cluster tendency to real-world alarms. Section 6.5.3 explains in more detail the mathematics that are used in the test of cluster tendency.

6.5.1 Test of Cluster Tendency

In Sect. 6.3.2, we said that CLARAty heuristically searches for large alarm clusters that are adequately modeled by generalized alarms. Hence, it is our

goal to experimentally validate the existence of such clusters. In other words, we want to validate the following hypothesis:

Definition 2 (Cluster Hypothesis). *Alarm logs generally contain large sets of alarms that are adequately modeled by generalized alarms.* □

To validate this hypothesis, we first derive a test statistic $\phi_p(\cdot)$ that maps alarm logs to integers. Specifically, for a given alarm log $\mathcal{L}$, the test statistic $\phi_p(\mathcal{L})$ measures how well $\mathcal{L}$ supports the cluster hypothesis. The proposed test statistic will return small values to indicate strong support. Unfortunately, there is no obvious threshold below which $\phi_p(\mathcal{L})$ is "small enough" to confirm the cluster hypothesis. Therefore, we proceed in analogy to all tests of cluster tendency, and define that $\phi_p(\mathcal{L})$ is "small enough" if a random alarm log has a negligible probability of scoring a ϕ_p-value that is equal to or smaller than $\phi_p(\mathcal{L})$.

We desire the test statistic $\phi_p(\mathcal{L})$ to measure whether the alarm log $\mathcal{L}$ contains – as predicted by the cluster hypothesis – large sets of alarms that are adequately modeled by generalized alarms. The problem with this formulation is that the terms "large" and "adequately" are too vague to be tested in a formal way.

We begin by making the meaning of "adequate" more precise. Recall that a generalized alarm adequately models a set of alarms if it correctly captures the key features of the set. Therefore, alarms themselves are the "most adequate" generalized alarms because they are maximally specific and do not sacrifice any information to the generalization of attribute values. However, alarms can only model sets of identical alarms. By contrast, the alarms that arise in the real world are generally mutually distinct. Therefore, alarms are too inflexible to model anything but the most trivial alarm sets. More flexibility is needed.

The following observation points the way to a more flexible but still "adequate" class of generalized alarms: For most alarms, moderate modifications of the source port value or the time-stamp value do not fundamentally change the alarm's semantic. In fact, the source port value is mostly set at random, and time is specified in units of seconds, even though a granularity of hours or less is generally sufficient. On the other hand, the source IP address, the destination IP address, the destination port, and the alarm type cannot be modified without substantially changing the meaning of an intrusion detection alarm. This motivates the following working definition:

Definition 3. *A generalized alarm is an* **adequate model** *of a set of alarms if it contains an exact (i.e., ungeneralized) value for the source IP address, the destination IP address, the destination port, and the alarm type, while permitting any arbitrary value for all other attributes.* □

This definition of adequacy is not the only one possible, but it certainly is a reasonable one. For example, let $\mathcal{G}$ be a set of alarms that can be modeled by the generalized alarm $\mathbf{g} \equiv ([Src\text{–}IP = 10.3.2.1] \wedge [Dst\text{–}IP = 10.3.2.2] \wedge$

$[Dst\text{–}port = 80] \wedge [Alarm\text{–}type = 10]$). (Attributes that $\mathbf{g}$ does not specify can assume arbitrary values.) If $\mathbf{g}$ is given, then we know with absolute precision the most important attribute values for the alarms in $\mathcal{G}$. For example, we know that all alarms in $\mathcal{G}$ have the source IP address 10.3.2.1 rather than, say, 10.3.2.0, which can make a big difference. It is this specificity with respect to the values of key attributes that makes $\mathbf{g}$ an adequate model for $\mathcal{G}$. For brevity, we call a generalized alarm *adequate* if it is an adequate model for a set of alarms.

To define the test statistic $\phi_p(\cdot)$, let $\mathcal{L}$ be an alarm log of size n, i.e., $n = |\mathcal{L}|$. Let $\mathcal{M}$ be the set of adequate generalized alarms that is obtained by projecting $\mathcal{L}$ on the four attributes source IP, destination IP, destination port, and alarm type, which characterize an adequate generalized alarm. Note that each alarm $\mathbf{a} \in \mathcal{L}$ matches a $\mathbf{g} \in \mathcal{M}$, and conversely, each $\mathbf{g} \in \mathcal{M}$ is matched by at least one $\mathbf{a} \in \mathcal{L}$. Let $m := |\mathcal{M}|$ denote the size of $\mathcal{M}$, let $\varphi(\mathbf{g})$, $\mathbf{g} \in \mathcal{M}$, be the number of alarms in $\mathcal{L}$ that match $\mathbf{g}$, and let the indices $i_1, \ldots, i_m$ be such that $\varphi(\mathbf{g_{i_1}}) \geq \varphi(\mathbf{g_{i_2}}) \geq \cdots \geq \varphi(\mathbf{g_{i_m}})$. For any fraction $p \in [0, 1]$, the test statistic $\phi_p(\mathcal{L})$ is defined as the smallest integer k for which $\sum_{z=1}^{k} \varphi(\mathbf{g_{i_z}}) \geq \lceil p \times n \rceil$ holds. Intuitively, the test statistic tells one that the $\phi_p(\mathcal{L})$ most frequently matched generalized alarms in $\mathcal{M}$ match at least $100 \times p$ percent of the alarms in $\mathcal{L}$.

Suppose that p is large (say, $p = 0.85$) and $\phi_p(\mathcal{L})$ is small in comparison to m. Then, the majority of alarms (namely, at least $\lceil p \times n \rceil$) match one out of a small set of $\phi_p(\mathcal{L})$ generalized alarms. It follows that on the average, each of these generalized alarms must be matched by $\lceil p \times n \rceil / \phi_p(L)$ alarms. Given our assumptions that p is large and $\phi_p(\mathcal{L})$ is small, we conclude that the quotient $\lceil p \times n \rceil / \phi_p(\mathcal{L})$ is large. As a consequence, each of the $\phi_p(\mathcal{L})$ generalized alarms models a *large* set of $\lceil p \times n \rceil / \phi_p(\mathcal{L})$ alarms on the average. For the alarm log $\mathcal{L}$, this implies that it consists of "large sets of alarms that are adequately modeled by generalized alarms". Hence, the alarm log $\mathcal{L}$ supports the cluster hypothesis.

This raises the need to decide in a quantitative manner when $\phi_p(\mathcal{L})$ is "small enough" to support the cluster hypothesis. As is typical for tests of cluster tendency, we decide that $\phi_p(\mathcal{L})$ is "small enough" if a random alarm log $\mathcal{L}'$ has a probability of at most 0.00001 to score a value $\phi_p(\mathcal{L}')$ that is equal to or smaller than $\phi_p(\mathcal{L})$. In other words, $\phi_p(\mathcal{L})$ is "small enough" if the condition $P[\phi_p(\mathcal{L}') \leq \phi_p(\mathcal{L}) \,|\, \mathcal{L}' \text{ is random}] \leq 0.00001$ holds. The threshold probability 0.00001 is arbitrary, and any other small probability could have been chosen. Random alarm logs must satisfy $|\mathcal{L}'| = |\mathcal{L}| = n$, and each alarm $\mathbf{a} \in \mathcal{L}'$ must have a $\mathbf{g} \in \mathcal{M}$, such that $\mathbf{a}$ matches $\mathbf{g}$. These requirements guarantee that the alarm logs $\mathcal{L}'$ and $\mathcal{L}$ are comparable [155, 170]. Conceptually, random alarm logs are obtained by repeating the following experiment n times: With all generalized alarms in $\mathcal{M}$ having the same probability, randomly choose one of them, generate an alarm that matches it, and add this alarm to $\mathcal{L}'$.

6.5.2 Experimental Setup and Results

While Sect. 6.5.1 has developed a test of cluster tendency, we will now apply it to real-world intrusion detection alarms. All the alarms we use are from network-based, commercial misuse detection systems[3] that were deployed in operational (i.e., "real-world") environments.

More specifically, Table 6.3 introduces the sixteen IDSs that we used in our experiments. The sixteen IDSs are deployed at eleven different Fortune 500 companies, and no two IDSs are deployed at the same geographic site. Moreover, for each IDS, we employ all alarms that were triggered during the year 2001. The "IDS" column contains a numerical identifier that we will use below to reference the IDSs. The "Type" column indicates the IDS type, namely, "A" or "B", both of which are leading commercial IDSs. To avoid unintended commercial implications, we do not reveal the product names or vendors of "A" and "B". The minimum, maximum, and average number of alarms per month are listed for each IDS in the "Min", "Max", and "Avg" columns, respectively. Finally, the "Location" column indicates where the IDSs are deployed: "Intranet" denotes an IDS on an internal corporate network without Internet access; "DMZ" denotes an IDS on a perimeter network that is protected by a firewall, but offers services to the Internet; "Extranet" denotes an IDS on a network that is shared between multiple cooperating companies, but is not accessible from the Internet; "Internet" denotes an IDS that is deployed before the external firewall on a direct link to the Internet.

For each IDS in Table 6.3 and for each month of the year 2001, we now apply the test developed in Sect. 6.5.1 to the alarm log $\mathcal{L}$ consisting of all alarms triggered by said IDS in said month. For example, the first IDS of Table 6.3 triggers $n = 42,018$ alarms in January 2001, and the set $\mathcal{M}$ consists of $m = 7,788$ generalized alarms. For $p := 0.85$, we obtain $\phi_p(\mathcal{L}) = 2,238$. In other words, using $2,238$ out of the $7,788$ generalized alarms in $\mathcal{M}$, it is possible to model 85% of the $42,018$ alarms in L. Moreover, a random alarm log $\mathcal{L}'$ satisfies $\phi_p(\mathcal{L}') \leq \phi_p(\mathcal{L})$ with a probability of less than 0.00001 (see Sect. 6.5.3 for a proof). Therefore, the cluster hypothesis of Definition 2 is supported by the alarm log that IDS-1 generates in January 2001. This is indicated by a tick in row 1 and column "Jan" of Table 6.4. The other entries of the table can be interpreted in the same way: Each IDS and month defines a separate alarm log, and a tick in the corresponding field indicates that the alarm log supports the cluster hypothesis. A dash, by contrast, stands for no support. In all experiments, p is set to 0.85, and the "IDS" column of Table 6.4 refers back to Table 6.3.

It follows from Table 6.4 that 165 out of 192 alarm logs confirm the cluster hypothesis. That offers strong evidence in favor of the cluster hypothesis. In experiments not documented here, we have shown that this result is robust

[3] A network-based misuse detection system is an IDS that sniffs network traffic and uses knowledge about attacks to detect instances of them in the network traffic it observes [135].

Table 6.3. Overview of IDSs used in experiments

IDS	Type	Location	Min	Max	Avg
1	A	Intranet	7,643	67,593	39,396
2	A	Intranet	28,585	1,946,200	270,907
3	A	DMZ	11,545	777,713	310,672
4	A	DMZ	21,445	1,302,832	358,735
5	A	DMZ	2,647	115,585	22,144
6	A	Extranet	82,328	719,677	196,079
7	A	Internet	4,006	43,773	20,178
8	A	Internet	10,762	266,845	62,289
9	A	Internet	91,861	257,138	152,904
10	B	Intranet	18,494	228,619	90,829
11	B	Intranet	28,768	977,040	292,294
12	B	DMZ	2,301	289,040	61,041
13	B	DMZ	3,078	201,056	91,260
14	B	Internet	14,781	1,453,892	174,734
15	B	Internet	248,145	1,279,507	668,154
16	B	Internet	7,563	634,662	168,299

with respect to variations in the definition of adequacy. Moreover, variations in the value of p do not fundamentally change the result, either. However, we have not experimented with other test statistics $\phi_p(\cdot)$ or other formalizations of the random log concept. Both could affect the results.

6.5.3 Derivation of Probabilities

The last section considered an alarm log $\mathcal{L}$ to support the cluster hypothesis if the probability $P[\phi_p(\mathcal{L}') \leq \phi_p(\mathcal{L}) \,|\, \mathcal{L}' \text{ is random}]$ is smaller than 0.00001. Here, we address the problem of calculating this probability. However, experience with similar problems [155, 170] suggests that it is very difficult to determine the exact value of the probability $P[\phi_p(\mathcal{L}') \leq \phi_p(\mathcal{L}) \,|\, \mathcal{L}' \text{ is random}]$. We will therefore overestimate this probability. Note that an overestimate makes us err at the expense of the cluster hypothesis. In other words, the number of ticks in Table 6.4 could only have increased if the exact probabilities had been used.

Let $\mathcal{L}$, $\mathcal{M}$, $n = |\mathcal{L}|$, $m = |\mathcal{M}|$, $p \in [0,1]$, and $\phi_p(\cdot)$ be as in Sect. 6.5.1, and set $k := \phi_p(\mathcal{L})$. Recall that a random alarm log is defined as the result of iterating the following experiment n times: Randomly choose a generalized alarm from $\mathcal{M}$, generate an alarm that matches this generalized alarm, and add this alarm to the random alarm log under construction. Note that all generalized alarms in $\mathcal{M}$ are equally likely to be chosen. These introductory remarks set the stage for the following proposition:

Table 6.4. Alarm logs that support the cluster hypothesis ($p = 0.85$)

IDS	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
1	√	√	√	√	√	√	√	√	√	√	√	√
2	√	√	√	√	√	√	√	√	√	√	√	√
3	√	–	–	–	–	–	–	–	–	–	–	–
4	√	√	√	√	√	√	√	√	√	√	√	√
5	√	√	√	√	√	√	√	√	√	√	√	√
6	√	√	√	–	–	√	√	√	√	√	√	√
7	√	√	√	√	√	√	√	√	√	√	√	–
8	√	√	√	√	√	–	–	–	√	√	√	√
9	√	√	√	√	–	√	√	√	√	√	–	–
10	√	√	√	√	√	√	√	√	√	√	√	√
11	√	√	√	√	√	√	√	√	√	√	√	√
12	√	√	√	√	√	√	√	√	√	√	√	√
13	√	√	√	√	√	√	√	√	√	√	√	√
14	–	–	–	√	√	–	√	–	√	–	√	–
15	√	√	√	√	√	√	√	√	√	√	√	√
16	√	√	√	√	√	√	√	√	√	√	√	√

Proposition 1. *Let n, m, p, $\phi_p(\cdot)$, and k be as above. For a random alarm log $\mathcal{L}'$, and for $\lambda := \lceil p \times n \rceil$, the following inequality holds:*

$$P[\phi_p(\mathcal{L}') \leq k] \;\leq\; \frac{1}{m^n} \cdot \binom{m}{k} \cdot \sum_{i=0}^{n-\lambda} \binom{n}{i} k^{n-i}(m-k)^i \;. \tag{6.1}$$

Proof. Attach an imaginary dartboard to each generalized alarm in $\mathcal{M}$ and imagine throwing n darts at the m dartboards. Suppose that we hit each dartboard with the same probability, namely, $1/m$. Then, $P[\phi_p(\mathcal{L}') \leq k]$ (which is the left-hand side of inequality (6.1)) equals the probability that after throwing all n darts, there are k dartboards that in total have λ or more darts sticking. We will use this more intuitive formulation of the problem, to prove inequality (6.1).

To generate all constellations where k dartboards have at least λ darts sticking, we can proceed as follows: Choose k dartboards and j darts, with $j = \lambda, \ldots, n$. There are $\binom{m}{k} \times \binom{n}{j}$ ways to do this. Moreover, there are $k^j \times (m-k)^{n-j}$ ways to throw the darts so that the selected j darts hit one out of the selected dartboards, whereas the remaining $n - j$ darts hit some nonselected dartboard. By summing over all j, we see that this process generates

$$\sum_{j=\lambda}^{n} \binom{m}{k} \binom{n}{j} k^j (m-k)^{n-j} \tag{6.2}$$

constellations. Note, however, that some constellations are generated multiple times. Equation (6.2) counts each constellation as many times as it is generated, and therefore overestimates the actual number of distinct constellations. It is easy to transform (6.2) into $\binom{m}{k} \cdot \sum_{i=0}^{n-\lambda} \binom{n}{i} k^{n-i}(m-k)^i$ (just move $\binom{m}{k}$ before the summation, substitute $j := n - i$, and observe that $\binom{n}{n-i} = \binom{n}{i}$). Finally, (6.1) is obtained by dividing this quantity by m^n, where m^n is the total number of ways to throw n darts at m dartboards. □

Two notes are in order. First, we researched ways to improve the bound given by Proposition 1, but the resulting formulas were complex and did not make a big difference in practice. Second, while calculating Table 6.4, there were nine instances where we found the estimate of (6.1) to be too coarse. In these instances, we used a Monte Carlo simulation to obtain a better estimate of the probability.

6.6 Conclusion

This chapter proposes and validates a new solution to the problem that intrusion detection systems overload their human operators by triggering thousands of mostly false alarms per day. Central to this solution is the notion of a root cause, which is the reason for which an alarm was triggered. Building on this notion, we proposed the following new alarm handling paradigm: First, CLARAty, a new clustering method, is used to identify root causes that account for large numbers of alarms, and second, these root causes are fixed or the processing of their associated alarms is automated (e.g., using a standard event correlation product).

In this chapter, we have seen many examples of root causes (see Sect. 6.2), we have studied how CLARAty works (see Sect. 6.3.2), and we have closely looked at a use case for how CLARAty supports root cause analysis in practice (see Sect. 6.3.3). Moreover, in Sect. 6.4.1 we have explained why all clustering results are a priori a bit suspect and should be validated. Section 6.4.2 explained the limitations of today's cluster validation techniques, and Sect. 6.4.3 showed how we used replication analysis to validate the alarm clusters found by CLARAty. Given the difficulty of validating clusters after the fact, we showed in Sect. 6.5 that intrusion detection alarms have, indeed, a natural tendency to cluster. Thus, by showing the existence of clusters, we provided further objective evidence that CLARAty produces meaningful results. This is further confirmed by our own (somewhat subjective) experience in using CLARAty for root cause analysis.

7

Behavioral Features for Network Anomaly Detection

James P. Early and Carla E. Brodley

7.1 Introduction

Research in network intrusion detection has traditionally been divided into two components – *misuse detection* and *anomaly detection*. The distinction between the two comes from the approaches used to build each component and the resulting alerts they generate. Misuse detection systems are built on knowledge of known attack behaviors. Alerts are generated when these attack behaviors (signatures) are identified within current network behavior. A misuse detection system must be periodically updated with new signatures in order to identify new attack types. In contrast, anomaly detection systems are built on knowledge of normal network behavior, and alerts are generated when deviations from normal behavior occur.

The principal use of anomaly detection methods in network intrusion detection is to identify novel attacks for which no attack signature exists. If we can ascertain what makes a particular anomalous event hostile, we can create a filter to block such events in the future. However, the key question arising from an anomalous event is, Is this a malicious or undesirable anomaly? For example, if an anomaly occurs in the context of a system that models behavior based on the number of connections from a particular host [171, 172], should the host be considered hostile? If the host belongs to a new customer placing orders on an e-commerce server, then clearly it is not. Deploying a filter to drop traffic from this "offending" host would negatively impact the business of the organization. This problem also exists when looking at anomalous combinations of values in packet headers. Trying to filter based on anomalous packet header values penalizes benign traffic that carries these values by coincidence, and does not provide any protection against an attacker (because the attacker can alter packet header values).

Anomaly detection systems are initially *trained* with a collection of examples of normal data. This data is referred to as the *training set*. Once trained, the system examines current behavior to identify instances that deviate from its model of normal behavior. An important design criterion is the choice of

features used to model normal behavior. Features provide semantics for the values in the data. For example, a feature can be the number of host connections or a value in a packet header. Feature extraction is important because it directly affects the accuracy and utility of the anomaly detection system. Good features provide strong correlation between anomalous behavior and malicious activity, whereas poorly selected features can result in irrelevant anomalies and high false-positive rates.

In this chapter, we present an approach to feature extraction for network anomaly detection based on protocol behavior. Such features help us identify when a party is not using a protocol as intended, or is using a protocol in a way that significantly deviates from the normal behavior of others using the protocol. Many common attacks depend on an improper use of a protocol. We refer to these types of attacks as *protocol anomalies*. Examples include Ping of Death [173], SYN Flood [174], and Teardrop [175]. Protocol anomalies are also used to conceal hostile activity from an intrusion detection system (IDS). For example, an attacker can partition traffic using overlapping IP fragments and TCP segments in an attempt to cause an IDS to interpret the traffic differently than the intended victim [176]. This discrepancy can result in a successful attack that is deemed benign by the IDS. By employing features that can identify these types of protocol anomalies, we can establish strong correlations between anomalous events and malicious behaviors. In this chapter, we describe our method of deriving these types of features and demonstrate its efficacy.

We now introduce terminology used throughout the remainder of this chapter. We will use the term *attribute* to refer to a particular characteristic of network traffic being measured. Examples of attributes include source and destination IP addresses, TCP ports, and ICMP message types. In essence, attributes are the names of values found in network packet header fields. A feature may relate directly to a single attribute, such as the Protocol field in an IP packet header. A feature may also be composed of multiple attributes, or constitute the sample average of a particular attribute over time.

A *protocol* specification can be considered a policy for interaction. The policy defines attributes and the range of acceptable values that are to be exchanged between parties. A given network protocol is likely to contain several types of attributes (e.g., fields in a packet header) – each used in different ways. These can identify the end points of the communication path, maintain the logical state of the connection, or carry information to verify the integrity of the connection. Identifying how an attribute is used in the protocol is an important step in determining whether the attribute should be used as a feature for anomaly detection.

Finally, we will use the term *flow* to describe a collection of information exchanged between entities engaged in the protocol. This information could be in the form of packets, sessions, or commands. Flows also have directionality – a typical protocol exchange consists of a client flow and corresponding server

flow. Examples of flows include a Telnet session, an HTTP request, and an ICMP message.

7.2 Inter-Flow versus Intra-Flow Analysis

In this section, we present our protocol analysis method for identifying the relative applicability of attributes as features for anomaly detection. We call this method *Inter-flow versus Intra-flow Analysis* or IVIA. The first step is to identify the protocol attributes that will be used to partition network traffic data into different flows and permit grouping of similar types of flows. For example, in TCP and UDP, a logical flow is indicated by the source and destination port values carried in the packet header [177, 178]. By partitioning the data on these attributes, we can examine all flows with a TCP source port value of 80 (an HTTP server flow) or a UDP source port value of 53 (a DNS Server), for example. The next step involves examining the remaining attributes to determine whether changes occur in the attribute's value between flows (inter-flow) and/or within a flow (intra-flow). Note that if attribute value changes are observed within a flow, the attribute value also exhibits changes among flows.

Table 7.1. Classification of protocol attributes based on observed changes in attribute values between flows (Inter-flow) and/or within flows (Intra-flow)

		Intra-flow Changes	
		Y	**N**
Inter-flow Changes	**Y**	I Operationally Variable	II Flow Descriptor
	N	III	IV Operationally Invariant

Table 7.1 shows a matrix of how different classes of attributes are organized using our analysis method. Attributes whose values do not change are assigned to Quadrant IV. Example attributes for this class include those that specify the name of the protocol, and those labeled as "reserved" by the protocol. We call the attributes in Quadrant IV *operationally invariant attributes*. Attributes in Quadrant II are designated to partition flows – for a given flow, all values for an attribute will be identical. Conversely, the values will likely be different when comparing different flows. Thus, we expect inter-flow changes in attribute values, but not intra-flow changes. We call these *logical flow attributes*. Examples include TCP and UDP port numbers, and IP source and destination addresses. Quadrant III will necessarily be empty because any changes observed intra-flow must also be visible inter-flow. Finally, Quadrant I contains attributes whose values change within flows. We refer to these as *operationally variable attributes*.

Table 7.2. IVIA method for the attributes of the IP Version 4 protocol

		Intra-flow Changes	
		Y	**N**
Inter-flow Changes	**Y**	I Header Length (IHL) Service type Total Length Identification Flags_DF Flags_MF Fragment Offset Time to Live Options	II Source Destination Protocol
	N	III	IV Version Flags_Reserved

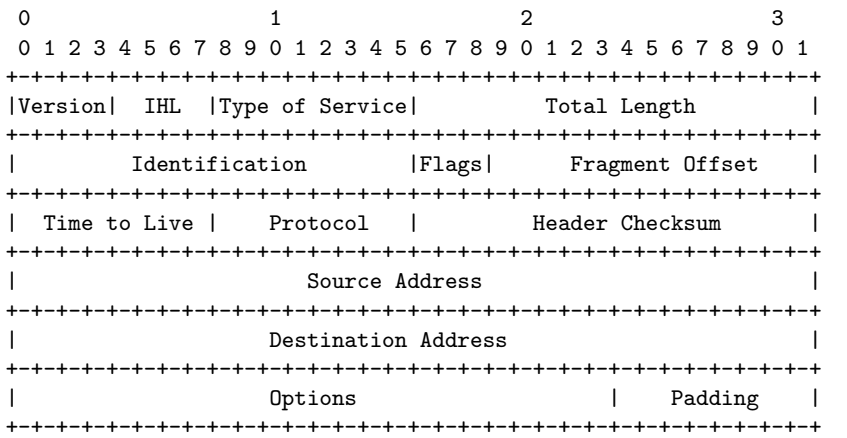


Fig. 7.1. IP Version 4 packet header

An example of the application of our analysis to the IP Version 4 protocol can be seen in Table 7.2. Figure 7.1 shows how the various protocol attributes are arranged in the packet header. The *Source* and *Destination* addresses, and *Protocol* are the logical flow descriptors that partition flows. These attributes are placed in Quadrant II. The protocol specifies that the *Version* and *Flags_Reserved* attributes have the same value for all flows, thus they are classified as operationally invariant attributes and placed in Quadrant IV. The remaining attributes are classified as operationally variable attributes and placed in Quadrant I, because changes in their values can be expected both within and among flows.[1] For example, the *Time to Live* attribute indicates the number of network "hops" the packet can traverse before it is discarded.

[1] Despite the fact that the *IP Checksum* attribute changes within flows, it was not included in Quadrant I because its value is derived from other values in the packet header.

Given that packets associated with a particular flow can traverse different network paths, the value may change within a flow from one packet to the next. A detailed description of the remaining attributes in this class can be found in [179].

The quadrant to which an attribute is assigned gives an indication of how particular attribute values might be enforced through policy. Firewalls are routinely used to filter based on attributes in Quadrants II and IV. In the case of IP Version 4, we selectively filter based on the source/destination/protocol of a particular flow. Similarly, we can also use a firewall to drop packets carrying a value other than 4 for the IP version. Quadrant assignment also identifies attributes where such filtering would disrupt the protocol: in particular, those attributes in Quadrant I. Again using IP Version 4 as an example, we cannot arbitrarily enforce certain values for the *Total Length* and *Identification* attributes without limiting the functionality of the protocol.

Quadrant assignment also provides some insight into the applicability of a given feature for anomaly detection. Attributes classified as operationally invariant (Quadrant IV) are likely to be very good features because they have only one acceptable value. Any other value should be viewed as suspect. Logical flow attributes (Quadrant II) may be useful for anomaly detection depending on whether policy statements can be made about normal and abnormal flows. A single home user may be able to identify a small set of acceptable hosts and protocols, but a network administrator for a large university may not. Attributes classified as operationally variable (Quadrant I) are of particular interest, because their values can vary significantly and still be considered normal in the context of protocol operation. In the next section, we will examine the implications of using these attributes as features for anomaly detection.

7.3 Operationally Variable Attributes

In this section, we will examine some practical issues arising from the use of operationally variable attributes as features for anomaly detection. The issues concern the size of the normal value space (and its implications for training data), and the use of these attributes in the context of data mining.

7.3.1 Size of Normal Value Space

Operationally variable attributes can be considered normal over their entire range. Therefore, there is little utility in focusing on discrete values of these attributes as indicators of normal or hostile behavior. If these attributes were used directly to construct an anomaly detection system, any combinations of values identified as anomalous in test data would themselves be normal with respect to the protocol specification. This means that the potential of this system for generating false alarms is quite large [180, 181].

To see why, let us assume IVIA on a given protocol has identified n operationally variable attributes labeled O_1 through O_n. The cardinality of a single attribute, $C(O_i)$, is the range of possible normal values for that attribute. Thus, the size of the space of normal values for an instance of the protocol is:

$$\prod_{i=1}^{n} C(O_i) \; . \tag{7.1}$$

By definition, every combination of values in this space is normal with respect to the protocol specification. If we were to build an anomaly detection system for these attributes, we would need an example for every value combination to completely cover the value space. Any combination excluded from training would likely result in a false positive when later encountered by the anomaly detection system.

To understand the scope of this requirement, let us return to the example of IVIA on the IP Version 4 protocol. We identified nine operationally variable attributes – *Header Length, Service Type, Total Length, Identification, DF Flag, MF Flag, Fragment Offset, Time to Live,* and *Options*. The cardinality for each of these attributes ranges from 2 to over 65 thousand.[2] The size of the normal value space for these attributes is 2^{67}, or approximately $1.5\text{E}10^{20}$. If we assume it takes a human operator an average of one second to collect and verify a training instance for every combination (a very fast operator!), it would take over $4.7\text{E}10^{12}$ person-*years* to complete the training set.

Certain combinations of operationally variable attribute values can be violations of the protocol. In IP Version 4, a datagram whose *Total Length* is less than the *Header Length* is invalid. Other combinations may reveal implementation errors for a particular system. For example, the Land denial of service (DoS) attack [182] caused some network stacks to fail by using a packet with the same value for both the source and destination IP address. Although some of these combinations may be useful for anomaly detection, the number of such combinations is likely to be small in comparison to the size of the overall value space. Using anomaly detection to identify these combinations creates the potential for unacceptably high false-positive rates. It is more reasonable to expect that such combinations can identified by studying the protocol specification.

7.3.2 Data Mining on Operationally Variable Attributes

As was seen in the previous section, the collection of training data using operationally variable attributes presents significant practical problems. We conducted the following experiment to illustrate the drawbacks of approaches

[2] *Options* is a variable-length attribute. We assumed the options field is not used for our calculation. If *Options* were used, the size of the value space is 2^{99} or roughly $6.3\text{E}10^{29}$.

that learn a model based on operationally variable attributes. In this experiment, our goal was to build a model of normal IP fragments. This can be seen as a subset of a larger data mining exercise wherein models for multiple protocol behaviors are simultaneously constructed.

We collected approximately 5,400 examples of normal fragments from the 1999 DARPA IDS Evaluation Data Sets [183]. The operationally variable attributes for the IP Version 4 protocol (as shown in Quadrant I of Table 7.2) were then extracted from each packet and used as features. With the training set constructed, we used Magnum Opus [184] to create a collection of association rules. Association rules are used to identify frequent relationships among attribute values in the data [58, 185]. They are often used to identify buying patterns of shoppers (e.g., those shoppers who bought bread and milk who also bought cereal). In the context of network traffic, association rules relate to values in the packet header frequently occurring together. This technique has been used to develop a set of rules describing normal traffic [186, 187]. Anomaly alerts are generated when a packet fails to match a sufficient number of these rules.

Table 7.3. Association rules generated for IP fragments in the 1999 DARPA IDS Evaluation data sets using operationally variable attributes as features

Rule	Support	Strength
ipFlagsMF = 1 & ipTTL = 63 → ipTLen = 28	0.526	0.981
ipID < 2817 & ipFlagsMF = 1 → ipTLen > 28	0.309	0.968
ipID < 2817 & ipTTL > 63 → ipTLen > 28	0.299	1.000
ipTLen > 28 → ipID < 2817	0.309	1.000
ipID < 2817 → ipTLen > 28	0.309	0.927
ipTTL > 63 → ipTLen > 28	0.299	0.988
ipTLen > 28 → ipTTL > 63	0.299	0.967
ipTLen > 28 & ipOffset > 118 → ipTTL > 63	0.291	1.000

Table 7.3 shows the top eight ranking rules as generated by Magnum Opus. The rules are sorted by two metrics – *support* and *strength*. The support metric refers to the percentage of items in the training set whose attributes match the left- and right-hand side of the rule. The strength metric indicates the probability of a match of the right-hand side (RHS) of the rule given a match of the attributes on the left-hand side (LHS). A more common name for this metric is *confidence*, and we will use the more common term in the remainder of this chapter.

Using the first rule as an example, we see that roughly 53% of the training instances match the attribute/value combinations in the rule. This rule states that fragments with the MF flag set (indicating it is not the last fragment) and a TTL value of 63 are 98% likely to have a total length of 28 bytes. However, suppose a fragment has a TTL value of 62 or a total length of 29 bytes. These

values are perfectly valid in the context of the protocol specification, but they would not match the above rule. Here we see the first problem associated with using operationally variable attributes as features. Thresholds for distinctions between normal and abnormal values are determined by the supplied training data, but can be arbitrary with respect to the protocol specification.

Looking at the remaining rules in Table 7.3, we see a number of operationally variable attributes appearing as relevant features for describing normal behavior. Examples include *ipID* and *ipTTL* fields. The implication is that the values associated with these attributes can be useful in distinguishing normal from abnormal fragments. Unfortunately, this is not the case. The *ipID* field is merely a label indicating the IP datagram to which the fragment belongs. The *ipTTL* field can be any nonzero value. Here we see another problem associated with using operationally variable attributes as features for anomaly detection. Features found to be significant in the training data are often not useful in discriminating malicious from non-malicious events.

If an anomaly detection system were based on the above rules, an alert would be generated if an incoming IP fragment did not match any of the rules. However, it is likely that many normal IP fragments would fail to match these rules and thus result in a false-positive alert. Therefore, we conclude that the features used to model IP fragments, the operationally variable attributes, are not able to adequately model the distinction between normal and abnormal fragments.

In conclusion, we avoid the direct use of operationally variable attributes in our models, because (1) they do not have an inherent notion of anomalous values, (2) the collection of adequate training data is time-consuming and human-intensive, and (3) in the absence of adequate training data, the resulting anomaly detection system has the potential for high false-positive rates. In the next section, we show how these attributes can be transformed into useful features for anomaly detection.

7.4 Deriving Behavioral Features

We have shown the practical problems associated with using operationally variable attributes as features for anomaly detection. In this section, we present a method to transform operationally variable attributes into features that better capture the notion of anomalous *behavior* of a protocol. To accomplish this, we focus on how the values of a given attribute change during operation. This allows us to better distinguish between normal and abnormal protocol usage than by observing the attribute values directly.

By definition, the values associated with operationally variable attributes change within a flow. Therefore, our approach examines *how* the attribute values change over time or an event sequence window. Revisiting the *Time to Live* attribute of the IP Version 4 protocol as an example, we can observe the

range of unique values (i.e., entropy) of this attribute over fixed-length packet window. Additional observations for other such attributes include:

- Mean and standard deviation of an attribute value
- Percentage of events with a given attribute value
- Percentage of events representing monotonically increasing (or decreasing) values
- Step size of an attribute value

This is not an exhaustive list, because other useful measurements arise when examining particular protocols in detail. Some examples include the percentage of TCP packets using the PSH flag, the range of values for the IP Identification field, and the average step size of the IP Fragment Offset. In addition, certain measurements among events have been found to be useful, such as the mean inter-arrival time of packets.

We refer to an attribute observed over time/events as a *behavioral feature*. Using behavioral features, we can now compare how different entities use a given protocol. Training data requirements are reduced because we do not have the large normal value space associated with operationally variable attributes. The definition of *normal* comes from the uses of the protocol deemed acceptable. In the next section, we will describe an application of behavioral features.

7.5 Authentication Using Behavioral Features

In this section, we present an approach to classify server flow behavior using decision trees based on behavioral features. Traditional methods of determining traffic type rely on the port label carried in the packet header. This method can fail, however, in the presence of proxy servers that remap port numbers or host services that have been compromised to act as backdoors or covert channels. Because our classification of the traffic type is independent of port label, it provides a more accurate classification in the presence of malicious activity. An empirical evaluation illustrates that models of both aggregate protocol behavior and host-specific protocol behavior obtain classification accuracies ranging from 82 to 100%.

7.5.1 The Need for Authentication of Server Flows

Understanding the nature of the information flowing into and out of a system or network is fundamental to determining if there is adherence to a usage policy. The traditional method of determining the client-server protocol is by inspecting the source and destination port numbers in the TCP header. The mappings between port numbers and their corresponding service are well known [188]. In essence, we rely on a correct *labeling* of the traffic to accurately

determine its type. The binding between the port label and the type of traffic is a *binding by convention*. This label is also used as a basis for firewall filtering and intrusion detection [172, 189].

There are several attack scenarios for which the port number may not be indicative of the true nature of the server traffic.

Malicious Proxies: Systems used to circumvent firewall rules [190]. The proxy takes traffic that would normally be dropped by the firewall and remaps the port numbers to make the traffic appear legitimate (e.g., HTTP traffic).

Server Backdoors: When a server has been compromised, the attacker often places a "backdoor" in the form of an altered executable. This new executable functions as both rogue service R for the attacker and authorized service S for all other users.

User-Installed Servers: This category includes the installation of unauthorized servers such as FTP, HTTP, peer-to-peer file sharing [191], and rogue mail servers [192]. The user may or may not be aware that the server is running. These servers can configured to use almost any port number.

Each of these scenarios represents an instance where the port number label fails to accurately indicate the type of traffic. Worse yet, it is precisely these scenarios where an accurate identification of the traffic would reveal a compromised service or policy violation. Thus, there exists a need to classify traffic associated with a particular service, what we will henceforth refer to as a *server flow*, using a method other than a mere label that is easily modified, ambiguous, or conceals unauthorized activity.

7.5.2 Classification of Server Flows

Server authentication can be naturally cast as either a supervised learning task or an anomaly detection task. To cast the problem as a supervised learning problem, we must choose k possible server applications, collect training data for each, and then apply a supervised learning algorithm to form a classifier. Given a new server flow, we can then classify it as one of these k types of servers. To cast the problem as an anomaly detection problem, we look at each service individually. For each of the k server applications of interest we form a model of normal behavior. Given a new server flow, we compare the new flow to each of the models to determine whether it conforms to any of these models. Casting the problem as an anomaly detection problem uses the same framework as user behavioral authentication [85]. In user authentication, the goal is to identify whether the user is behaving normally with respect to a learned profile of behavior.

We have chosen to investigate server flow authentication based on the supervised learning framework, because we assume a policy exists specifying the services that are to be run on a given host. A drawback of this assumption

is that if an attacker replaces or alters an existing service it may not behave like any of the permitted services, and this may not be readily detectable. However, it is unlikely that it will behave *identically* to any of the permitted services, but we plan to examine this conjecture in future work.

7.5.3 An Empirical Evaluation

Our experiments are designed to investigate whether we can classify server flows based on a set of behavioral features. The following features were extracted from a variable length packet window – the percentage of packets with each of the six TCP state flags set [177], the mean inter-arrival time, and the mean packet length. We next describe the data used in the experiments and the supervised learning algorithm we chose. Finally, we present experimental results with learning aggregate flows and by-host flows using both synthetic and real network traffic.

The first data set chosen for our experiments is the 1999 MIT Lincoln Labs Intrusion Detection Evaluation Data Sets [183]. Although created for a specific evaluation exercise, these data sets have subsequently been widely used for research into other later intrusion detection systems not part of the original evaluation [187, 193, 194].

7.5.4 Aggregate Server Flow Model

Our first experiment was designed to determine the extent to which FTP, SSH, Telnet, SMTP, and HTTP traffic can be differentiated using a decision-tree classifier. We used the data from week one of the Lincoln Labs data to build our training data set. The set was created by first randomly selecting fifty server flows for each of the five protocols. Each server flow consists of the packets from a server to a particular client host/port. The largest flow contained roughly $37,000$ packets, and the smallest flow contained 5 packets. The 250 flows represented a total of approximately $290,000$ packets. We refer to this as an *aggregate model* because the collection of flows came from many different servers.

The fact that this data is certified as attack-free meant that we could have confidence in the port numbers as indicative of the type of traffic. We used the server port to label each of the flows in the training set. Each server flow was then used to generate data observations based on our feature set. The result is a data set consisting of approximately $290,000$ labeled observations. We repeated this process for each of seven packet window sizes. The window size is an upper bound on the number of packets used to compute the means and percentages. If an individual flow contains fewer packets than the packet window size, the number of available packets is used to calculate each observation.

Each of the seven training sets was then used to build a decision tree using C5 [53] – a widely used and tested decision-tree algorithm implementation.

We constructed test sets in the same manner – fifty server flows from each protocol were randomly selected from week three of the Lincoln Labs data (also certified as attack-free). These were then passed to our feature extraction algorithm using the same seven window sizes.

```
tcpPerFIN > 0.01:
:...tcpPerPSH <= 0.4: www (45)
:   tcpPerPSH > 0.4:
:   :...tcpPerPSH <= 0.797619: smtp (13)
:       tcpPerPSH > 0.797619: ftp (38)
tcpPerFIN <= 0.01:
:...meanIAT > 546773.2:
    :...tcpPerSYN <= 0.03225806: telnet (6090)
    :   tcpPerSYN > 0.03225806:
    :   :...meanipTLen > 73.33: ftp (21)
    :       meanipTLen <= 73.33:
    :       :...tcpPerPSH > 0.7945206: smtp (8)
```

Fig. 7.2. Portion of a decision tree generated by C5

Table 7.4. Classification accuracy of the aggregate model decision trees on unseen individual server flows. Each value represents the percentage of correctly classified flows out of the fifty flows for each protocol

Window Size	FTP	SSH	Telnet	SMTP	WWW
1,000	100%	88%	94%	82%	100%
500	100%	96%	94%	86%	100%
200	98%	96%	96%	84%	98%
100	100%	96%	96%	86%	100%
50	98%	96%	96%	82%	100%
20	100%	98%	98%	82%	98%
10	100%	100%	100%	82%	98%

Before describing how a tree is used to classify a flow, we give an example of a portion of a decision tree generated by C5 in Fig. 7.2. In this example, the root node tests the percentage of packets in the packet window with the FIN flag set (tcpPerFIN). If this percentage exceeds 1%, a test is made on the percentage of packets with the PSH flag set (tcpPerPSH). If this value is less than or equal to 40%, the observation is classified as "www", indicating HTTP traffic. The numbers in parentheses indicate the number of training observations classified with this leaf node. Other tests can be seen involving the mean inter-arrival time (meanIAT) and mean packet length (meanipTLen).

During testing, the class label for a given flow was calculated by summing the confidence values for each observation in the flow. The class with the highest total confidence was assigned to that flow. The classification results are

shown in Table 7.4. For each of seven window sizes, we report the percentage of correctly classified server flows out of the set of fifty flows for each protocol. As can be seen in the table, the classification accuracy ranges from 82% to 100%.

In general, the classification accuracy was lower for SMTP server flows than for other protocols. We examined the misclassified flows in more detail and discovered that these flows were generally 2–4 times longer than correctly classified flows. Longer SMTP server flows represented longer periods of interaction, and thus contain increasing numbers of observations classified as Telnet or FTP. In these few cases, our feature set is not adequate for discriminating between the behaviors of these flows.

It is more desirable to use a smaller window size because this decreases the time to detect that a service is behaving abnormally. Indeed, for SSH, we see that too large a packet window size $(1,000)$ hurts classification accuracy. For FTP, SSH, and Telnet, a window size as small as ten packets achieves 100% classification accuracy.

We conclude from our experimental results that the behavior of server flows for the five protocols can be differentiated using a decision-tree classifier built on aggregate flows. We will later discuss how this method can be used to complement an intrusion detection system.

7.5.5 Host-Specific Models

Our second experiment addresses whether creating models for specific hosts provides better performance than the aggregate model. There are three advantages to using host-specific models:

1. By creating models for individual server flows, we can monitor these flows for changes in behavior.
2. A host-specific model can capture the implementation subtleties of a particular service running on a host. This resolution is missing in the aggregate model consisting of many server flows.
3. The training examples in an aggregate model will be dominated by the server generating the most traffic. This may dilute examples from other servers. The host-specific model solves this problem.

We first identified a set of hosts in the Lincoln Labs data that each ran three or more server protocols. Training data for each host was collected by randomly selecting server flows from week one for each of the protocols running on these hosts. The number of flows used in each model was chosen such that each protocol was represented by the same number of flows. Table 7.5 lists the number of training and test flows per host.

Based on our results using the aggregate models, we chose a packet window size of 100 for generating observations. The selection was driven by the fact that SMTP accuracy was greatest using this window size with the aggregate

Table 7.5. Number of flows used for each protocol in training and test sets for each host model

Host	Training Flows	Test Flows
172.16.112.100	20	20
172.16.112.50	30	25
172.16.113.50	35	23
172.16.114.50	10	20
197.218.177.69	25	35

Table 7.6. Classification accuracy of host model decision trees on unseen server flows. Each row reports the host address and the percentage of correctly classified flows for each protocol. Fields with a "–" indicate there was no traffic of this protocol type for this host

Host	FTP	SSH	Telnet	SMTP	WWW
172.16.112.100	95%	–	100%	90%	100%
172.16.112.50	92%	100%	84%	100%	–
172.16.113.50	100%	–	100%	100%	–
172.16.114.50	100%	95%	100%	95%	95%
197.218.177.69	100%	–	100%	100%	–

models, and other protocol classification accuracies were between 96% and 100%. We then trained a decision tree for each host that could be used to differentiate the server flows coming from that host. Test data was collected from week three in the same manner as the training data.

The results in Table 7.6 indicate that, in general, the host-specific models achieve approximately the same classification accuracy as the aggregate models. One difference observed is that classification accuracy varies by protocol. For example, the classification accuracy of Telnet flows for host *172.16.112.50* is 84% whereas the classification of Telnet flows in the aggregate models averaged 96.2%. Examination of the packets in the misclassified Telnet flows revealed an interesting phenomenon. We often observed large time gaps between packets. The time gaps indicate lapses in user activity where the Telnet server is not echoing characters or supplying responses to commands. In our framework, a single large gap can radically alter the values for the mean interarrival time of packets, thus resulting in misclassification of the subsequent observations. We refer to this as the *Water Cooler Effect* – the user temporarily leaves the interactive session, then resumes it a short while later. We are investigating the sensitivity of our classifiers to this effect. One possible solution would be to subdivide flows based on some time gap threshold and use the interactive sub-flows to build our classifiers.

7.5.6 Models from Real Network Traffic

In this section, we present experiments with real network traffic. We collected a number of server flows using the protocols described. We augmented this

set to include flows from hosts acting as Kazaa servers. Kazaa [191, 195] is a peer-to-peer file sharing system that is growing in popularity [196, 197]. Peer-to-peer network traffic was not part of the Lincoln Labs data set.

Our goal was to determine if there was a significant difference in classification accuracy when using synthetic versus real traffic. We observed classification accuracies by protocol ranging from 85% to 100% for both the aggregate and host models. The peer-to-peer traffic was classified correctly for 100% of the unseen flows. This is an especially interesting result because Kazaa flows carry a port label that is user-defined. Thus, we are able to correctly classify peer-to-peer flows behaviorally – without the use of the port number. These results indicate that our classification method is effective for real network traffic. The range of accuracies match those observed with the synthetic data. Thus, we can identify no appreciable difference in the per-flow behavior in the synthetic Lincoln Labs data versus those in real network traffic.

7.5.7 Classification for Intrusion and Misuse Detection

The focus for the use of behavioral features is the creation of behavior models that are highly useful in practice. To this end, no such work is complete without a discussion of its operational use. The two types of classification models presented here give rise to new functionality in the context of intrusion and misuse detection. Aggregate models try to classify a flow based on the general behavior of many flows of a given type. The question the aggregate model tries to answer is, What other flows does this flow resemble? In contrast, host models are based on the previously observed behavior of flows for a specific host. Given an unseen flow, the host models try to answer the question, Does this flow resemble previous server flows from this host?

Intrusion/misuse detection systems and firewalls try to identify actions a priori as being harmful to the system or network. An IDS may passively monitor traffic and alert in the presence of some attack condition. Firewalls actively drop network packets that violate some network policy. Our classification method attempts to identify activities indicative of intrusion or misuse *after* such an event has occurred. Working in concert with a priori mechanisms, we can attempt to determine at any moment in time whether there is an impending attack or artifacts of a successful attack.

Figure 7.3 shows how our classification methods can be integrated into a network with an existing IDS. The organization uses servers to provide network services (internally, externally, or both) to some community of users. Our host-flow classification system monitors the output of these servers directly. The purpose is to determine if currently observed flows continue to behave as expected. If an attacker manages to take control of a particular service, he or she will need to interact with the server in such a way as to exactly match the previous behavior.

The network carries additional user traffic to servers that are external to the organization. This traffic is monitored with the aggregate model. Here,

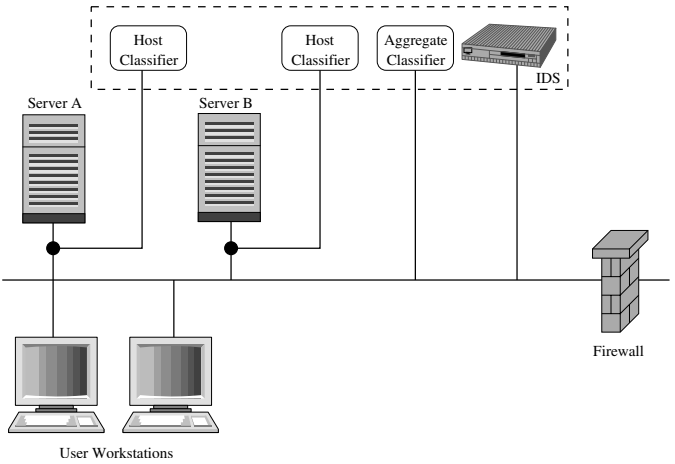


Fig. 7.3. Network placement of the host and aggregate classifiers. Host classifiers monitor specific server flows (outbound), while aggregate classifiers monitor user traffic (inbound)

we classify the flow generally and compare this to the port label. Observation of traffic that resembles Telnet to some nonstandard server port may be an indication of an installed backdoor. Traffic labeled as Web traffic (with a server port of 80) that behaves more like Telnet traffic may indicate the presence of a proxy used to evade firewall rules. A peer-to-peer client operating at some user-defined port may be a violation of the network policy. In each of these cases, the aggregate classifier can indicate if a given flow behaves in a manner consistent with its port label.

7.6 Related Work

Introduced by Anderson [198] and formalized by Denning [199], anomaly detection has been used in an attempt to identify novel attack behaviors. However, two questions posed by Denning remain unanswered to this day:

> *Soundness of Approach* – Does the approach actually detect intrusions? Is it possible to distinguish anomalies related to intrusions from those related to other factors?
>
> *Choice of Metrics, Statistical Models, and Profiles* – What metrics, models, and profiles provide the best discriminating power? Which are cost-effective? What are the relationships between certain types of anomalies and different methods of intrusion?

Both of these questions relate to the types of attributes, and ultimately, to the features used for anomaly detection. These questions can be rephrased as,

What should be measured in order to model normal behavior and subsequently identify new types of intrusions with high accuracy? Choosing a feature with good predictive power for inclusion in a behavior model is as important as avoiding the choice of a likely irrelevant feature.

A variety of statistical and data mining methods have been applied to attributes of network traffic in an attempt to identify attack behaviors. These have used three methods for feature extraction: (1) direct use of packet attributes (e.g., operationally variable, invariant, and flow attributes), (2) manually crafted features, and (3) automated feature extraction.

The Packet Header Anomaly Detector (PHAD) [194] is an example of the use of machine learning techniques to identify intrusions using attributes of network packets. An anomalous packet is indicated by its "rarity" in the training data. Training and testing of this system was performed using the 1998 Lincoln Labs Intrusion Detection Evaluation Data Sets [200]. The authors identified an artifact in this data set wherein the inclusion of the IP TTL (Time to Live) attribute in the feature set increased intrusion detection rates by more than 30%. This was caused by the use of a small set of TTL values for injected attacks. The authors correctly reasoned that TTL values would be unlikely to identify attacks in real network traffic. However, such is also the case for other operationally variable attributes. This illustrates a problem associated with the use of operationally variable attributes (i.e., IP TTL) as features. Such features contribute to irrelevant rules and point to inadequacies in training data.

A number of systems have employed manually crafted features that attempt to model aspects of connection behavior. These features include the number of connections between hosts [172] and the number of bytes transferred between hosts [171, 201] in a given time frame. Models based on such features are focused on certain types of attacks, such as worm propagation. As such, these features may not provide the descriptive power needed for a broad collection of attack behaviors.

In [122, 181], Lee and Stolfo presented a process of using data mining to identify those features that are most relevant to detecting attack behaviors. The mining techniques are performed on high-level network events called *network connection records*. Each record is an aggregate of TCP/IP connection information containing a combination of packet attributes (e.g., source and destination addresses and port numbers) and crafted features (e.g., a fragment behavior flag and Land attack flag) as their base set of features. Further, they define a set of *axis attributes* used to uniquely identify connections. These are comparable to the flow attributes we use to organize flows. The technique relies on a comprehensive set of initial features and adequate training data to obtain useful results. In [122], the authors note:

> There are also much needed improvements to our current approach; First, deciding upon the right set of features is difficult and time-

consuming. For example, many trials were attempted before we came up with the current set of features and time intervals.

The work presented in this chapter provides a method to aid in the construction of the initial feature set. Further, behavioral features provide finer granularity than features based on high-level events. Thus, we view the use of behavioral features and the a priori identification of likely irrelevant features (i.e., operationally variable attributes) as an opportunity to enhance this and other feature selection techniques.

7.7 Conclusion

In this chapter, we presented our method for feature extraction based on the interpretation of protocol specifications. Our method identifies protocol attributes whose values frequently change within the context of normal operation and, therefore, do not make useful features for anomaly detection. We show that the inclusion of these operationally variable attributes as features complicates the process of collecting training data and can yield anomaly detection models that are not useful in practice.

We also show how these attributes can be transformed into behavioral features by examining how the values of these attributes change during operation. An application of behavioral features was presented showing that a collection of TCP application protocols can be differentiated. Using models of normal application behavior, we can confirm that a given server flow is behaving in a manner consistent with its port label.

8

Cost-Sensitive Modeling for Intrusion Detection

Wenke Lee, Wei Fan, Salvatore J. Stolfo, and Matthew Miller

Summary. A real-time intrusion detection system (IDS) has several performance objectives: good detection coverage, economy in resource usage, resilience to stress, and resistance to attacks upon itself. A cost-based modeling approach can be used to consider the trade-offs of these IDS performance objectives in terms of cost and value.

In this paper, we study the problem of building cost-sensitive models for real-time IDS. We first discuss the major cost factors in IDS, including consequential and operational costs. We propose a multiple model cost-sensitive machine learning technique to produce models that are optimized for user-defined cost metrics. Empirical experiments in off-line analysis show a reduction of approximately 97% in operational cost over a single model approach, and a reduction of approximately 30% in consequential cost over a pure accuracy-based approach.

8.1 Introduction

Intrusion detection (ID) is an important component of infrastructure protection mechanisms. Many intrusion detection systems (IDSs) are emerging in the marketplace, following research and development efforts in the past two decades. They are, however, far from the ideal security solution for customers. Investment in IDSs should bring the highest possible benefit and maximize user-defined security goals while minimizing costs. This requires ID models to be sensitive to cost factors. Currently these cost factors are ignored as unwanted complexities in the development process of IDSs.

We developed a data mining framework for building intrusion detection models. It uses data mining algorithms to compute activity patterns and extract predictive features, and applies machine learning algorithms to generate detection rules [202]. In this paper, we describe our research in extending our data mining framework to build cost-sensitive models for intrusion detection. We briefly examine the relevant cost factors, models, and metrics related to IDSs. We propose a multiple model, cost-sensitive machine learning technique that can automatically construct detection models optimized for given cost

metrics. Our models are learned from training data which was acquired from an environment similar to one in which a real-time detection tool may be deployed. Our data consists of network connection records processed from raw `tcpdump` [203] files using MADAM ID (a system for Mining Audit Data for Automated Models for Intrusion Detection) [202].

The rest of the paper is organized as follows: Section 8.2 examines major cost factors related to IDSs and outlines problems inherent in modeling and measuring the relationships among these factors. Section 8.3 describes our multiple model approach to reducing operational cost and a MetaCost [204] procedure for reducing damage cost and response cost. In Sect. 8.4, we evaluate this proposed approach using the 1998 DARPA Intrusion Detection Evaluation data set. Section 8.5 reviews related work in cost-sensitive learning and discusses extensions of our approach to other domains and machine learning algorithms.

8.2 Cost Factors, Models, and Metrics in IDSs

8.2.1 Cost Factors

There are three major cost factors involved in the deployment of an IDS. Damage cost, *DCost*, characterizes the maximum amount of damage inflicted by an attack when intrusion detection is unavailable or completely ineffective. Response cost, *RCost*, is the cost to take action when a potential intrusion is detected. Consequential cost, *CCost*, is the total cost caused by a connection and includes DCost and RCost as described in detail in Sect. 8.2.2. The operational cost, *OpCost*, is the (computational) cost inherent in running an IDS.

8.2.2 Cost Models

The cost model of an IDS formulates the total expected cost of the IDS. In this paper, we consider a simple approach in which a prediction made by a given model will always result in some action being taken. We examine the cumulative cost associated with each of these outcomes: false negative (FN), false positive (FP), true positive (TP), true negative (TN), and misclassified hits. The costs associated with these outcomes are known as *consequential costs* (CCost), and are outlined in Table 8.1.

FN Cost is the cost of not detecting an intrusion. It is therefore defined as the damage cost associated with the particular type of intrusion i_t, $\mathrm{DCost}(i_t)$.

TP Cost is the cost incurred when an intrusion is detected and some action is taken. We assume that the IDS acts quickly enough to prevent the damage of the detected intrusion, and therefore only pays $\mathrm{RCost}(i_t)$.

FP Cost is the cost incurred when an IDS falsely classifies a normal connection as intrusive. In this case, a response will ensue, and we therefore pay $\mathrm{RCost}(i)$, where i is the detected intrusion.

TN Cost is always 0, as we are not penalized for correct normal classification.

Misclassified Hit Cost is the cost incurred when one intrusion is incorrectly classified as a different intrusion – when i is detected instead of i_t. We take a pessimistic approach that our action will not prevent the damage of the intrusion at all. Since this simplified model assumes that we always respond to a predicted intrusion, we also include the response cost of the detected intrusion, RCost(i).

Table 8.1. Consequential Cost (CCost) matrix. c is the connection, i_t is the true class, and i is the predicted class

Outcome	CCost(c)
Miss (FN)	DCost(i_t)
False Alarm (FP)	RCost(i)
Hit (TP)	RCost(i_t)
Normal (TN)	0
Misclassified Hit	RCost(i) + DCost(i_t)

8.2.3 Cost Metrics

Cost-sensitive models can only be constructed and evaluated using given cost metrics. Qualitative analysis is applied to measure the relative magnitudes of the cost factors, as it is difficult to reduce all factors to a common unit of measurement (such as dollars). We have thus chosen to measure and minimize CCost and OpCost in two orthogonal dimensions.

An intrusion taxonomy must be used to determine the damage and response cost metrics which are used in the formulation of CCost. A more detailed study of these cost metrics can be found in our ongoing work [205]. Our taxonomy is the same as that used in the DARPA evaluation, and consists of four types of intrusions: probing (PRB), denial of service (DoS), remotely gaining illegal local access (R2L), and a user gaining illegal root access (U2R). All attacks in the same category are assumed to have the same DCost and RCost. The relative scale or metrics chosen are shown in Table 8.2a.

The operational cost of running an IDS is derived from an analysis of the computational cost of computing the features required for evaluating classification rules. Based on this computational cost and the added complexity of extracting and constructing predictive features from network audit data, features are categorized into three relative levels. Level 1 features are computed using at most the first three packets of a connection. Level 2 features are computed in the middle of or near the end of a connection using information of the current connection only. Level 3 features are computed using information from all connections within a given time window of the current

Table 8.2. Cost metrics. (**a**) Intrusion classes. (**b**) Feature categories

(**a**)

Category	DCost	RCost
U2R	100	40
R2L	50	40
DoS	20	20
PRB	2	20
Normal	0	0

(**b**)

Category	OpCost
Level 1	1 or 5
Level 2	10
Level 3	100

connection. Relative magnitudes are assigned to these features to represent the different computational costs as measured in a prototype system we have developed using Network Flight Recorder (NFR) [206]. These costs are shown in Table 8.2b. The cost metrics chosen incorporate the computational cost as well as the availability delay of these features. It is important to note that level 1 and level 2 features must be computed individually. However, because all level 3 features require iteration through the entire set of connections in a given time window, all level 3 features can be computed at the same time, in a single iteration. This saves operational cost when multiple level 3 features are computed for analysis of a given connection.

8.3 Cost-Sensitive Modeling

In the previous section, we discussed the consequential and operational costs involved in deploying an IDS. We now explain our cost-sensitive machine learning methods for reducing these costs.

8.3.1 Reducing Operational Cost

In order to reduce the operational cost of an IDS, the detection rules need to use low-cost features as often as possible while maintaining a desired accuracy level. Our approach is to build multiple rule sets, each of which uses features from different cost levels. Low-cost rules are always evaluated first by the IDS, and high-cost rules are used only when low-cost rules cannot predict with sufficient accuracy. We propose a multiple rule set approach based on RIPPER, a popular rule induction algorithm [51].

Before discussing the details of our approach, it is necessary to outline the advantages and disadvantages of two major forms of rule sets that RIPPER computes, *ordered* and *unordered*. An ordered rule set has the form **if** $rule_1$ **then** $intrusion_1$ **elseif** $rule_2$ **then** $intrusion_2$, ..., **else** *normal*. To generate an ordered rule set, RIPPER sorts class labels according to their frequency in the training data. The first rule classifies the most infrequent class, and the end of the rule set signifies prediction of the most frequent (or default) class, *normal*, for all previously unpredicted instances. An ordered rule set

is usually succinct and efficient, and there is no rule generated for the most frequent class. Evaluation of an entire ordered rule set does not require each rule to be tested, but proceeds from the top of the rule set to the bottom until any rule evaluates to *true*. The features used by each rule can be computed one by one as evaluation proceeds. An unordered rule set, on the other hand, has at least one rule for each class and there are usually many rules for frequently occurring classes. There is also a default class which is used for prediction when none of these rules are satisfied. Unlike ordered rule sets, all rules are evaluated during prediction and all features used in the rule set must be computed before evaluation. Ties are broken by using the most accurate rule. Unordered rule sets are less efficient in execution, but there are usually several rules of varying precision for the most frequent class, *normal*. Some of these *normal* rules are usually more accurate than the default rule for the equivalent ordered rule set.

With the advantages and disadvantages of ordered and unordered rule sets in mind, we propose the following multiple rule set approach:

- We first generate multiple training sets T_{1-4} using different feature subsets. T_1 uses only cost 1 features. T_2 uses features of costs 1 and 5, and so forth, up to T_4, which uses all available features.
- Rule sets R_{1-4} are learned using their respective training sets. R_4 is learned as an ordered rule set for its efficiency, as it may contain the most costly features. R_{1-3} are learned as unordered rule sets, as they will contain accurate rules for classifying *normal* connections.
- A *precision* measurement p_r[1] is computed for *every rule*, r, except for the rules in R_4.
- A threshold value τ_i is obtained for every single class, and determines the tolerable precision required in order for a classification to be made by any rule set except for R_4.

In real-time execution, the feature computation and rule evaluation proceed as follows:

- All cost 1 features used in R_1 are computed for the connection being examined. R_1 is then evaluated and a prediction i is made.
- If $p_r > \tau_i$, the prediction i will be fired. In this case, no more features will be computed and the system will examine the next connection. Otherwise, additional features required by R_2 are computed and R_2 will be evaluated in the same manner as R_1.
- Evaluation will continue with R_3, followed by R_4, until a prediction is made.
- When R_4 (an ordered rule set) is reached, it computes features as needed while evaluation proceeds from the top of the rule set to the bottom. The

[1] Precision describes how accurate a prediction is. Precision is defined as $p = \frac{|P \cap W|}{|P|}$, where P is the set of predictions with label i, and W is the set of all instances with label i in the data set.

evaluation of R_4 does not require any firing condition and will always generate a prediction.

The OpCost for a connection is the total computational cost of all unique features used before a prediction is made. If any level 3 features (of cost 100) are used at all, the cost is counted only once since all level 3 features are calculated in one function call.

This evaluation scheme is further motivation for our choice of learning R_{1-3} as unordered rule sets. If R_{1-3} were learned as ordered rule sets, a *normal* connection could not be predicted until R_4 since the default *normal* rules of these rule sets would be less accurate than the default rule of R_4. OpCost is thus reduced, resulting in greater system throughput, by only using low-cost features to predict normal connections.

The precision and threshold values can be obtained during model training from either the training set or a separate hold-out validation set. Threshold values are set to the precisions of R_4 on that data set. Precision of a rule can be obtained easily from the positive, p, and negative, n, counts of a rule, $\frac{p}{p+n}$. The threshold value will, on average, ensure that the predictions emitted by the first three rule sets are not less accurate than using R_4 as the only hypothesis.

8.3.2 Reducing Consequential Cost

We applied the MetaCost algorithm, introduced by Domingos [204], to reduce CCost. MetaCost relabels the training set according to the cost-matrix and decision boundaries of RIPPER. Instances of intrusions with $DCost(i) < RCost(i)$ or a low probability of being learned correctly will be relabeled as *normal*.

8.4 Experiments

8.4.1 Design

Our experiments use data that was distributed by the 1998 DARPA evaluation, which was conducted by MIT Lincoln Lab [207]. The data was gathered from a military network with a wide variety of intrusions injected into the network over a period of 7 weeks. The data was then processed into connection records using MADAM ID [202]. The processed records are available from the UCI KDD Archive as the 1999 KDD Cup Data Set [87]. A 10% sample was taken which maintained the same distribution of intrusions and normal connections as the original data.[2] We used 80% of this sample as training

[2] The full data set is around 743M. It is very difficult to process and learn over the complete data set in a reasonable amount of time with limited resources given the fact that RIPPER is memory-based and MetaCost must learn multiple bagging models to estimate probabilities.

data. For infrequent intrusions in the training data, those connections were repeatedly injected to prevent the learning algorithm from neglecting them as statistically insignificant and not generating rules for them. For overwhelmingly frequent intrusions, only 1 out of 20 records were included in the training data. This is an ad hoc approach, but produced a reasonable rule set. The remaining 20% of our sample data was left unaltered and used as test data for evaluation of learned models. Table 8.3 shows the different intrusions present in the data, the category within our taxonomy that each belongs to, and their sampling rates in the training data.

Table 8.3. Intrusions, categories, and sampling

U2R		R2L		DoS		PRB	
buffer_overflow	1	ftp_write	4	back	1	ipsweep	1
loadmodule	2	guess_passwd	1	land	1	nmap	1
multihop	6	imap	2	neptune	$\frac{1}{20}$	portsweep	1
perl	6	phf	3	pod	1	satan	1
rootkit	2	spy	8	smurf	$\frac{1}{20}$		
		warezclient	1	teardrop	1		
		warezmaster	1				

We used the training set to calculate the precision for each rule and the threshold value for each class label. We experimented with the use of a hold-out validation set to calculate precisions and thresholds. The results (not shown) are similar to those reported below.

8.4.2 Measurements

We measure expected operational and consequential costs in our experiments. The expected OpCost over all occurrences of each connection class and the average OpCost per connection over the entire test set are defined as $\frac{\sum_{c \in S_i} OpCost(c)}{|S_i|}$ and $\frac{\sum_{c \in S} OpCost(c)}{|S|}$, respectively, where S is the entire test set, i is a connection class, and S_i represents all occurrences of i in S. In all of our reported results, $OpCost(c)$ is computed as the sum of the feature computation costs of all unique features used by all rules evaluated until a prediction is made for connection c. CCost is computed as the cumulative sum of the cost matrix entries, defined in Table 8.1, for all predictions made over the test set.

8.4.3 Results

In all discussion of our results, including all tables, "RIPPER" is the single model learned over the original data set, "Multi-RIPPER" is the respective multiple model, "MetaCost" is the single model learned using RIPPER with a

MetaCost relabeled data set, and "Multi-MetaCost" is the respective multiple model.

Table 8.4. Average OpCost per connection class

IDS	RIPPER	Multi-RIPPER	MetaCost	Multi-MetaCost
back	223	143	191	1
buffer_overflow	172	125.8	175	91.6
ftp_write	172	113	146	71.25
guess_passwd	198.36	143	191	87
imap	172	107.17	181	108.08
ipsweep	222.98	100.17	191	1
land	132	2	191	1
loadmodule	155.33	104.78	168.78	87
multihop	183.43	118.43	182.43	100.14
neptune	223	100	191	1
nmap	217	119.63	191	1
normal	222.99	111.14	190.99	4.99
perl	142	143	151	87
phf	21	143	191	1
pod	223	23	191	1
portsweep	223	117.721	191	1
rootkit	162	100.7	155	63.5
satan	223	102.84	191	1
smurf	223	143	191	1
spy	131	100	191	46.5
teardrop	223	23	191	1
warezclient	223	140.72	191	86.98
warezmaster	89.4	48.6	191	87

Table 8.5. Average OpCost per connection

	RIPPER	Multi-RIPPER	MetaCost	Multi-MetaCost
OpCost	222.73	110.64	190.93	5.78

As shown in Table 8.5, the average OpCost per connection of the single MetaCost model is 191, while the Multi-MetaCost model has an average OpCost of 5.78. This is equivalent to the cost of computing only a few level 1 features per connection and offers a reduction of 97% from the single rule set approach. The single MetaCost model is 33 times more expensive. This means that in practice we can classify most connections by examining the first three packets of the connection at most 6 times. Additional comparison shows that the average OpCost of the Multi-RIPPER model is approximately half as much as that of the single RIPPER model. This significant reduction

by Multi-MetaCost is due to the fact that R_{1-3} accurately filter out *normal* connections (including low-cost intrusions relabeled as *normal*), and a majority of connections in real network environments are *normal*. Our multiple model approach thus computes more costly features only when they are needed to detect intrusions with $DCost > RCost$. Table 8.4 lists the detailed average OpCost for each connection class. It is important to note that the difference in OpCost between RIPPER and MetaCost models is explainable by the fact that MetaCost models do not contain (possibly costly) rules to classify intrusions with $DCost < RCost$.

Table 8.6. CCost and error rate

	RIPPER	Multi-RIPPER	MetaCost	Multi-MetaCost
CCost	42,026	41,850	29,866	28,026
Error	0.0847%	0.1318%	8.24%	7.23%

Our CCost measurements are shown in Table 8.6. As expected, both MetaCost and Multi-MetaCost models yield a significant reduction in CCost over RIPPER and Multi-RIPPER models. These reductions are both approximately 30%. The consequential costs of the Multi-MetaCost and Multi-RIPPER models are also slightly lower than those of the single MetaCost and RIPPER models.

The detailed precision and TP rates[3] of all four models are shown in Table 8.7 for different connection classes. The values for the single classifier and multiple classifier methods are very close to each other. This shows that the coverage of the multiple classifier methods are identical to those of the respective single classifier methods. It is interesting to point out that MetaCost fails to detect *warezclient*, but Multi-MetaCost is highly accurate. The reason is that R_4 completely ignores all occurrences of *warezclient* and classifies them as *normal*.

The error rates of all four models are also shown in Table 8.6. The error rates of MetaCost and Multi-MetaCost are much higher than those of RIPPER and Multi-RIPPER. This is because many intrusions with $DCost < RCost$ are relabeled as *normal* by the MetaCost procedure. Multi-RIPPER misclassified such intrusions more often than RIPPER, which results in its slightly lower CCost and slightly higher error rate. Multi-MetaCost classifies more intrusions correctly (*warezclient*, for example) and has a lower CCost and error rate than MetaCost.

[3] Unlike precision, TP rate describes the fraction of occurrences of a connection class that were correctly labeled. Using the same notation as in the definition of precision, $TP = \frac{P \cap W}{W}$.

Table 8.7. Precision and recall for each connection class

		RIPPER	Multi-RIPPER	MetaCost	Multi-MetaCost
back	*TP*	1.0	1.0	0.0	0.0
	p	1.0	1.0	na	na
buffer_overflow	*TP*	1.0	1.0	0.8	0.6
	p	1.0	1.0	0.67	0.75
ftp_write	*TP*	1.0	0.88	0.25	0.25
	p	1.0	1.0	1.0	1.0
guess_passwd	*TP*	0.91	0.91	0.0	0.0
	p	1.0	1.0	na	na
imap	*TP*	1.0	0.83	1.0	0.92
	p	1.0	1.0	1.0	1.0
ipsweep	*TP*	0.99	0.99	0.0	0.0
	p	1.0	1.0	na	na
land	*TP*	1.0	1.0	0.0	0.0
	p	1.0	1.0	na	na
loadmodule	*TP*	1.0	1.0	0.44	0.67
	p	0.9	1.0	1.0	1.0
multihop	*TP*	1.0	1.0	1.0	0.86
	p	0.88	0.88	0.88	1.0
neptune	*TP*	1.0	1.0	na	na
	p	1.0	1.0	na	na
nmap	*TP*	1.0	1.0	0.0	0.0
	p	1.0	1.0	na	na
normal	*TP*	0.99	0.99	0.99	0.99
	p	0.99	0.99	0.92	0.93
perl	*TP*	1.0	1.0	1.0	1.0
	p	1.0	1.0	1.0	1.0
phf	*TP*	1.0	1.0	0.0	0.0
	p	1.0	1.0	na	na
pod	*TP*	1.0	1.0	0.0	0.0
	p	0.98	0.98	na	na
portsweep	*TP*	0.99	0.99	0.0	0.0
	p	1.0	1.0	na	na
rootkit	*TP*	1.0	0.6	0.5	0.2
	p	0.77	1.0	0.83	1.0
satan	*TP*	1.0	0.98	0.0	0.0
	p	0.99	0.99	na	na
smurf	*TP*	1.0	1.0	0.0	0.0
	p	1.0	1.0	na	na
spy	*TP*	1.0	1.0	0.0	0.0
	p	1.0	1.0	na	na
teardrop	*TP*	1.0	1.0	0.0	0.0
	p	1.0	1.0	na	na
warezclient	*TP*	0.99	0.99	0.0	0.9
	p	1.0	1.0	na	1.0
warezmaster	*TP*	0.6	0.6	0.0	0.0
	p	1.0	1.0	na	na

Table 8.8. Comparison with fcs-RIPPER

	Multi-MetaCost	MetaCost	fcs-RIPPER									
			$\omega = .1$	.2	.3	.4	.5	.6	.7	.8	.9	1.0
OpCost	5.78	191	151	171	191	181	181	161	161	171	171	171

8.4.4 Comparison with fcs-RIPPER

In previous work, we introduced a feature cost-sensitive method, fcs-RIPPER, to reduce OpCost [205, 208]. This method favors less costly features when constructing a rule set. Cost sensitivity is controlled by the variable $\omega \in [0, 1]$ and sensitivity increases with the value of ω. We generated a single ordered rule set using different values of ω with fcs-RIPPER. In Table 8.8, we compare the average OpCost over the entire test set for the proposed multiple classifier method with that of fcs-RIPPER. We see that fcs-RIPPER reduces the operational cost by approximately 10%, whereas Multi-MetaCost reduces this value by approximately 97%. The expected cost of Multi-MetaCost is approximately 30 times lower than that of fcs-RIPPER, RIPPER, and MetaCost. This difference is significant.

8.5 Related Work

Much research has been done in cost-sensitive learning, as indicated by Turney's online bibliography [209]. Within the subset of this research which focuses on multiple models, Chan and Stolfo proposed a meta-learning approach to reduce consequential cost in credit card fraud detection [210]. MetaCost is another approach which uses bagging to estimate probabilities. Fan et al. proposed a variant of AdaBoost for misclassification cost-sensitive learning [211]. Within research on feature-cost-sensitive learning, Lavrac et al. applied a hybrid genetic algorithm effective for feature elimination [212].

Credit card fraud detection, cellular phone fraud detection, and medical diagnosis are related to intrusion detection because they deal with detecting abnormal behavior, are motivated by cost-saving, and thus use cost-sensitive modeling techniques. Our multiple model approach is not limited to IDSs and is applicable in these domains as well.

In our study, we chose to use an inductive rule learner, RIPPER. However, the multiple model approach is not restricted to this learning method and can be applied to any algorithm that outputs a precision along with its prediction.

8.6 Conclusion and Future Work

Our results using a multiple model approach on off-line network traffic analysis show significant improvements in both operational cost (a reduction of 97%

over a single monolithic model) and consequential costs (a reduction of 30% over accuracy-based model). The operational cost of our proposed multiple model approach is significantly lower than that of our previously proposed fcs-RIPPER approach. However, it is desirable to implement this multiple model approach in a real-time IDS to get a practical measure of its performance. Since the average operational cost is close to computing at most six level 1 features, we expect efficient real-time performance. The moral of the story is that computing a number of specialized models that are accurate and cost-effective for particular subclasses is demonstrably better than building one monolithic ID model.

9

Data Cleaning and Enriched Representations for Anomaly Detection in System Calls

Gaurav Tandon, Philip Chan, and Debasis Mitra

9.1 Introduction

Computer security research has two major aspects: intrusion prevention and intrusion detection. While the former deals with preventing the occurrence of an attack (using authentication and encryption techniques), the latter focuses on the detection of successful breach of security. Together, these complementary approaches assist in creating a more secure system.

Intrusion detection systems (IDSs) are generally categorized as misuse-based and anomaly-based. In misuse (signature) detection, systems are modeled upon known attack patterns and the test data is checked for occurrence of these patterns. Examples of signature-based systems include virus detectors that use known virus signatures and alert the user when the system has been infected by the same virus. Such systems have a high degree of accuracy but suffer from the inability to detect novel attacks. Anomaly-based intrusion detection [199] models normal behavior of applications and significant deviations from this behavior are considered anomalous. Anomaly detection systems can detect novel attacks but also generate false alarms since not all anomalies are hostile. Intrusion detection systems can also be categorized as network-based, which monitors network traffic, and host-based, where operating system events are monitored.

There are two focal issues that need to be addressed for a host-based anomaly detection system: cleaning the training data, and devising an enriched representation for the model(s). Both these issues try to improve the performance of an anomaly detection system in their own ways. First, all the proposed techniques that monitor system call sequences rely on *clean* training data to build their model. The current audit sequence is then examined for anomalous behavior using some supervised learning algorithm. An attack embedded inside the training data would result in an erroneous model, since all future occurrences of the attack would be treated as normal. Moreover, obtaining clean data by hand could be tedious. Purging all malicious content from audit data using an automated technique is hence imperative.

Second, normal behavior has to be modeled using features extracted from the training set. It is important to remember that the concept of normalcy/abnormality in anomaly detection is vague as compared to a virus detector which has an exact signature of the virus it is trying to detect, making anomaly detection a hard problem. Traditional host-based anomaly detection systems focus on system call sequences to build models of normal application behavior. These techniques are based upon the observation that a malicious activity results in an abnormal (novel) sequence of system calls. Recent research [213, 214] has shown that sequence-based systems can be compromised by conducting mimicry attacks. Such attacks are possible by astute execution of the exploit by inserting dummy system calls with invalid arguments such that they form a legitimate sequence of events, thereby evading the IDS. A drawback of sequence-based approaches lies in their non-utilization of other key attributes, namely, the system call arguments. The efficacy of such systems might be improved upon if a richer set of attributes (return value, error status and other arguments) associated with a system call is used to create the model.

In this chapter, we address the issues of data cleaning and anomaly detection, both of which essentially try to detect outliers, but differ in character:

1. Off-line vs. online techniques. We present two enriched representations: (a) motifs and their locations are used for cleaning the data (an off-line procedure) whereas (b) system call arguments are modeled for online anomaly detection.
2. Supervised algorithms assume no attacks in the training data. Unsupervised algorithms, on the other hand, relax this constraint and could have small amounts of unlabeled attacks. We present two modified detection algorithms: Local Outlier Factor or LOF (unsupervised) and LEarning Rules for Anomaly Detection or LERAD (supervised).
3. Low false-alarm rates are critical in anomaly detection and desirable in data cleaning. False alarms are generated in anomaly detection systems as not all anomalies are representative of attacks. For an online detection system, a human expert has to deal with the false alarms and this could be overwhelming if in excess. But for data cleaning, we would like to retain generic application behavior to provide a clean data set. Rendering the data free of attacks is highly critical and some other non-attack abnormalities may also get removed in the process. Purging such anomalies (program faults, system crashes among others) is hence justifiable, if still within reasonable limits. Thus the evaluation criteria vary in the two cases.

Empirical results indicate that our technique is effective in purging anomalies in unlabeled data. Our representation can be effectively used to detect malicious sequences from the data using unsupervised learning techniques. The filtered training data leads to better application modeling and an enhanced performance (in terms of the number of detections) for online anomaly

detection systems. Our system does not depend on the user for any parameter values, as is the norm for most of the anomaly detection systems. Our sequence- and argument-based representations also result in better application modeling and help detect more attacks than the conventional sequence-based techniques.

This chapter is organized as follows. Section 9.2 reviews some prevalent anomaly detection systems. In Sect. 9.3, we present the concept of motifs (in the context of system call sequences) and motif-based representations for data cleaning. Section 9.4 presents the argument-based representations and supervised learning algorithm for anomaly detection. An experimental evaluation is presented in Sect. 9.5, and we conclude in Sect. 9.6.

9.2 Related Work

Traditional host-based anomaly detection techniques create models of normal behavioral patterns and then look for deviations in test data. Such techniques perform supervised learning. Forrest et al. [215] memorized normal system call sequences using look-ahead pairs (tide). Lane and Brodley [216, 217] examined UNIX command sequences to capture normal user profiles using a fixed size window. Later work (stide and t-stide) by Warrender et al. [218] extended sequence modeling by using n-grams and their frequency. Wespi et al. [219, 220] proposed a scheme with variable length patterns using Teiresias [221], a pattern discovery algorithm in biological sequences. Ghosh and Schwartzbard [222] used artificial neural networks, Sekar et al. [223] proposed a finite state automaton, Jiang et al. [224] also proposed variable length patterns, Liao and Vemuri [225] used text categorization techniques, Jones and Li [226] learned temporal signatures, Coull et al. [227] suggested sequence alignment, Mazeroff et al. [228] proposed probabilistic suffix trees, and Lee at al. [229] used a machine learning algorithm called RIPPER [51] to learn normal user behavior. All these techniques require *clean* or labeled training data to build models of normal behavior, which is hard to obtain. The data sets used are synthetic and generated in constrained environments. They are not representative of actual application behavior, which contains many irregularities. The need for a system to filter audit data and produce a *clean* data set motivates our current research.

Unsupervised learning is an extensively researched topic in network anomaly detection [230–233]. Network traffic comprises continuous and discrete attributes which can be considered along different dimensions of a feature space. Distance and density-based algorithms can then be applied on this feature space to detect outliers. Due to the lack of a similar feature space, not much work has been done using unsupervised learning techniques in host-based systems.

From the modeling/detection point of view, all the above-mentioned approaches for host-based systems use system call sequences. Parameter effec-

tiveness for window-based techniques has been studied in [214]. Given some knowledge about the system being used, attackers can devise some methodologies to evade such intrusion detection systems. Wagner and Soto [213] modeled a malicious sequence by adding *no-ops* (i.e., system calls having no effect) to compromise an IDS based upon the sequence of system calls. Such attacks would be detected if the system call arguments are also taken into consideration, and this provides the motivation for our work.

9.3 Data Cleaning

The goal is to represent sequences in a single feature space and refine the data set off-line by purging anomalies using an unsupervised learning technique on the feature space.

9.3.1 Representation with Motifs and Their Locations

Let Σ be a finite set of all distinct system calls. A system call sequence (*SCS*) s is defined as a finite sequence of system calls and is represented as $(c_1 c_2 c_3 \cdots c_n)$, where $c_i \in \Sigma, 1 \leq i \leq n$.

After processing the audit data into process executions, system call sequences are obtained as finite length strings. Each system call is then mapped to a unique symbol using a translation table. Thereafter, they are ranked by utilizing prior knowledge as to how susceptible the system call is to malicious usage. A ranking scheme similar to the one proposed by Bernaschi et al. [234] was used to classify system calls on the basis of their threat levels.

Motifs and Motif Extraction

A *motif* is defined as a subsequence of length greater than p if it appears more than k times, for positive integers p and k, within the finite set $S = \{s_1, s_2, \ldots, s_m\}$ comprising m *SCS*s. Motif discovery has been an active area of research in bioinformatics, where interesting patterns in amino and nucleic acid sequences are studied. Since motifs provide a higher level of abstraction than individual system calls, they are important in modeling system call sequences. Two sets of motifs are extracted via *auto-match* and *cross-match*, explained next.

The set of motifs obtained through auto-match comprise frequently occurring patterns within each sequence. For our experiments, we considered any pattern at least 2 characters long, occurring more than once as frequent. While the set of *SCS*s S is the input to this algorithm, a set of unique motifs $M = \{m_1, m_2, \ldots, m_q\}$ is the output. It may happen that a shorter subsequence is subsumed by a longer one. We prune the smaller motif only if it is not more frequent than a larger motif that subsumes it. More formally, a

motif extracted using auto-match (1) has length ≥ 2, (2) has frequency ≥ 2, and (3) if there exists a motif $m_j \in M$ in a sequence $s_k \in S$ such that m_i is a subsequence of m_j but occurs independently in *SCS* s_k.

To illustrate this idea, consider the following synthetic sequence

$$acggcggfgjcggfgjxyz\,. \tag{9.1}$$

Note that in this sequence we have a motif *cgg* with frequency 3, and another motif *cggf* with frequency 2, which is longer and sometimes subsumes the shorter motif but not always. We consider them as two different motifs since the frequency of the shorter motif was higher than the longer one. The frequently occurring subsequences (with their respective frequency) are *cg* (3), *gg* (3), *gf* (2), *fg* (2), *gj* (2), *cgg* (3), *cggf* (2), *ggfg* (2), *gfgj* (2), *cggfg* (2), *ggfgj* (2), and *cggfgj* (2). The longest pattern, *cggfgj*, subsumes all the smaller subsequences except *cg*, *gg*, and *cgg*, since they are more frequent than the longer pattern, implying independent occurrence. But *cg* and *gg* are subsumed by *cgg*, since they all have the same frequency. Thus, the final set of motifs $M = \{cgg, cggfgj\}$.

Apart from frequently occurring patterns, we are also interested in patterns which do not occur frequently but are present in more than one *SCS*. These motifs could be instrumental in modeling an intrusion detection system since they reflect common behavioral patterns across sequences (benign as well as intrusive). We performed pair-wise cross-match between different sequences to obtain these. In other words, motif m_i extracted using cross-match (1) has length ≥ 2, (2) appears in at least a pair of sequences $s_k, s_l \in S$, and (3) is maximal, i.e., there does not exist a motif $m_j \in M (j \neq i)$ such that $m_j \subseteq s_k, s_l$ and $m_i \subset m_j$. Let us consider the following pair of synthetic sequences:

$$acgfgjcgfgjxyzcg\,, \tag{9.2}$$

$$cgfgjpqrxyzpqr\,. \tag{9.3}$$

Using cross-match between the example sequences (9.2) and (9.3), we get the motifs *cgfgj* and *xyz*, since these are the maximal common subsequences across the two given sequences.

A simple method for comparing amino acid and nucleotide sequences called the *Matrix Method* is described by Gibbs and McIntyre [235]. A matrix is formed with one sequence written across and the other in the downward position on the left of the matrix. Any common element was marked with a dot and a series of dots along a diagonal gave a common subsequence between the two sequences. Using a technique similar to the Matrix Method, motifs are extracted which occur across sequences but may not be frequent within a single sequence itself.

Motifs obtained for a sequence (auto-match) or pairs of sequences (cross-match) are added to the motif database. Redundant motifs are removed. Motifs are then ordered based upon the likelihood of being involved in an attack.

The ranking for individual system calls is used here and motifs are ordered using dictionary sort. The motifs are then assigned a unique ID based upon their position within the ordered motif database.

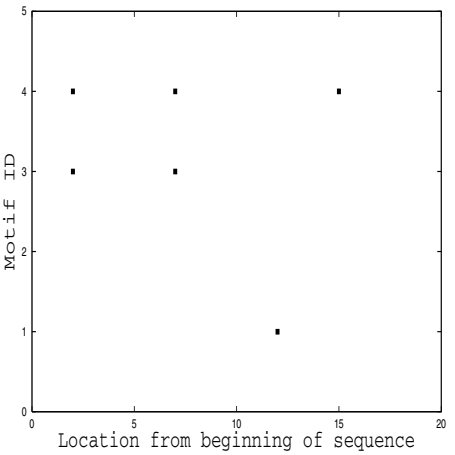


Fig. 9.1. Motif-oriented representation for sequence (9.2)

Motif-Based Representation of a Sequence

After collecting all the motifs that exist in the set S of sequences in the motif database M, we would like to represent each sequence in terms of the motifs occurring within it. For each sequence $s_i \in S$, we list all the motifs occurring within it along with their starting positions within the sequence.

This creates a two-dimensional representation for each *SCS* s_i, where the x-axis is the distance along the sequence from its beginning, and the y-axis is the motif ID of those motifs present in s_i. A sequence can thus be visualized as a scatter plot of the motifs present in the sequence. Figure 9.1 depicts such a representation for the synthetic sequence (9.2), where the motifs *cg*, *cgfgj*, and *xyz* are represented at the positions of occurrence within the respective sequence. A total of 4 unique motifs (i.e., *cg*, *cgfgj*, *pqr*, and *xyz*), obtained from auto-match and cross-match of sequences (9.2) and (9.3), are assumed in the motif database for the plot in Fig. 9.1. At the end of this phase, our system stores each *SCS* as a list of all motifs present within along with their spatial positions from the beginning of the sequence.

All the *SCS*s are modeled based upon the contained motifs. Malicious activity results in alterations in the *SCS* which is reflected by the variations in the motifs and their spatial positions. Plotting all the *SCS*s (based upon their motif-based representations) in a single feature space could reflect the similarity/dissimilarity between them.

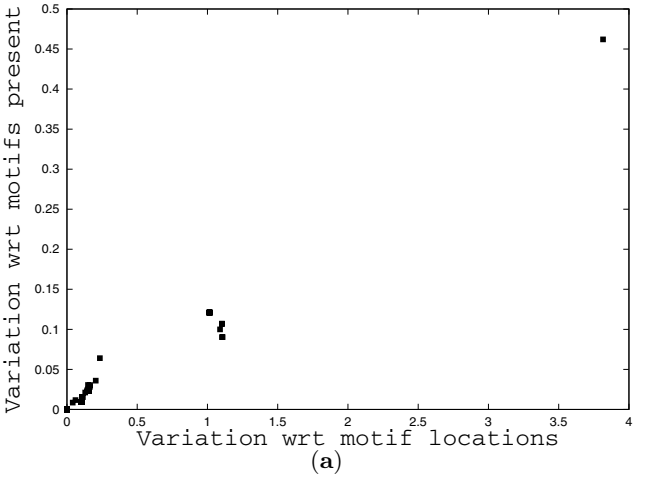


Fig. 9.2. Sequence space for two applications. (**a**) ftpd. (**b**) lpr

A Single Representation for Multiple Sequences

After creating a motif-based representation for each sequence, all the test sequences S are plotted in a feature space called the *sequence space*. In this representation, we measure the distance between pairs of *SCS*s along each of the two axes (motifs and their locations). Utilizing one (arbitrarily chosen) *SCS* from the set S as a reference sequence s_1, we measure (d_x, d_y) distances for all *SCS*s. Thus, the sequences are represented as points in this 2D sequence space, where the sequence s_1 is at the origin (reference point) on this plot. Let s_2 be any other sequence in S whose relative position with respect to

s_1 is to be computed. Let x_{1_i} (x_{2_i}) be the position of the ith motif in s_1 (s_2). Inspired by the symmetric Mahalanobis distance [236], the distance is computed as follows:

$$d_x = \frac{\frac{\sum_{i=1}^{n_1}(x_{1_i} - \bar{x}_2)}{\sigma_{x_2}} + \frac{\sum_{j=1}^{n_2}(x_{2_j} - \bar{x}_1)}{\sigma_{x_1}}}{n_1 + n_2}\ ; \tag{9.4}$$

$$d_y = \frac{\frac{\sum_{i=1}^{n_1}(y_{1_i} - \bar{y}_2)}{\sigma_{y_2}} + \frac{\sum_{j=1}^{n_2}(y_{2_j} - \bar{y}_1)}{\sigma_{y_1}}}{n_1 + n_2}\ , \tag{9.5}$$

where s_1 has n_1 motif occurrences and s_2 has n_2 motif occurrences, (d_x, d_y) is the position of s_2 with respect to s_1, and ($\bar{x}$, $\bar{y}$) is the mean and (σ_x, σ_y) is the standard deviation along the x and y axes. Using this metric, we try to calculate the variation in motifs and their locations in the two sequences.

After computing (d_x, d_y) for all sequences in S with respect to the reference sequence (s_1), we plot them in the sequence space, as represented by the two plots in Fig. 9.2. The origin represents the reference sequence. It is important to note that the position of another sequence (calculated using (9.4) and (9.5)) with respect to the randomly selected reference sequence can be negative (in the x and/or y direction). In that case the sequence space will get extended to other quadrants as well, as in Fig. 9.2b.

9.3.2 Unsupervised Training with Local Outlier Factor (LOF)

Similar sequences are expected to cluster together in the sequence space. Malicious activity is known to produce irregular sequences of events. These anomalies would correspond to spurious points (global outliers) or local outliers in the scatter plot. In Fig. 9.2a, the point on the top-right corner of the plot is isolated from the rest of the points, making it anomalous. In this section we will concentrate on outlier detection, which has been a well-researched topic in databases and knowledge discovery [131, 132, 237, 238]. It is important to note that an outlier algorithm with our representation is inappropriate for online detection since it requires complete knowledge of all process sequences.

LOF [132] is a density-based outlier finding algorithm which defines a local neighborhood, using which a degree of "outlierness" is assigned to every object. The number of neighbors (*MinPts*) is an input parameter to the algorithm. A reachability density is calculated for every object which is the inverse of the average reachability distance of the object from its nearest neighbors. Finally, a local outlier factor (*LOF*) is associated with every object by comparing its reachability density with each of its neighbors. A local outlier is one whose neighbors have a high reachability density as compared to that object. For each point this algorithm gives a degree to which that point is an outlier as compared to its neighbors (anomaly score). Our system computes the anomaly scores for all the *SCS*s (represented as points in sequence space).

All the points for which the score is greater than a threshold are considered anomalous and removed.

We made some modifications to the original LOF algorithm to suit our needs. In the original paper [132], all the points are considered to be unique and there are no duplicates. In our case, there are many instances when the sequences are exactly the same (representative of identical application behavior). The corresponding points would thus have the same spatial coordinates within the sequence space. Density is the basis of our system, and hence we cannot ignore duplicates. Also, a human expert would be required to analyze the sequence space and suggest a reasonable value of *MinPts*. But the *LOF* values increase and decrease nonmonotonically [132], making the automated selection of *MinPts* highly desirable. We present some heuristics to automate the *LOF* and threshold parameters, making it a parameter-free technique.

Automating the Parameters

To select *MinPts*, we use clustering to identify the larger neighborhoods. Then, we scrutinize each cluster and approximate the number of neighbors in an average neighborhood. We use the L-Method [239] to predict the number of clusters in the representation. This is done by creating a *number of clusters vs. merge distance* graph obtained from merging one data point at a time in the sequence space. Starting with all N points in the sequence space, the 2 closest points are merged to form a cluster. At each step, a data point with minimum distance to another cluster or data point is merged. At the final step, all points are merged into the same cluster. The graph obtained has 3 distinct areas: a horizontal region (points/clusters close to each other merged), a vertical region (far away points/clusters merged), and a curved region in between. The number of clusters is represented by the knee of this curve, which is the intersection of a pair of lines fitted across the points in the graph that minimizes the root mean square error. Further details can be obtained from [239].

Assume k clusters are obtained in a given sequence space using L-Method (with each cluster containing at least 2 points). Let α_i be the actual number of points in cluster i, $1 \leq i \leq k$. Let ρ_i be the maximum pair-wise distance between any 2 points in cluster i; and τ_i is the average (pair-wise) distance between points in cluster i. Let β_i be the expected number of points in cluster i. Its value can be computed by dividing the area of the bounding box for the cluster with the average area occupied by the bounding box of any 2 points in the cluster (for simplicity we assume square-shaped clusters). Therefore, we get

$$\beta_i = \left(\frac{\rho_i}{\tau_i}\right)^2. \tag{9.6}$$

This gives us the expected number of points within the cluster. But the actual number of points is a_i. Thus, we equally distribute the excess points among

all the points constituting the cluster. This gives us an approximate value for *MinPts* (the number of *close* neighbors) of the cluster i:

$$\gamma_i = \left\lceil \frac{\alpha_i - \beta_i}{\beta_i} \right\rceil . \tag{9.7}$$

After obtaining *MinPts* for all k clusters, we compute a weighted mean over all clusters to obtain the average number of *MinPts* for the entire sequence space:

$$MinPts = \left\lceil \frac{\sum_{i=1}^{k} \gamma_i \alpha_i}{\sum_{i=1}^{k} \alpha_i} \right\rceil . \tag{9.8}$$

Only clusters with at least 2 points are used in this computation. But this approach gives a reasonable value for the average number of *MinPts* in a sequence space if all the points are unique. In case of duplicates, (9.6) is affected since the maximum distance still remains the same whereas the average value is suppressed due to the presence of points with same spatial coordinates. If there are q points corresponding to a coordinate (x, y), then each of the q points is bound to have at least $(q - 1)$ *MinPts*.

Let p be the number of frequent data points (i.e., *frequency* ≥ 2) in cluster i. Let ψ_j be the frequency of a data point j in cluster i. In other words, it is the number of times that the same instance occurs in the data. We compute γ' the same way as (9.7), where γ' is the *MinPts* value for cluster i assuming unique points (no multiple instance of the same data point) in the sequence space:

$$\gamma'_i = \left\lceil \frac{\alpha_i - \beta_i}{\beta_i} \right\rceil . \tag{9.9}$$

This value is then modified to accommodate the frequently occurring points (corresponding to sequences sharing the same spatial positions in the sequence space). We compute a weighted mean to obtain an appropriate value of *MinPts* in cluster i as follows:

$$\gamma_i = \left\lceil \frac{\gamma'_i \alpha_i + \sum_{j=1}^{p} \psi_j (\psi_j - 1)}{\alpha_i + \sum_{j=1}^{p} \psi_j} \right\rceil . \tag{9.10}$$

Average *MinPts* for the entire plot can then be computed using (9.8).

LOF only assigns a local outlier factor for a point in the sequence space which corresponds to its anomaly score. If the score is above a user-specified threshold, then it is considered as anomalous and hence filtered from the data set. If the threshold is too low, there is a risk of filtering a lot of points, many of which may depict normal application behavior. On the contrary, if the threshold is too high, some of the data points corresponding to actual intrusions (but close to many other data points on the sequence space) may not get filtered. We present a heuristic to compute the threshold automatically without the need of any human expert. One point to note here is that the effect

of false alarms for data purification is not as adverse as that of false-alarm generation during online detection, if still within reasonable limits which are defined by the user. We compute the threshold automatically by ordering the LOF scores and plotting them in increasing order (with each data point along the x-axis and the anomaly/LOF score along the y-axis). Since the normal points are assumed in abundance, their LOF scores are ideally 1. We are interested in the scores after the first steep rise of this plot, since these correspond to outliers. Ignoring all the scores below the first steep rise (corresponding to normal sequences), the cutoff value can be computed as the median of all the scores thereafter. This heuristic gives a reasonable threshold value for the various applications in our data sets.

9.4 Anomaly Detection

The filtered data set obtained above can provide attack-free training input to any supervised learning algorithm that performs anomaly detection.

9.4.1 Representation with Arguments

System call sequences have been effectively used in host-based systems where a sliding window of fixed length is used. We introduce the term *tuple* to represent an instance of the sliding window. The simplest representation (denoted by *S-rep*) would therefore consist of 6 contiguous system call tokens (since length 6 is claimed to give best results in stide and t-stide [218]):

$$s_0\ s_1\ s_2\ s_3\ s_4\ s_5\ ,$$

where s_i is the system call at a distance i from the current system call within the 6-gram.

Consider the following sequence of system calls: *open, read, write, ..., close*. This would seem like a perfectly normal sequence of events corresponding to a typical file access. But what happens if the file being accessed is *passwd*? Only the super-user should have the rights to make any modifications to this file. Malicious intent involving this file would not be captured if only the system call sequences are monitored. This lays stress on the enhancement of features to enrich the representation of application behavior at the operating system level. The enhancement for host-based systems, as is obvious from the example above, is to scrutinize the system call arguments as well.

The argument-based representation, denoted by *A-rep*, takes into consideration the various attributes like return value, error status, besides other arguments pertaining to the current system call. Let the maximum number of attributes for any system call be η. The representation would now consist of tuples of the form:

$$s_0\ a_1\ a_2\ a_3 \cdots a_\eta\ ,$$

where s_0 is the current system call and a_i is its ith argument. It is important to note that the order of the attributes within the tuple is system call dependent. Also, by including all possible attributes associated with the system call, we can maximize the amount of information that can be extracted from the audit logs. We fix the total number of arguments in the tuple to the maximum number of attributes for any system call. If any system call does not have a particular attribute, it is replaced by a NULL value.

S-rep models system call sequences, whereas A-rep adds argument information, so merging the two representations is an obvious choice. The merged representation, called *M-rep*, comprises tuples containing all the system calls within the 6-gram (S-rep) along with the attributes for the current system call (A-rep). The tuple is thus represented as

$$s_0\ a_1\ a_2\ a_3 \cdots a_\eta\ s_1\ s_2\ s_3\ s_4\ s_5\ .$$

A further modification to the feature space includes all the system calls in the fixed sliding window and the η attributes for all the system calls within that window. This enhanced representation is denoted as *M*-rep*.

9.4.2 Supervised Training with LERAD

The efficacy of host-based anomaly detection systems might be enhanced by enforcing constraints over the system calls and their arguments. Due to the enormous size and varied nature of the applications, manual constraint formulation is a tedious task. Moreover, due to the nature of the attributes it might not be feasible for even a group of human experts to develop a complete set of constraints in a short span of time. The work-around to this problem is to use a machine learning approach which can automatically learn the important associations between the various attributes without the intervention of a human expert. An important aspect of rule learning is the simplicity and comprehensibility of the rules. The solution can be formulated as a 5-tuple $(A, \Phi, I, \Re, \varsigma)$, where A is the set of N attributes, Φ is the set of all possible values for the attributes in A, I is the set of input tuples which is a subset of the N-ary Cartesian product over A, $\Re$ is the rule set, and ς is the maximum number of conditions in a rule.

LERAD is an efficient conditional rule learning algorithm. A LERAD rule is of the form

$$(\alpha_i = \phi_p) \wedge (\alpha_j = \phi_q) \wedge \cdots \wedge \varsigma terms \Rightarrow \alpha_k \in \{\phi_a, \phi_b, \dots\}\ ,$$

where α_i, α_j, α_k are the attributes, and ϕ_p, ϕ_q, ϕ_a, and ϕ_b are the values for the corresponding attributes. Algorithms for finding association rules (such as Apriori [58]) generate a large number of rules. But a large rule set would incur large overhead during the detection phase and may not be appropriate to attain our objective. We would like to have a *minimal* set of rules describing

the normal training data. LERAD forms a small set of rules. It is briefly described here; more details can be obtained from [240].

For each rule in $\Re$, LERAD associates a probability p of observing a value not in the consequent:

$$p = \frac{r}{n} , \tag{9.11}$$

where r is the cardinality of the set in the consequent and n is the number of tuples that satisfy the rule during training. This probability estimation of novel (zero frequency) events is due to Witten and Bell [241]. Since p estimates the probability of a novel event, the larger p is, the less anomalous a novel event is. During the detection phase, tuples that match the antecedent but not the consequent of a rule are considered anomalous, and an anomaly score is associated with every rule violated. When a novel event is observed, the degree of anomaly (anomaly score) is estimated by:

$$AnomalyScore = \frac{1}{p} = \frac{n}{r} . \tag{9.12}$$

A nonstationary model is assumed for LERAD since novel events are "bursty" in conjunction with attacks. A factor t is introduced, which is the time interval since the last novel (anomalous) event. When a novel event occurred recently (i.e., small value for t), a novel event is more likely to occur at the present moment. Hence the anomaly should be low. This factor is therefore multiplied by the anomaly score, modifying it to t/p. Since a record can deviate from the consequent of more than one rule, the total anomaly score of a record is aggregated over all the rules violated by the tuple to combine the effect from violation of multiple rules:

$$Total\ Anomaly\ Score = \sum_i \left(\frac{t_i}{p_i} \right) = \sum_i \left(\frac{tn_i}{r_i} \right) , \tag{9.13}$$

where i is the index of a rule which the tuple has violated. The anomaly score is aggregated over all the rules. The more the violations, the more critical the anomaly is, and the higher the anomaly score should be. An alarm is raised if the total anomaly score is above a threshold.

We used the various feature representations discussed in Sect. 9.4.1 to build models per application using LERAD. We modified the rule generation procedure enforcing a stricter rule set. All the rules were forced to have system call in the antecedent since it is the key attribute in a host-based system. The only exception we made was the generation of rules with no antecedent.

Sequence of System Calls: S-LERAD

Before using argument information, it was important to know whether LERAD would be able to capture the correlations among system calls in a sequence. So we used the S-rep to learn rules of the form:

$$(s_0 = close) \wedge (s_1 = mmap) \wedge (s_5 = open) \Rightarrow s_2 \in \{munmap\}$$
$$(1/p = n/r = 455/1)$$

This rule is analogous to encountering *close* as the current system call (represented as s_0), followed by *mmap* and *munmap*, and *open* as the sixth system call (s_5) in a window of size 6 sliding across the audit trail. Each rule is associated with an n/r value. The number 455 in the numerator refers to the number of training instances that comply with the rule (n in (9.12)). The number 1 in the denominator implies that there exists just one distinct value of the consequent (*munmap* in this case) when all the conditions in the premise hold true (r in (9.12)).

Argument-Based Model: A-LERAD

We propose that argument and other key attribute information is integral to modeling a good host-based anomaly detection system. We used A-rep to generate rules. A sample rule is

$$(s_0 = munmap) \Rightarrow a_1 \in \{0x134, 0x102, 0x211, 0x124\}$$
$$(1/p = n/r = 500/4)$$

In the above rule, 500/4 refers to the n/r value (9.12) for the rule, that is, the number of training instances complying with the rule (500 in this case) divided by the cardinality of the set of allowed values in the consequent. The rule in the above example is complied by 500 tuples and there are 4 distinct values for the first argument when the system call is *munmap*.

Merging System Call Sequence and Argument Information of the Current System Call: M-LERAD

A merged model (M-rep) is produced by concatenating S-rep and A-rep. Each tuple now consists of the system call, various attributes for the current system call, and the previous five system calls. The n/r values obtained from all the rules violated are aggregated into the anomaly score, which is then used to generate an alarm based upon the threshold. An example of an M-LERAD rule is

$$(s_0 = munmap) \wedge (s_5 = close) \wedge (a_3 = 0) \Rightarrow s_2 \in \{munmap\}$$
$$(1/p = n/r = 107/1)$$

Merging System Call Sequence and Argument Information for All System Calls in the Sequence: M*-LERAD

All the proposed variants, namely, S-LERAD, A-LERAD, and M-LERAD, consider a sequence of 6 system calls and/or take into the arguments for the current system call. We propose another variant called multiple argument

LERAD (M*-LERAD). In addition to using the system call sequence and the arguments for the current system call, the tuples now also comprise the arguments for the other system calls within the fixed length sequence of size 6 (M*-rep).

Can a rule learning algorithm efficiently generalize over the various features extracted? Does extracting more features, namely, arguments along with system call sequences, result in better application modeling? More specifically, does the violation of rule(s) relate to an attack? And would this result in an increase in attack detections while lowering the false-alarm rate? These are some of the questions we seek answers for.

9.5 Experimental Evaluations

We evaluated our techniques on applications obtained from three different data sets:

1. The DARPA intrusion detection evaluation data set was developed at the MIT Lincoln Labs [88]. Various intrusion detection systems were evaluated using a test bed involving machines comprising Linux, SunOS, Sun Solaris, and Windows NT systems. We used the BSM audit log for the Solaris host;
2. The University of New Mexico (UNM) data set has been used in [215, 218, 224]; and
3. FIT-UTK data set consists of Excel macro executions [228, 242].

Table 9.1. Effect of LOF *MinPts* values

Application	Total Attacks	Number of attacks detected (with false-alarm count) for different values of *MinPts* (% of total population)				
		5%	10%	15%	20%	Automated
eject	2	1 (1)	2 (1)	2 (0)	2 (0)	2 (0)
fdformat	3	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)
ftpd	6	0 (6)	0 (11)	6 (6)	6 (1)	0 (11)
ps	4	0 (6)	4 (1)	4 (1)	4 (2)	4 (49)
lpr	1	0 (123)	1 (193)	1 (198)	1 (157)	1 (97)
login	1	0 (1)	0 (2)	1 (2)	1 (2)	1 (2)
excel	2	2 (0)	0 (3)	0 (0)	0 (0)	2 (0)
Total	19	6 (137)	10 (211)	17 207	17 (162)	13 (159)

9.5.1 Data Cleaning

Evaluation Procedures and Criteria

From the DARPA/LL data set, we used data for ftpd, ps, eject, and fdformat applications, chosen due to their varied sizes. We also expected to find a good mix of benign and malicious behavior in these applications which would help us to evaluate the effectiveness of our models. We used BSM data logged for weeks 3 (attack-free), 4 and 5 (with attacks and their time stamps). Two applications (lpr and login) from the UNM data set and Excel logs from the FIT-UTK data set were used. LOF was used to detect outliers in the sequence space for all the applications. The number of true positives was noted for different *MinPts* values 5%, 10%, 15%, 20% of the entire population, as well as the value obtained using our heuristic.

Results and Analysis

For various values of the *MinPts* parameter to LOF, our technique was successfully able to detect all the 19 attacks in the data set, as depicted in Table 9.1. But no single value of *MinPts* was ideal to detect all the attacks. The two parameter values 15% and 20% seem to have the maximum number of detections (17 each). The only attacks missed were the ones in the Excel application where a reasonable value of *MinPts* is best suggested as 5%. Our methodology for automated *MinPts* calculation was successful in computing the correct number for the parameter, and hence successfully detected the attack sequence as outlier (for which the 15% and 20% values failed). The automated *LOF* parameter detected all the attacks except the ones in the ftpd application. The inability of LOF to detect the anomalies in this representation is attributed to the fact that all the points in the cluster correspond to attacks. The automated technique successfully detected all other attacks, suggesting that the *MinPts* values computed using our heuristic are generally reasonable.

The number of false alarms generated is very high for the lpr application, which constitutes over $3,700$ sequences and approximately 3.1 million system calls. The data was collected over 77 different hosts and represents high variance in application behavior. Though we were able to capture the lpr attack invoked by the lprcp attack script, we also detected other behavioral anomalies which do not correspond to attacks. We reiterate that our goal is to retain generic application behavior and shun anomalies. Peculiar (but normal) sequences would also be deemed anomalous since they are not representative of the general way in which the application functions.

The reason why our representation plots attacks as outliers is as follows. An attack modifies the course of events, resulting in (1) either the absence of a motif, or (2) altered spatial positions of motifs within the sequence due to repetition of a motif, or (3) the presence of an entirely new motif. All these

instances affect the spatial relationships among the different motifs within the sequence. Ultimately, this affects the distance of the malicious sequence with respect to the reference sequence, resulting in an outlier being plotted on the sequence space. It is this drift within the sequence space that the outlier detection algorithm is able to capture as an anomaly.

9.5.2 Anomaly Detection with Arguments

Evaluation Procedures and Criteria

For the DARPA/LL data set, we used data for ftpd, telnetd, sendmail, tcsh, login, ps, eject, fdformat, sh, quota, and ufsdump applications, chosen due to their varied sizes – ranging from very large (over 1 million audit events) to very small ($\sim 1,500$ system calls). We used week 3 data for training and weeks 4 and 5 for testing. We also used data for three applications (lpr, login, and ps) from the UNM data set as well as logs from the FIT-UTK Excel macro data set. The input tuples for S-LERAD, A-LERAD, M-LERAD, and M*-LERAD were as discussed in Sect. 9.4.2. For tide, stide, and t-stide, we used a window size of 6. For all the techniques, alarms were merged in decreasing order of the anomaly scores and then evaluated for the number of true detections at varied false-alarm rates.

Table 9.2. Comparison of sequence- and argument-based representations

Technique	Number of attacks detected at different false-alarm rates ($\times 10^{-3}$% false alarms)				
	0.0	0.25	0.5	1.0	2.5
tide	5	6	6	8	9
stide	5	6	9	10	12
t-stide	5	6	9	10	13
S-LERAD	3	8	10	14	15
A-LERAD	3	10	13	17	19
M-LERAD	3	8	14	16	19
M*-LERAD	1	2	4	11	18

Results and Analysis

When only sequence-based techniques are compared, both S-LERAD and t-stide were able to detect all the attacks in UNM and FIT-UTK data sets. However, t-stide generated more false alarms for the ps and Excel applications (58 and 92, respectively), as compared to 2 and 0 false alarms in the case of S-LERAD. For the DARPA/LL data set, tide, stide, and t-stide detected the

most attacks at zero false alarms, but are outperformed by S-LERAD as the threshold is relaxed (Table 9.2). It can also be observed from the table that A-LERAD fared better than S-LERAD and the other sequence-based techniques, suggesting that argument information is more useful than sequence information. A-LERAD performance is similar to that of M-LERAD, implying that the sequence information is redundant; it does not add substantial information to what is already gathered from arguments. M*-LERAD performed the worst among all the techniques at false-alarm rate lower than $0.5 \times 10^{-3}\%$. The reason for such a performance is that M*-LERAD generated alarms for both sequence- and argument-based anomalies. An anomalous argument in one system call raised an alarm in six different tuples, leading to a higher false-alarm rate. As the alarm threshold was relaxed, the detection rate improved. At the rate of $2.5 \times 10^{-3}\%$ false alarms, S-LERAD detected 15 attacks as compared to 19 detections by A-LERAD and M-LERAD, whereas M*-LERAD detected 18 attacks correctly.

The better performance of LERAD variants can be attributed to its anomaly scoring function. It associates a probabilistic score with every rule. Instead of a binary (present/absent) value (as in the case of stide and t-stide), this probability value is used to compute the degree of anomalousness. It also incorporates a parameter for the time elapsed since a novel value was seen for an attribute. The advantage is twofold: (1) it assists in detecting long-term anomalies and (2) it suppresses the generation of multiple alarms for novel attribute values in a sudden burst of data. An interesting observation is that the sequence-based techniques generally detected the U2R attacks whereas the R2L and DoS attacks were better detected by the argument-based techniques.

Our argument-based techniques detected different types of anomalies. In most of the cases, the anomalies did not represent the true nature of the attack. Some attacks were detected by subsequent anomalous user behavior; few others were detected by learning only a portion of the attack, while others were detected by capturing intruder errors. Although these anomalies led to the detection of some attacks, some of them were also responsible for raising false alarms, a problem inherent to all anomaly detection systems.

Table 9.3. Effects of filtering the training data on the performance in test phase

Application	Number of attacks detected (false alarms generated) – stide		Number of attacks detected (false alarms generated) – LERAD	
	Without filtering	With filtering	Without filtering	With filtering
ftpd	0(0)	3(0)	0(0)	3(0)
eject	1(0)	1(0)	1(0)	1(0)
fdformat	2(0)	2(0)	1(0)	1(0)
ps	0(0)	1(0)	0(1)	1(1)

9.5.3 Anomaly Detection with Cleaned Data vs. Raw Data

Evaluation Procedures and Criteria

Only the MIT Lincoln Labs data set was used for this set of experiments since it contained sufficient attacks and argument information to be used in both *adulterated* training and test data sets. We combined the *clean* week 3 data with the *mixed* week 4 data of the MIT Lincoln Labs data set to obtain an unlabeled data set. We used this to train stide and LERAD. We then tested on week 5 data (containing attacks with known time stamps). Subsequently, we filtered out the outliers detected from the combined data set. The refined data set was then used to train stide and LERAD. Week 5 data was used for testing purposes. The input to stide and LERAD was discussed in Sect. 9.5.2. In all cases, alarms are accumulated for the applications and then evaluated for the number of true detections and false positives. The results are displayed in Table 9.3.

Results and Analysis

For the four applications in the data set, stide was able to detect 3 attacks without filtering the training data compared to 7 attacks after refining the training set. For LERAD, there were 2 detections before filtering versus 6 true positives after purging the anomalies in training data. Thus, in both cases, there was an improvement in the performance of the system in terms of the number of attack detections. The better performance is attributed to the fact that some of the attacks in the adulterated training and testing data sets were similar in character. With the adulterated training data, the attacks were assumed normal resulting in the creation of an improper model. Hence both the systems missed the attacks during the test phase. But our filtering procedure was able to remove the anomalies from the training data to create a correct model of normal application behavior. This led to the detection of the attacks during testing.

It can also be noted from the table that no false alarms were generated in any experiment with stide. For LERAD, there was only one false alarm for the ps application (with and without filtering). Our results indicate that the filtering mechanism was effective in purging out anomalies in training data, resulting in the detection of more attacks without increasing the number of false alarms in the test phase.

9.6 Concluding Remarks

In this chapter, we present two enriched representations: (1) motifs and their locations are used to represent sequences in a single sequence space, this being an off-line procedure, and (2) system call arguments modeled for online

anomaly detection. We also presented two different aspects of machine learning: unsupervised learning and supervised learning for anomaly detection. Our techniques cater to the various issues and can be integrated to form a complete host-based system.

Most of the traditional host-based IDSs require a *clean* training data set which is difficult to obtain. We present a motif-based representation for system call sequences (*SCS*s) based upon their spatial positions within the sequence. Our system also creates a single representation for all *SCS*s called a sequence space – using a distance metric between the motif-based representations. A local outlier algorithm is used to purge this data of all attacks and other anomalies. We demonstrate empirically that this technique can be effectively used to create a *clean* training data set. This data can then be used by any supervised anomaly detection system to learn and create models of normal application behavior. Results from our experiments with two online detection systems (stide and LERAD) indicate a drastic improvement in the number of attacks detected (without increasing the number of false alarms) during test phase when our technique is used for filtering the training set.

Merging argument and sequence information creates a richer model for anomaly detection. This relates to the issue of feature extraction and utilization for better behavioral modeling in a host-based system. We portrayed the efficacy of incorporating system call argument information and used a rule learning algorithm to model a host-based anomaly detection system. Based upon experiments on well-known data sets, we claim that our argument-based model, A-LERAD, detected more attacks than all the sequence-based techniques. Our sequence-based variant (S-LERAD) was also able to generalize better than the prevalent sequence-based techniques, which rely on pure memorization.

The data cleaning procedure can be integrated with a hybrid of signature- and anomaly-based systems for better accuracy and the ability to detect novel attacks. Our system can also be used for user profiling and detecting masquerades. In terms of efficiency, the only bottleneck in our system is the motif extraction phase where cross-match is performed pair-wise. Speed-up is possible by using other techniques, such as suffix trees [243–246]. We are also working on refining the motif relationships in the motif-based representation. Our argument- and sequence-based representations assume fixed-size tuples. A possible extension is a variable-length attribute window for more accurate modeling. Also, more sophisticated features can be devised from the argument information.

10

A Decision-Theoretic, Semi-Supervised Model for Intrusion Detection

Terran Lane

Summary. In this chapter, we develop a model of intrusion detection (IDS) based on semi-supervised learning. This model attempts to fuse misuse detection with anomaly detection and to exploit strengths of both. In the process of developing this model, we examine different cost functions for the IDS domain and identify two key assumptions that are often implicitly employed in the IDS literature. We demonstrate that relaxing these assumptions requires a decision-theoretic control maker based on the partially observable Markov decision process (POMDP) framework. This insight opens up a novel space of IDS models and allows precise quantification of the computational expense of optimal decision-making for specific IDS variants (e.g., additional data sources) and cost functions. While decision-making for many POMDPs is formally intractable, recognizing the equivalence of the IDS problem to solution of a POMDP makes available the wide variety of exact and approximate learning techniques developed for POMDPs. We demonstrate the performance of the simplest of these models (for which optimal decision-making is tractable) on a previously studied user-level IDS problem, showing that, at the lower limit, our semi-supervised learning model is equivalent to a pure anomaly detection system, but that our model is also capable of exploiting increasing degrees of intermittently labeled data. When such intermittently labeled data *is* available, our system performs strongly compared to a number of current, pure anomaly detection systems.

10.1 Introduction

The research presented in this chapter is an attempt to achieve three desiderata for adaptive intrusion detection systems (IDS):

1. An adaptive IDS should be able to exploit known attacks, known normal data, and the copious amounts of unlabeled data that are typically available.
2. An adaptive IDS should be able to take advantage of labeled data whenever it becomes available, not just during a restricted "training phase".
3. An IDS should be able to implement a number of different cost functions, according to site-specific security policies.

The first desideratum arises from examining the commonly cited strengths of misuse and anomaly detection: misuse detection systems can take advantage of historical attacks, while anomaly detection systems, by neglecting this source of data, are vulnerable to even well-known attacks. On the anomaly detection side, however, is the argument that pure anomaly detection systems are not biased for or against any particular attack and so are less likely to be fooled by minor variants on old attacks or wildly novel attacks. Furthermore, anomaly detection systems do not rely on labeled training data, which is often quite difficult to come by. We argue that this distinction is a false dichotomy – it is vital to exploit knowledge of current attacks, but it is equally vital to not be susceptible to radically new attacks. We believe that the usual formulation of misuse and anomaly detection systems represent only the endpoints on a spectrum between fully supervised (misuse) and fully unsupervised (anomaly) learning. We demonstrate a system that lies in the continuum between the two by modeling both the state of the system and of the attacker. This system begins with a default, prior model of the attacker that is equivalent to the null hypothesis often employed in anomaly detection. As labeled attack data is added to the system, this model is updated, pushing the system closer to misuse detection.

The second desideratum comes from a general dissatisfaction with the traditional train/test framework that is employed for many (though certainly not all) misuse/anomaly detection systems. Typically, the model is developed from hand-labeled data during the training phase, then is fixed and employed as a detector on arbitrary amounts of unlabeled data in the testing phase. In field conditions, however, things are rarely so nicely partitioned. Some labeled data is likely to be available for initializing the model, and the vast majority of data thereafter is unlabeled, but labeled data does occasionally become available later. For example:

- Short periods of normal data can be examined by hand and certified as clean.
- New users or services are put online or the network reconfigured. At this point, some quantity of data can be hand-labeled as normal to update the model to the new circumstances.
- Vulnerability notifications, often with example exploits or reverse-engineered code, are regularly posted through security agencies.
- Attacks that are initially missed by the IDS but are discovered post factum (e.g., through consequences of the attack, such as lost data) can be hand-labeled by a security officer from recorded audit data.

Thus, we argue that a more flexible approach to adaptive intrusion detection is to treat it as a *partially labeled* data problem. That is, rather than dividing time into training and testing phases, the learning system should treat both uniformly, classifying any unlabeled data while learning from any available labeled data, whenever it occurs. This allows a system to take maximum advantage of labeled data. When only a single block of labeled data is available,

the system's behavior should reduce to that of a standard, train/test anomaly or misuse detection system. We give a semi-supervised training algorithm for our IDS that allows it to flexibly handle either supervised or unsupervised data, as available. When only unlabeled data is available, the system both labels it and uses it to refine the existing model, while when labeled data becomes available the learner incorporates it into the model.

The final desideratum is a result of observations made in our previous work [247, 248] as well as the work of Fawcett and Provost [249] and a number of others. While much IDS research has employed false-positive/false-alarm rates as success criteria, these measures are too coarse for many practical applications. For example, the elapsed time between the onset of an attack and its detection is a critical measure of the damage that an attacker could cause. Also, multiple alarms for a single attack may provide only marginal additional utility beyond the initial alarm (indeed, they may contribute *negative* utility by annoying the security officer). We wish to allow the IDS to optimize a general cost function that can be chosen to reflect the site security policy. In developing our IDS model, we examine common cost functions for the IDS domain and make explicit two widely employed cost assumptions. We demonstrate that these assumptions lead to a formulation of the intrusion detection task as a belief state (probability distribution) monitoring problem.

We demonstrate that many desirable cost functions for the IDS problem require relaxing these assumptions, which, in turn, yields a formulation of the detection problem as a stochastic control problem with hidden state – that is, as learning and planning in a partially observable Markov decision process (POMDP). This is an unfortunate consequence because, unlike the monitoring problem, optimal planning and learning in general POMDPs is known to be intractable. It is, however, important to realize this because it clarifies the computational complexity of various cost functions and allows us a rigorous way to trade off complexity against descriptiveness. To our knowledge, this is the first result on the formal complexity of decision-making in the intrusion detection task. We also spend some discussion on possible directions for addressing the intractability of these models. In particular, reducing the IDS problem to learning and planning in POMDPs opens it up to a wide variety of approximate, yet effective, methods that have been developed in recent years.

Finally, returning to the simplest formulation of our semi-supervised approach, we demonstrate empirically that the belief-state-monitoring approach performs well on a command-line level IDS task. Specifically, when run in a semi-supervised mode (mixed anomaly and misuse detection, with sporadically introduced newly labeled data), it outperforms a number of previously proposed pure anomaly detection techniques for that domain, while it performs comparably to them when run in a pure anomaly detection mode.

10.2 Related Work

Work in adaptive or statistical intrusion detection in the security community is rich and widespread, dating back to at least 1980 with Anderson's initial proposal for such systems [198], followed by Denning's model of statistical intrusion detection which proposed n-gram and Markov chain data models [199], and Lunt et al.'s work on expert system [250, 251] and statistical expert system IDSs [252]. There are a number of surveys on work through the mid-1990s in this area, including a good one by Mukherjee et al. [253].

More recently, the intrusion detection question has entered the statistics and machine learning communities and researchers have proposed techniques based on Markov chains [254], hidden Markov models [248], association rules [255, 256], and a number of others. Beginning with Forrest et al. [257, 258], a number of researchers have developed IDSs based on models of biological immune systems [259, 260]. These "artificial immune systems" employ, effectively, instance- or rule-based learners in either anomaly or misuse detection frameworks.

One difficulty in comparing these proposals is that they are developed for very different data sets – different methods have been applied to user command-line data, system call data, and network packet data. There are few uniform data sets that are widely used for measuring and comparing IDS performance. The significant exceptions are the DARPA/Lincoln Labs network attack detection testbed data set [261], and a multi-method study of user-level anomaly detection by Schonlau et al. [262]. We base our experimental work on the Schonlau et al. data and compare our results with the six anomaly detection methods that they examine. They tested the relative performances of a hypothesis test between a first-order Markov chain and a Dirichlet mixture of multinomials model of command generation [263]; a Markov chain whose statistics are exponentially discounted over time in an attempt to track concept drift [254]; a model of the frequency distribution of rare and unique commands [264]; an approximate order-10 Markov chain model [265]; a technique that measures the compressibility of the data with respect to a model of "normal" (i.e., attack-free) data; and an instance-based method [266]. The general result is that all methods suffered from high false-alarm rates (none were able to achieve the 1% rate that they were targeting) and that none uniformly dominated the others, with the exception of the uniqueness method which was uniformly the poorest.

10.3 A New Model of Intrusion Detection

In this section, we develop a new class of intrusion detection methods. We begin with a generative model that supports the semi-supervised learning component of our models, followed by a description of inference and learning

algorithms for this model. We then examine action (i.e., alarm) selection, including the influence of costs, and explicitly identify two commonly employed assumptions about action selection. The experimental work we present in Sect. 10.4 is based on semi-supervised learning under these assumptions, but first we demonstrate that relaxing these assumptions yields a class of IDSs based on the POMDP framework.

10.3.1 Generative Data Model

We will model both valid user and intruder (attacker) as unobservable, homogeneous Markov random variables denoted $U_t \in \{u^1 \cdots u^m\}$ and $A_t \in \{a^1 \cdots a^n\}$, respectively. This represents a distinct break from previous anomaly detection models which model only the valid user. The observable state of the computer system will be represented with the vector variable $\mathbf{X}$, taking on values defined by the domains of the measurable components of the system (e.g., network latency, memory footprint, GUI event). At time t, the measured state of the system, $\mathbf{X}_t$, is generated either by the valid user or the intruder, governed by an *intermittently observed* decision variable $D_t \in \{\texttt{USER}, \texttt{ATTACKER}\}$. The system is experiencing an intrusion just when the observed data is generated by the intruder, i.e., when $D_t = \texttt{ATTACKER}$. Initially, we model the D_t variables as a series of independent, binomially distributed random variables. This is a bit of a strong assumption, as it neglects the possibility that attacks occur in runs or that the attacker has a preferred temporal distribution,[1] but it is the simplest assumption about the mixing variable and can be straightforwardly relaxed. We discuss extensions to this model in Sect. 10.5.

By treating D_t as an intermittent variable, rather than purely observed (as in misuse detection) or purely hidden (as in anomaly detection), we can train this model on *partially labeled* data. As discussed in Sect. 10.1, partially labeled data is a plausible scenario for intrusion detection.

The generative model is displayed as a graphical model in Fig. 10.1. This model is a temporal or dynamic Bayes network, or DBN [267–269] – a graphical representation of the Markov independence properties of a statistical distribution over time-series data.[2] Each node in this network represents one of the random variables of the system, while arcs represent statistical dependence. The model is quantified by conditional probability distributions (CPDs; not pictured here) that specify the probability distribution of a node, conditioned on the values of its parent nodes. Specifically, this model requires the CPDs $\Pr[A_{t+1}|A_t, \theta_A]$, $\Pr[U_{t+1}|U_t, \theta_U]$, $\Pr[D_t|\theta_D]$, and $\Pr[\mathbf{X_t}|U_t, A_t, D_t, \theta_X]$, parameterized by θ_A, θ_U, θ_D, and θ_X, respectively. To

[1] E.g., injecting attack elements periodically to avoid easily discerned clusters of attack commands.

[2] The model of Fig. 10.1 can also be viewed as a mixture of hidden Markov models or as a variant of a factorial HMM.

handle the first time step, $t = 0$, we also require $\Pr[U_0|\theta_{U0}]$ and $\Pr[A_0|\theta_{A0}]$. In the case of discrete variables (as we assume in this chapter), the various distributions are conditionally multinomial, the CPDs $\Pr[A_{t+1}|A_t, \theta_A]$ and $\Pr[U_{t+1}|U_t, \theta_U]$ are single-step Markov transition matrices, and there are a total of $|A| + |U| + |A|^2 + |U|^2 + |D| + |A||U||D||\mathbf{X}|$ parameters to estimate. If a fully Bayesian model is desired, we could also specify parameter priors, but we will assume uniform priors in this chapter. We will discuss learning these parameters in the next section.

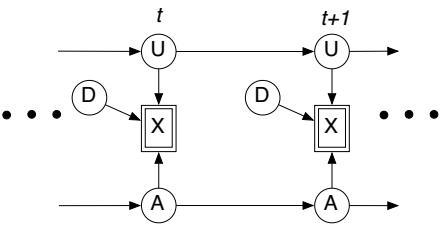


Fig. 10.1. Graphical model structure for intrusion detection inference and decision-making. Circular nodes are hidden or intermittently observed variables; the rectangular node **X** is a special node for the vector valued observable, potentially containing additional, currently unspecified rich Bayesian network structure

10.3.2 Inference and Learning

There are two important operations on the model of Fig. 10.1: estimating the probability of hidden variables, such as D_t, given observed data (inference) and estimating the values of the model parameters such as θ_U, etc. (learning). Both algorithms can be run in batch mode, in which all time points are examined at every step, or in online mode, in which the model is updated dynamically at every time point without reexamining all the previous time points. We cover only the batch-mode cases here; the online versions can be derived from the algorithms we present by rewriting them in a recursive fashion. The learning algorithm is the expectation-maximization (EM) algorithm [25, 270, 271], which uses inference as a subroutine, so we will begin with inference. Exact inference algorithms for (dynamic) Bayesian networks are well understood, so we do not present the derivation of the following algorithm here. For details on the derivation, we refer the reader to a standard text on Bayesian networks, such as Jensen's [268].

The inference algorithm for the IDS probability model (Fig. 10.2) employs the notion of *potentials*. Essentially, potentials are just denormalized probability functions – nonnegative functions with finite (but not necessarily unity) integrals. The model of Fig. 10.1 is compiled to a series of "clique potentials",

$\phi(U_t, A_t, D_t, \mathbf{X}_t)$, along with a pair of intermediate potentials, $\phi(U_t, A_t, U_{t+1})$, and $\phi(A_t, U_{t+1}, A_{t+1})$, that allow us to propagate information backward and forward between $\phi(U_t, A_t, D_t, \mathbf{X}_t)$ and $\phi(U_{t+1}, A_{t+1}, D_{t+1}, \mathbf{X}_{t+1})$. Intuitively, in the inference algorithm, potential functions act as "intermediate" or "scratch" variables, to track the estimates of probability distributions as we propagate information about observations and labels across time. In practice, the potentials will be implemented as tables of probability values, with one cell for each configuration of the argument variables of that potential. During the execution of `inference()`, the potentials will store intermediate results; when execution completes, the potentials will exactly represent denormalized versions of the corresponding joint probability distributions. The desired PDFs can be recovered through a normalization step, as in the `learnEM()` algorithm of Fig. 10.3. Thus, after the execution of the inference algorithm,

$$\Pr[U_t, A_t, D_t, X_t | X_{1:T}, D_{1:T}^{\text{observed}}, \Theta] = \frac{\phi(U_t, A_t, D_t, \mathbf{X}_t)}{\sum_{U,A,D,X} \phi(U_t, A_t, D_t, \mathbf{X}_t)} \ .$$

In addition to the potential functions, we will also employ *separator* functions, $s(U_t, A_t)$, $s(A_t, U_{t+1})$, and $s(U_{t+1}, A_{t+1})$. Separators[3] are just another set of tables for storing intermediate results. Roughly, they correspond to marginalizations of potentials in order to remove particular variables but, unlike potentials, they do not have a purpose beyond the scope of the inference algorithm.

The batch-mode version of the inference algorithm for the IDS model with discrete variables is given in Fig. 10.2. In this pseudo-code, the function $\delta()$ represents the unit potential – 1 at the value of its argument and 0 elsewhere. The multiplication/division operation is point-by-point multiplication/division across the range of two potentials, and the "$*=$" operation is a C-like combined multiplication and assignment operation. All summations are taken to be over the entire range of hidden variables (e.g., when D_t is unobserved) and over only the single observed value when a variable is observed.

The inference algorithm proceeds in three stages: initialization, the forward (or evidence collection) pass, and the backward (or evidence distribution) pass. After these three stages, the potentials are guaranteed to accurately reflect all available observable evidence [268]. Actual probabilities for the variables of interest (e.g., D_t for some t) can be obtained by marginalization and normalization from $\phi(U_t, A_t, D_t, \mathbf{X}_t)$. During the initialization phase, evidence (i.e., $\mathbf{X}_t$ and D_t when it is observed/available) is entered into the potentials. The forward pass propagates information forward through time, essentially calculating $\Pr[U_t, A_t, D_t, \mathbf{X}_t | \text{observables}_0, \ldots, \text{observables}_{t-1}]$ (expressed in denormalized potentials, rather than proper conditional probabilities). The backward pass propagates information back in time, to estab-

[3] So called because they appear between cliques in an intermediate step of compiling the graph structure of Fig. 10.1 into the inference algorithm of Fig. 10.2.

function `inference`
inputs:
$\mathbf{X}_t$ for all $t \in 0, \ldots, T$
D_t for some t
model parameters, $\Theta = \langle \theta_U, \theta_A, \theta_D, \theta_X, \theta_{U0}, \theta_{A0} \rangle$
outputs:
potentials $\Phi = \langle \phi(U_t, A_t, D_t, \mathbf{X}_t), \phi(U_t, A_t, U_{t+1}), \phi(A_t, U_{t+1}, A_{t+1}) \rangle$ for all t incorporating all observable evidence
// Initialization phase
for $t = 1, \ldots, T$
$\phi(U_t, A_t, D_t, \mathbf{X}_t) = \delta$(observed value of $\mathbf{X}_t$)
if D_t observed at this time step
$\phi(U_t, A_t, D_t, \mathbf{X}_t) *= \delta$(observed value of D_t)
endif
$\phi(U_t, A_t, D_t, \mathbf{X}_t) *= \Pr[D_t|\theta_D] * \Pr[\mathbf{X}_t|U_t, A_t, D_t, \theta_X]$
$\phi(U_t, A_t, U_{t+1}) = Pr[U_{t+1}|U_t, \theta_U] \quad \forall a_i \in A$
$\phi(A_t, U_{t+1}, A_{t+1}) = Pr[A_{t+1}|A_t, \theta_A] \quad \forall u_i \in U$
endfor
// Forward (collect evidence) pass
for $t = 0, \ldots, T-1$
$s(U_t, A_t) = \sum_{D_t} \sum_{\mathbf{X}_t} \phi(U_t, A_t, D_t, \mathbf{X}_t)$
$\phi(U_t, A_t, U_{t+1}) *= s(U_t, A_t) \quad \forall u_i \in U$
$s(A_t, U_{t+1}) = \sum_{U_t} \phi(U_t, A_t, U_{t+1})$
$\phi(A_t, U_{t+1}, A_{t+1}) *= s(A_t, U_{t+1}) \quad \forall a_i \in A$
$s(U_{t+1}, A_{t+1}) = \sum_{A_t} \phi(A_t, U_{t+1}, A_{t+1})$
$\phi(U_{t+1}, A_{t+1}, D_{t+1}, \mathbf{X}_{t+1}) *= s(U_{t+1}, A_{t+1}) \quad \forall d_i \in D, \mathbf{x}_j \in \mathbf{X}$
endfor
// Backward (distribute evidence) pass
for $t = T-1, \ldots, 0$
$s'(U_{t+1}, A_{t+1}) = s(U_{t+1}, A_{t+1})$
$s(U_{t+1}, A_{t+1}) = \sum_{D_{t+1}} \sum_{\mathbf{X}_{t+1}} \phi(U_{t+1}, A_{t+1}, D_{t+1}, \mathbf{X}_{t+1})$
$\phi(A_t, U_{t+1}, A_{t+1}) *= \frac{s(U_{t+1}, A_{t+1})}{s'(U_{t+1}, A_{t+1})} \quad \forall a_i \in A$
$s'(A_t, U_{t+1}) = s(A_t, U_{t+1})$
$s(A_t, U_{t+1}) = \sum_{A_{t+1}} \phi(A_t, U_{t+1}, A_{t+1})$
$\phi(U_t, A_t, U_{t+1}) *= \frac{s(A_t, U_{t+1})}{s'(A_t, U_{t+1})} \quad \forall u_i \in U$
$s'(U_t, A_t) = s(U_t, A_t)$
$s(U_t, A_t) = \sum_{U_{t+1}} \phi(U_t, A_t, U_{t+1})$
$\phi(U_t, A_t, D_t, \mathbf{X}_t) *= \frac{s(U_t, A_t)}{s'(U_t, A_t)} \quad \forall d_i \in D, \mathbf{x}_j \in \mathbf{X}$
endfor

Fig. 10.2. Pseudo-code for the batch-mode inference algorithm for the IDS model of Fig. 10.1. We have omitted the $t = 0$ initialization step which is similar to that given above, except using $\Pr[U_0|\theta_{U0}]$ and $\Pr[A_0|\theta_{A0}]$ rather than their temporal, Markov equivalents

function `learnEM`
inputs:
$\mathbf{X}_t$ for all $t \in 0, \ldots, T$
D_t for some t
outputs:
model parameters, $\Theta = \langle \theta_U, \theta_A, \theta_D, \theta_X, \theta_{U0}, \theta_{A0} \rangle$
// Initialization
set Θ arbitrarily, subject to $\Pr[\mathbf{X}_t|U_t, A_t, D_t = \texttt{USER}, \theta_X] = \Pr[\mathbf{X}_t|U_t, \theta_{X|D=\texttt{USER}}]$
and $\Pr[\mathbf{X}_t|U_t, A_t, D = \texttt{ATTACKER}, \theta_X] = \Pr[\mathbf{X}_t|A_t, \theta_{X|D=\texttt{ATTACKER}}]$
until parameters converge, do
// Expectation step
Φ=`inference`($\mathbf{X}$,D,Θ)
// Maximization step

$$\theta_D = \sum_t \sum_{U_t} \sum_{A_t} \sum_{\mathbf{X}_t} \phi(U_t, A_t, D_t, \mathbf{X}_t)$$

$$\theta_U = \sum_t \frac{\sum_{A_t} \phi(U_t, A_t, U_{t+1})}{\sum_{A_t} \sum_{U_{t+1}} \phi(U_t, A_t, U_{t+1})}$$

$$\theta_A = \sum t \frac{\sum_{U_{t+1}} \phi(A_t, U_{t+1}, A_{t+1})}{\sum_{U_{t+1}} \sum_{A_{t+1}} \phi(A_t, U_{t+1}, A_{t+1})}$$

$$\theta_{X|D=\texttt{USER}} = \sum_t \frac{\sum_{A_t} \phi(U_t, A_t, D_t=\texttt{USER}, \mathbf{X}_t)}{\sum_{A_t} \sum_{\mathbf{X}_t} \phi(U_t, A_t, D_t=\texttt{USER}, \mathbf{X}_t)}$$

$$\theta_{X|D=\texttt{ATTACKER}} = \sum_t \frac{\sum_{A_t} \phi(U_t, A_t, D_t=\texttt{ATTACKER}, \mathbf{X}_t)}{\sum_{A_t} \sum_{\mathbf{X}_t} \phi(U_t, A_t, D_t=\texttt{ATTACKER}, \mathbf{X}_t)}$$

`normalize`(θ_D)
`normalize`(θ_U)
`normalize`(θ_A)
`normalize`(θ_X)
enduntil

Fig. 10.3. Pseudo-code for the batch-mode EM-based learning algorithm for the IDS model of Fig. 10.1. Again, we have omitted the $t = 0$ parameter estimation which is similar to the above

lish $\Pr[U_t, A_t, D_t, \mathbf{X}_t|\text{observables}_0, \ldots, \text{observables}_T]$. After completion of the backward phase, $\Pr[D_t|\text{observables}_0, \ldots, \text{observables}_T]$ can be extracted from the appropriate joint potential.

Notice that the inference algorithm makes no distinction between training and testing phases – when D_t is observed, it is treated as an evidence variable and is fixed to the observed value. When D_t is unobserved, it is treated as a hidden variable and its probability distribution is estimated. This property is key in the use of this algorithm in a semi-supervised IDS framework in which data labels are available only intermittently.

The second critical algorithm is the learning algorithm, `learnEM()` (see Fig. 10.3), which updates the model parameters. While there are a number of possible methods for selecting model parameters in the face of missing data, a popular and convenient method is the EM algorithm [25, 270, 271].

This approach attempts to isolate maximum-likelihood parameters[4] through gradient descent in the parameter-likelihood space. It consists of an initialization phase and two steps that are iterated until the parameters converge to a local maximum. In the initialization phase, the parameters of the system, $\Theta = \langle \theta_A, \theta_U, \theta_D, \theta_X \rangle$, representing the conditional probability tables of the model, are set arbitrarily,[5] subject to certain consistency constraints that ensure that the D random variable functions correctly as a decision variable (i.e., when $D_t = \texttt{USER}$, $\mathbf{X}_t$ is influenced only by A_t, etc.) In the "Expectation" step, the inference algorithm is run to find the expected values of each hidden variable at each time step, represented by the potential functions. In the "Maximization" step, the expectations are used to update the model parameters by marginalizing potentials down to the relevant variables for each of the CPD θs and then normalizing to obtain valid probability distributions. For discrete variables, this operation turns out to be exactly maximum likelihood parameter estimation – thus the name of this step.

Pseudo-code for the learning algorithm is given in Fig. 10.3. In this figure, the `normalize`() function calculates the normalization of the functions $\Pr[D_t|\theta_D]$ et al. to ensure that they reflect well-formed probability functions. For discrete variables, this operation entails normalizing the conditional multinomial parameter vectors to unit mass. The conditions on the initialization of Θ and the two-part calculation of θ_X exist to enforce the semantics of D_t as an indicator of the source of the observed value $\mathbf{X}_t$. After convergence, the parameters Θ are a maximum-likelihood estimate, while the potentials Φ from the final inference step represent best estimates of the values of hidden variables. These final potentials can be used for alarm selection.

10.3.3 Action Selection

IDS actions are modeled by an additional exogenous control variable, $C_t \in \{\texttt{ALARM}, \texttt{NOALARM}\}$ (not pictured in Fig. 10.1). Proper selection of C_t depends on knowledge of the effect of C_t on the dynamics of the system of Fig. 10.1, as well as an instantaneous cost function, $v()$, that relates the *history* of system states and control actions to a utility. Under the principle of maximum utility, the goal of action selection is to choose C_t at every time so as to maximize some time-aggregate of cost, V. V relates the instantaneous cost, v, to the long-term effects of actions; optimizing V allows the agent to balance the two. For fixed, finite time-horizons a simple average cost (or, equivalently, total cost) is possible. In the IDS domain, however, time is usually taken to be unbounded (our computer may be on the network indefinitely) so the commonly employed *infinite horizon discounted* utility aggregate [272] is appropriate:

[4] Or maximum a posteriori (MAP) parameters, in a Bayesian framework.

[5] Typical choices are uniform or random initialization or sampling multinomial probability distributions from a Dirichlet prior distribution.

$$V^{\pi}(U_0, A_0, D_0, \mathbf{X}_0) = \mathrm{e}\left[\sum_{t=0}^{\infty} \gamma^t v(U_t, A_t, D_t, \mathbf{X}_t)\right]$$
$$= \sum_{t=0}^{\infty} \sum_{u_i, a_j, d_k, x_l} \left(\gamma^t v(u_i, a_j, d_k, x_l) \cdot \right.$$
$$\left. \Pr[U_t = u_i, A_t = a_j, D_t = d_k, \mathbf{X}_t = x_l | U_0, A_0, D_0, \mathbf{X}_0, \pi]\right) \ , \quad (10.1)$$

where $0 \leq \gamma < 1$ is a discount factor introduced to prevent the infinite sum from diverging, and π indicates the *policy* by which control actions are chosen.

Most common approaches to IDS decision-making in the literature depend on two (often implicit) assumptions:

Assumption 1 *$v()$ is purely a function of the instantaneous state of the system:*

$$v_t(D_t, C_t) = \begin{cases} 0 & \text{if } C_t = D_t \ ; \\ v_{\mathrm{FP}} & \text{if } C_t = \texttt{ALARM} \text{ and } D_t = \texttt{USER} \ ; \\ v_{\mathrm{FN}} & \text{if } C_t = \texttt{NOALARM} \text{ and } D_t = \texttt{ATTACKER} \ , \end{cases}$$

where $C_t = D_t$ denotes "($C_t = \texttt{ALARM}$ & $D_t = \texttt{ATTACKER}$) OR ($C_t = \texttt{NOALARM}$ & $D_t = \texttt{USER}$)". The values v_{FP} and v_{FN} are the costs of false alarms and false accepts, respectively.

Assumption 2 *The state of the system, $\langle U_t, A_t, D_t, \mathbf{X}_t \rangle$, at all times is independent of the control action (i.e., the alarm state does not affect either the user or the attacker's actions).*

Under these two assumptions, the entire IDS task reduces to a state monitoring problem. Assumption 2 removes the dependence on π from (10.1), making system model of Fig. 10.1 an uncontrolled, partially observable Markov chain. In this case, our job is to monitor the state of the variable D (e.g., by maximum likelihood or Viterbi decoding) as closely as possible; because the dynamics of the system are unaffected by C_t, the optimal decision at time t is just given by the estimate of D_t, possibly weighted by the costs v_{FN} and v_{FP}. The experimental work we present in Sect. 10.4 employs the semi-supervised learning approach under these assumptions, but we now demonstrate that relaxing these assumptions leads to a new class of models for the IDS problem.

10.3.4 Relaxing the Cost Function

Assumptions 1 and 2 are unrealistically strong. Assumption 2, for example, directly contradicts the fundamental assumption of intrusion detection – we fervently *hope* that an effective IDS will allow us to stop attackers, while alarms disrupt the work of the normal user. Assumption 1 is also problematic in realistic conditions. For example, additional alarms during a single attack session may contribute only incremental utility above the value of the

first alarm, and may even contribute *negative* utility (by annoying the security officer). This is effectively the argument that Fawcett and Provost [249] make when arguing for modeling the IDS problem as a change detection task. We [248] and Fawcett and Provost have also proposed time to alarm (time between onset of an attack and its detection) as an important cost criterion.

Fawcett and Provost propose a utility function for this domain of the form $v_t(H_t) = f(\tau, C_t, H_t)$, where $H_t = \langle \mathbf{X}_0, D_0, C_0, \ldots, \mathbf{X}_t, C_t, D_t \rangle$ is the *history* of attacks and alarms to date. The variable τ indicates the elapsed time since the onset of (the most recent) attack: $\tau = \min_{t'}\{t - t' : D_{t'} = \texttt{ATTACKER} \;\&\; D_{t'-1} = \texttt{USER}\}$. The cost of time to alarm is expressed through the dependence on τ. By making τ integer, rather than binary (recent missed attack or not), we can model that the cost of missing the start of an attack varies nonlinearly with age of the attack. For example, at the very beginning of an attack, the attacker has had little time to do damage; as the attacker is allowed more time unchecked, more damage can be done, until some asymptotic point at which no additional damage can be done.

We can address this function by incorporating τ as an explicit variable in our generative model, depending on C_t, D_t, and τ_{t-1}. Initially, $\tau_0 = \phi$ (a special, undetermined value). As soon as $D_t = \texttt{ATTACKER}$, τ_t becomes 0. Thereafter, τ is incremented every time step until $C_t = \texttt{ALARM}$, at which point it is reset to ϕ. Although τ_t is deterministically generated by its parent variables, by virtue of D_t's partial observability τ_t is also a partially observed variable. The updated model, including the control variable, C_t, its influence on τ_t, and potential influences that we might choose to model (dashed line arrows), is given in Fig. 10.4.

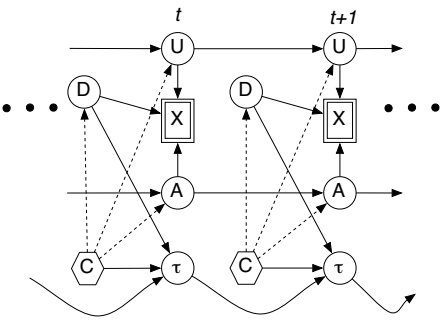


Fig. 10.4. Extended IDS model, incorporating the control variable (C) and time-to-alarm variable (τ). Solid arrows indicate key dependencies; dashed-line arrows indicate dependencies that we may or may not choose to model, such as the influence of alarms on the user's activities

The instantaneous utility of the system is now a function of τ. For example, Fawcett and Provost employ a linear utility that we would write as[6]

$$v(\tau_t) = \begin{cases} 0 & \text{if } C_t = \texttt{NOALARM} \text{ and } \tau_t = \phi \ ; \\ 5 & \text{if } C_t = \texttt{ALARM} \text{ and } \tau_t = \phi \ ; \\ 0.4 & \text{if } \tau \neq \phi \ . \end{cases}$$

Other, nonlinear time-to-alarm functions are also possible.

τ accounts for time-to-alarm; if we wish to also account for the nonlinear utility of multiple alarms during a single attack session, we would have to introduce an additional variable to the system to count such alarms. More complex models are also possible, incorporating additional knowledge about the `USER` or `ATTACKER`, structure of the observation vector, more complex cost functions, etc. We omit such extensions here for lack of space, but they do not change our fundamental observations.

The presence of τ increases the cardinality of the state space by as much as $|\mathbb{Z}|$ in the worst case.[7] More critically, however, τ couples C_t to the data model. Specifically, if τ_t is observed for any t (e.g., if any `USER` data is ever labeled or any `ATTACKER` data is ever identified), then C_i and D_i become d-connected for $i \leq t$ [273, 274]. That is, observing τ_t renders our estimates of the probability distribution over D_i dependent on our choice of C_i over time. This is a consequence of the statistical reasoning embodied in the update equations for a graphical model of this structure, but it reflects commonsense reasoning about the domain. The observed value of τ_t can be "explained" either through the influence of C_t or through the dependence on D_t. If we fix C_t as well (by picking an action), that accounts for (some of) the value of τ_t, which in turn forces us to reassess our estimate of the probability distribution over D_t (and, indirectly, of $\mathbf{X}$, A, and U as well).[8] This interplay means that C_t intimately affects our understanding of the state of the system (and, therefore, our future choice of actions) even if C_t does not *directly* influence the state of the user or attacker. However, as we argued above, the presence of alarms is likely to affect U and A directly (represented by the dashed influence lines in Fig. 10.4).

[6] To be precise, our utility is still nonlinear because of the discount factor γ in (10.1) that Fawcett and Provost do not use. We argue, however, that this factor is necessary to prevent potentially unbounded utilities in the infinite horizon IDS model. For finite horizons, our function can be made arbitrarily close to their linear function via an appropriate choice of γ.

[7] In principle, this is an infinite state space, but this is probably not a severe practical difficulty in itself. In practice, we can likely safely truncate τ to a finite range representing the maximum damage (length of attack) that an attacker can do. Alternatively, we could choose a parametric form for τ and maintain a distribution over parameters rather than a full multinomial.

[8] If this reasoning still seems somewhat convoluted, we refer the reader to a very accessible introduction to reasoning in Bayesian networks by Charniak [274] that describes this case in detail.

The resulting system is formally a partially observable Markov decision process (POMDP) [275]. Because the action variable C_t now affects the dynamics of the system (the evolution of the probability distribution, or *belief state*, over τ and, through it, D), a pure monitoring approach is no longer sufficient. Optimal decision-making involves searching the space of policies π to find one which optimizes the time-aggregate utility V given by (10.1).

Unfortunately, for the partially observable case, exact decision-making is known to be quite difficult. Even given a complete model of the system, including all parameter values, finding exactly or even boundedly approximate optimal policies over such a space (the planning problem) is known to be difficult – formally uncomputable [275] and EXP-hard [276], respectively. Finding such a policy when the system parameter values are unknown is the *reinforcement learning* problem for partially observable environments. No ideal learning algorithm is known for reinforcement learning in POMDPs, though it remains an active area of research (see the Kaelbling et al. survey paper [275] for a summary of proposed learning methods for this environment). While we do not address solving the POMDP version of the IDS problem in this chapter, it remains an important area of ongoing work for us, and we hope to apply the active research from the reinforcement learning literature to this domain in the near future.

The critical observation here is that the monitoring case that arises from Assumptions 1 and 2 is the simplest of a set of complex learning and planning tasks rooted in the POMDP formulation. The full-blown POMDP framework quickly emerges as a consequence of employing complex cost functions – cost functions that depend on the time since the onset of attack require us to maintain a probability distribution over that elapsed time. Simplifying the model of Fig. 10.4 by, e.g., removing the Markov dependencies in U or A will not help – the critical factor is the existence of τ and that it couples C_t into the rest of the system.

In practice, this leaves us with two choices:

1. Accept simpler cost functions, and the representational paucity that comes with them (an insensitivity to the differential impacts of different attacks, multiple alarms for the same incident, etc.) in favor of tractable exact monitoring.
2. Employ more expressive cost functions, accept the more difficult and intractable POMDP formulation, and seek tractable approximations for learning and reasoning in these complex models.

In this chapter, we take the former approach, but we are exploring the latter in ongoing work. Heuristic methods for reasoning in POMDPs have had great practical success recently, even when they do not have guaranteed approximation bounds. For example, hierarchical POMDPs have had strong success in robotics [277] and may be appropriate for modeling complex, structured decision-making in the IDS domain.

While the POMDP formulation is heavyweight and somewhat intimidating, bringing it to the IDS task does offer a number of theoretical and practical advantages. We can conceive of a hierarchy of increasingly sophisticated models employing more observables, models of multiple users, more complex cost functions, etc. By situating the IDS problem in the POMDP framework with DBN models of transition probabilities, we can assess the computational complexity of learning and reasoning in each of these models, as well as the effect of various approximations. Thus, we have a principled framework in which to trade off IDS model complexity for tractability and accuracy. Furthermore, in the POMDP paradigm, cost is an intrinsic part of the learning *task* and is divorced from the learning algorithm. POMDPs employ an extremely general cost framework that subsumes many cost functions of practical interest – it is not necessary to reengineer a POMDP learning algorithm to accommodate each specific cost function. Thus, costs could, in principle, be set on a site-specific basis in response to security policy. Finally, we can bring to bear the rich set of tools that have been developed in past years for working with POMDPs.

10.4 Experiments

For our initial work, we treat only the simplest version of our framework, adopting the semi-supervised component of the framework and neglecting the extended POMDP formulation. We adopt the generative model of Fig. 10.1 and Assumptions 1 and 2 directly. This is nearly the simplest model possible under our framework,[9] but still represents an extension to existing systems by explicitly modeling the attacker (A_t) and allowing an intermittently observed decision variable (D_t).

Because this simplified model is an uncontrollable process, learning reduces to inferring the parameters of the generative process – the transition CPDs $\Pr[U_{t+1}|U_t]$ and $\Pr[A_{t+1}|A_t]$; the variable priors $\Pr[D_t]$, $\Pr[U_0]$, and $\Pr[A_0]$; and the observation/sensor function $\Pr[\mathbf{X}_t|U_t, A_t, D_t]$. We impose the constraint that

$$Pr[\mathbf{X}_t|U_t, A_t, D_t] = \begin{cases} Pr[\mathbf{X}_t|U_t] \text{ if } D_t = \texttt{USER} ; \\ Pr[\mathbf{X}_t|A_t] \text{ if } D_t = \texttt{ATTACKER} , \end{cases}$$

to reflect the role of D_t as a decision variable. As noted in Sect. 10.3.3, action selection in this case is simply estimation of D_t.

We compile this dynamic Bayesian network model into a junction tree [278] in which inference can be carried out in time $O(T|U||A||D|)$ for a T-time-step data set. Given a data sequence, including $\mathbf{X}$ at all time steps and D at

[9] We could relax Markov assumptions further, allowing U_t and A_t to be multinomial variables, but we have shown the temporal coupling of user hidden state variables to be important for modeling many users in previous work [248].

some time steps, we learn the parameters of this model via the EM algorithm, employing inference on the junction tree for the E step. For observation vectors $\mathbf{X}$ with only discrete elements, the M step is multinomial density estimation from the inferred distribution.[10] We take the maximum likelihood estimates of D_t from the inference step as our action choices, weighted by $v_{\mathrm{FN}}/v_{\mathrm{FP}}$ for ROC analysis.

10.4.1 Data Set

For our tests, we applied our semi-supervised IDS to the "user masquerading" data set generated by Schonlau et al. [262] for their comparative study of anomaly detection techniques. This data comprises individual time series from 50 different UNIX users. Each user's time series consists of 15,000 UNIX commands such as `ls` or `emacs`. Command arguments are omitted. In each time series, the first 5,000 commands are known to be valid `USER` data, while the remainder of the data is unlabeled. The unlabeled data is "corrupted" with the introduction of simulated intrusions – commands drawn from a different user's activities ("masquerades", in Schonlau's parlance). For each block of 100 commands in the unlabeled section, an intrusion is initiated with probability 0.01 or continued (if the previous block was an intrusion) with probability 0.8. Approximately 5% of the unlabeled data is intruder data. The true labels for the unlabeled sections of the data are available separately. For more details on this data set, please refer to Schonlau et al.'s original report [262].

The systems examined in the Schonlau et al. report were all pure anomaly detection systems and were employed in a strict training/testing mode – training on the labeled section of the data (first 5,000 tokens) and testing on the remaining 10,000, unlabeled tokens. This case is a limiting case of our model, in which we run it in a pure anomaly detection mode with no real semi-supervised learning taking place (though our model can still adapt parameters using completely unlabeled data). For comparison's sake, we demonstrate our model in this mode, but the real strength of our approach is its ability to learn continually in a semi-supervised way, and to employ whatever fragments of labeled data are available. (We discuss some ways in which such data might become available in "field conditions" in Sect. 10.1.) This represents a qualitative break from the earlier, pure anomaly detection systems, and we demonstrate that, when such data is available, our system can exploit it to advantage when the previous systems would not have.

To examine the performance of our system in semi-supervised conditions, we constructed partially labeled data sets from the original Schonlau et al. data by labeling a small fraction of the unlabeled data. Specifically, starting at the beginning of the unlabeled section ($t = 5,001$), we selected subsequent

[10] Observation vectors containing continuous variables are quite conceivable in this domain (time stamps, latencies, memory loads, etc.), but such variables complicate inference considerably and we do not address them in this chapter.

windows of data with length chosen from a geometrically distributed random variable with parameter 1/20. At each window of data, we fully labeled the window with probability p_u if it began with a normal user datum and with probability p_a if it began with an attacker datum. A window labeled in this way might contain both user and attacker data. Note that while the original data contained intrusions in units of 100 time-point blocks, we did not provide this information to our learning algorithm.

We examined the effects of changing the amount of labeled data available to the semi-supervised IDS by varying p_u across 0.1%, 1%, and 10%, while we varied p_a across 1%, 10%, and 50%. Typically, `USER` data has a much higher frequency than `ATTACKER` data, so it is plausible that a smaller fraction of it would be labeled. We also examined the pure anomaly detection case, corresponding to $p_u = p_a = 0$. We applied the semi-supervised IDS learner to the 50 partially labeled time series, learning a model for each time series with $|U| = 10$ and $|A| = 4$, reflecting the relative paucity of labeled attacker data. The number of states in the `USER` model was chosen based on previous research that found ten-state HMMs to be reasonable anomaly detection models for many users [248]. After learning the model parameters via EM, we assigned the maximum likelihood labels to the remaining unlabeled data (corresponding to the cost function $v_{\mathrm{FN}} = v_{\mathrm{FP}}$ in Assumption 1). The selection of costs in this model displaces the selection of "acceptable false-alarm rate" that is used in a number of pure anomaly detection systems, and we examined a spectrum of cost ratios, $v_{\mathrm{FN}}/v_{\mathrm{FP}}$, for the purposes of ROC analysis.

Table 10.1. Performance of the semi-supervised IDS for varying amounts of labeled `USER` and `ATTACKER` data. (**a**) Accuracy. (**b**) False-alarm/false-accept rates. The "pure anomaly detection" mode corresponds to $p_u = p_a = 0$. All figures given are percentages

(**a**)

		p_a			
		0%	1%	10%	50%
p_u	0%	93.7	–	–	–
	0.1%	–	93.9	94.1	96.3
	1%	–	94.6	94.7	96.5
	10%	–	95.2	95.4	97.4

(**b**)

	p_a			
	0%	1%	10%	50%
0%	2.15/49.43	–	–	–
0.1%	–	2.21/50.53	2.60/46.75	2.45/34.89
1%	–	1.53/50.07	1.79/49.07	2.17/33.60
10%	–	0.31/51.98	0.52/48.90	1.04/36.14

10.4.2 Results

Average accuracy and false-alarm/false-accept figures for classifying unlabeled time points are given in Table 10.1. The "pure anomaly detection" case (in which the system is trained only on the initial 5,000 labeled data points that were used by the Schonlau et al. systems for training) is given in the upper left cell of each table.

We see that as more labeled data becomes available (moving from upper left to lower right in each table), the IDS does increasingly well overall, demonstrating that the semi-supervised approach is behaving as we would expect. In particular, false-alarm rates drop with increasing levels of labeled `USER` data, while false-accept rates drop with increasing levels of labeled `ATTACKER` data. There is a tension between the two, reflected in the difference between the lower right cell ($p_u = 10\%$, $p_a = 50\%$) and first cells in its row ($p_a = 1\%$) and column ($p_u = 0.1\%$) of Table 10.1b. This effect is not unexpected – increasing the relative proportion of `USER` to `ATTACKER` data, or vice versa (head of a row or column, respectively) should naturally bias the model in favor of one class or another by skewing the class priors.

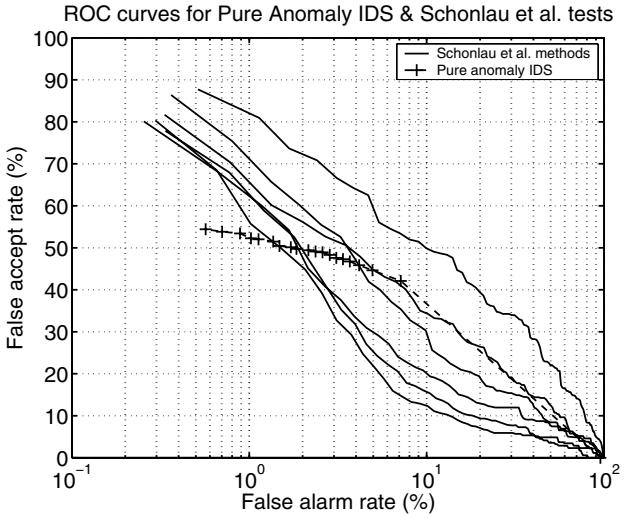
The performance of the pure anomaly detection mode of this IDS learner is nearly indistinguishable from that of the semi-supervised learner when run on the most sparsely labeled data ($p_u = 0.1\%$; $p_a = 1\%$), indicating that the pure anomaly detection mode is the limiting case of the semi-supervised learner, as expected. Note that the lower right cell still dominates the upper left, indicating that uniformly increasing the available labeled data does, in fact, aid the learner.

Table 10.2. False-alarm and false-accept rates for the six methods examined by Schonlau et al. All figures given are percentages

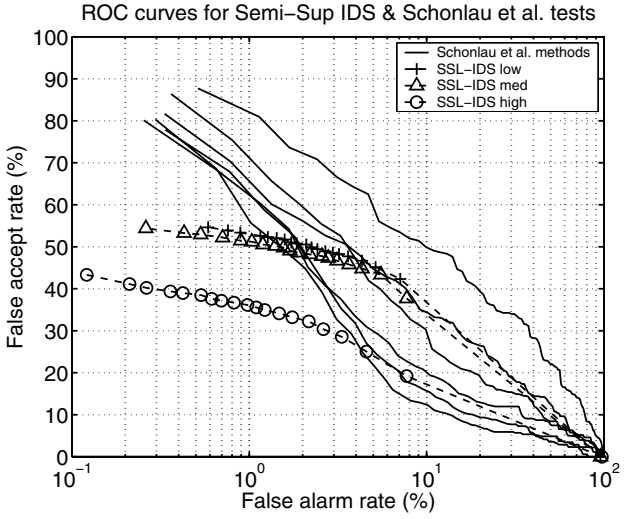
Model Name	f. alarm	f. accept
One-step Markov	6.7	30.7
Markov chain + drift	2.7	58.9
Uniqueness	1.4	60.6
Order-10 Markov	3.2	50.7
Compression	5.0	65.8
Instance-based	3.7	63.2

For comparison, Table 10.2 gives the false-alarm/false-accept rates reported by Schonlau et al. for the six systems that they applied to this data. Even in the the pure anomaly detection test and the most parsimoniously labeled data experiment ($p_u = 0.1\%$; $p_a = 1\%$), the semi-supervised learner dominates four of the six and is beaten only by "Uniqueness" on false-alarm rate and "Bayes one-step Markov" on false-accept rate. In more generously labeled circumstances, the semi-supervised learner's lead improves further.

Figure 10.5 displays ROC plots of the performance of the six methods that Schonlau et al. examined (solid lines) and of our semi-supervised learner IDS (SSL-IDS), run in pure anomaly detection mode (a) and with varying degrees of partially labeled data (b). (Broken lines with symbols indicate SSL-IDS.) Note that these plots use modified ROC axes, with false-alarm rate on the abscissa and false-accept rate on the ordinate. In these axes, the lower left corner is the optimal condition (no error of either type) and curves closer to the origin dominate curves farther from it. False-alarm rates are



(**a**)



(**b**)

Fig. 10.5. ROC curves for the six anomaly detection methods examined by Schonlau et al. and our semi-supervised learner IDS (SSL-IDS). Note that plots use non-standard ROC axes. The optimal condition is the origin (no error of either type). (**a**) When SSL-IDS is run in "pure anomaly detection" mode. (**b**) SSL-IDS run with three different fractions of partially labeled data

given on a logarithmic scale to improve legibility. The results for the SSL-IDS are given for $\langle p_u = 0.1\%, p_a = 1\% \rangle$ (SSL-IDS LOW; broken line with plus signs); $\langle p_u = 1\%, p_a = 10\% \rangle$ (SSL-IDS MED; broken line with triangles); and $\langle p_u = 10\%, p_a = 50\% \rangle$ (SSL-IDS HIGH; broken line with circles). The pure anomaly detection curve is not plotted with the SSL-IDS curves because it is visually indistinguishable from SSL-IDS LOW (although the curves do actually vary slightly).

Figure 10.5a shows that the performance of the IDS, when run in pure anomaly detection mode, is largely consistent with the performance of the six anomaly detectors tested by Schonlau et al. The learner proposed in this chapter does appear to have stronger performance at lower false-alarm rates, probably because it learns continually, even on the unlabeled data, and is able to refine its model of the normal user when biased in favor of believing that the majority of the unlabeled data originates from that source (i.e., the bias imposed by the high cost of false alarms at the left end of the ROC curves).

Figure 10.5b shows two important features of the semi-supervised IDS. First, increasing the fraction of labeled training data does uniformly improve performance, as the more richly labeled SSL-IDS curves fully dominate the more scarcely labeled curves. Second, and more importantly, while the semi-supervised learner does not fully dominate the six methods reported in Schonlau et al., it *does* outperform them at low false-alarm rates (probably for the reasons given above, as well as its ability to exploit bits of labeled data that are introduced later in the data stream). This is significant because false-alarm rate is generally seen as the greatest barrier to practical application of adaptive IDS technologies. Even modest false-alarm rates can yield thousands or millions of false alarms per day on large networks, so it is imperative that we find methods that improve false-alarm rates. SSL-IDS's superiority at low false-alarm rates is quite promising in this regard, as is the fact that its performance scales with increasing amounts of labeled data. While we can never hope to have large quantities of labeled data in practice, it is plausible that we can take advantage of increasing amounts of labeled hostile data over time to substantially improve the performance of our intrusion detection systems.

10.5 Conclusions and Future Work

In this chapter, we have demonstrated a novel approach to intrusion detection that offers a number of advantages over current approaches. By posing the problem in a semi-supervised learning framework, we are able to exploit both known normal and known hostile data as well as to operate in unlabeled data conditions. Doing so eliminates the distinction between misuse and anomaly detection while preserving some of the strengths of both methods. The distinction between "training" and "testing" data is also eliminated, allowing a more

flexible intrusion detection system whose model can be constantly updated as new data is seen.

We demonstrated that the resulting semi-supervised IDS performs strongly relative to a number of previously proposed pure anomaly detection systems on a large, human-generated, command-line intrusion detection task. Pure anomaly detection is a limiting case of our model, achieved when only a fixed block of single-class training data is available. When additional data becomes available (e.g., as a result of an exploit report or a forensic analysis of an attack that the IDS initially missed), the system adapts to this data and takes advantage of it.

We also made explicit a pair of common assumptions in the anomaly detection literature and showed that they were equivalent to the simplest of a space of decision-theoretic models rooted in the POMDP formulation. This observation clarifies the formal difficulty of the optimal solution of the IDS problem under a variety of cost functions and domain models, and opens up the IDS learning task to the entire space of POMDP learning algorithms.

It is important to note that the introduction of the POMDP formalism is not a *choice*, but a position that is mathematically *required* in order to attain more powerful cost functions. Nonetheless, employing this formalism does offer a number of advantages. It allows us to explicitly analyze the complexity of optimal decision-making for the IDS domain under a wide spectrum of observable variables and cost functions, it decouples the cost function from the learning algorithm, and it gives us access to a large body of literature on planning and learning in POMDPs.

We are currently pursuing this direction and examining the applicability of various reinforcement learning and POMDP approximate planning methods in this domain. A second direction of our near future research is handling more complex intrusion data sets such as network intrusion detection (e.g., in the DARPA/Lincoln Labs IDS data set [261]). While this chapter examined a single channel of observable data (the command issued at time t), network packet data typically includes many channels of relevant data such as time stamps, host and destination IP address, TCP control bits, higher-level protocol information, etc. Taking advantage of these additional channels requires a sensor-fusion type of approach. While dependencies among these channels can be described in our DBN framework easily enough, the additional channels do seriously complicate learning and decision-making. Overcoming these issues will likely require large approximations – either existing POMDP methods, or new approaches specific to this domain. An open issue is whether it is tractable or desirable to attain a bounded approximation to this model, or if unboundable heuristic approaches work well in practice. Finally, we are interested in modeling the effects of alarms on `USER` activities (e.g., the dotted-line arrows in Fig. 10.4) as well as more complex decisions than simple `ALARM`/`NOALARM` states, such as variable assessments of threat level, passive alarms vs. active interruption of service, etc.

References

[1] Bishop, M.: Computer security: Art and science. Addison-Wesley, Boston, MA (2003)

[2] Mitchell, T.M.: Machine learning. McGraw-Hill, New York, NY (1997)

[3] Hand, D., Mannila, H., Smyth, P.: Principles of data mining. MIT Press, Cambridge, MA (2001)

[4] McCumber, J.R.: Information systems security: A comprehensive model. In: Proceedings of the Fourteenth NIST-NCSC National Computer Security Conference. National Institute of Standards and Technology, Gaithersburg, MD (1991) 328–337

[5] Kahn, D.: The codebreakers. Macmillan, New York, NY (1967)

[6] Matsumoto, T., Matsumoto, H., Yamada, K., Hoshino, S.: Impact of artificial gummy fingers on fingerprint systems. In: Proceedings of the SPIE International Symposium on Electronic Imaging: Optical Security and Counterfeit Deterrence Techniques. Volume 4677 (2002) 275–289

[7] Thalheim, L., Krissler, J., Ziegler, P.M.: Body check: Biometric access protection devices and their programs put to the test. c't Magazine (November 2002) 114. `http://www.heise.de/ct/english/02/11/114/`

[8] Denning, D.E., MacDoran, P.F.: Location-based authentication: Grounding cyberspace for better security. In Denning, D.E., Denning, P.J., eds.: Internet Besieged: Countering Cyberspace Scofflaws. ACM Press/Addison-Wesley, New York, NY (1998) 167–174. Reprint from Computer fraud and security, Elsevier Science, Ltd. (1996)

[9] Schneier, B.: Applied cryptography. 2nd edn. John Wiley & Sons, New York, NY (1996)

[10] Gligor, V.: A guide to understanding covert channel analysis of trusted systems. Technical Report NCSC-TG-030, National Computer Security Center, Fort George G. Meade, Maryland (1993)

[11] National Security Agency: TEMPEST fundamentals. Technical Report 5000, Fort Meade, MD (1982)

[12] Kuhn, M., Anderson, R.: Soft tempest: Hidden data transmission using electromagnetic emanations. In: Information hiding. Volume 1525 of Lecture Notes in Computer Science. Springer-Verlag, New York, NY (1998) 124–142. Second International Workshop, IH '98, Portland, OR, April 14–17, 1998. Proceedings

[13] Kim, G.H., Spafford, E.H.: Writing, supporting, and evaluating Tripwire: A publically available security tool. In: Proceedings of the Symposium on Unix Applications Development. USENIX Association, Berkeley, CA (1994) 89–108

[14] Alves-Foss, J.: Security implications of quantum technologies. In: Proceedings of the Twenty-first NIST-NCSC National Information Systems Security Conference. National Institute of Standards and Technology, Gaithersburg, MD (1998) 196–202

[15] Gordon, L., Loeb, M., Lucyshyn, W., Richardson, R.: 2004 CSI/FBI computer crime and security survey. Technical report, Computer Security Institute, San Francisco, CA (2004)

[16] Duda, R.O., Hart, P.E.: Pattern classification and scene analysis. John Wiley & Sons, New York, NY (1973)

[17] Duda, R.O., Hart, P.E., Stork, D.G.: Pattern classification. 2nd edn. John Wiley & Sons, New York, NY (2000)

[18] Fukunaga, K.: Introduction to statistical pattern recognition. Academic Press, Boston, MA (1990)

[19] McLachlan, G.J.: Discriminant analysis and statistical pattern recognition. John Wiley & Sons, New York, NY (1992)

[20] Grossman, D., Frieder, O.: Information retrieval: Algorithms and heuristics. Kluwer Academic Publishers, Boston, MA (1998)

[21] Weiss, S.M., Kulikowski, C.A.: Computer systems that learn: Classification and prediction methods from statistics, neural nets, machine learning and expert systems. Morgan Kaufmann, San Francisco, CA (1992)

[22] Langley, P.: Elements of machine learning. Morgan Kaufmann, San Francisco, CA (1996)

[23] Han, J., Kamber, M.: Data mining: Concepts and techniques. Morgan Kaufmann, San Francisco, CA (2001)

[24] Witten, I.H., Frank, E.: Data mining: Practical machine learning tools and techniques with Java implementations. Morgan Kaufmann, San Francisco, CA (2000)

[25] Hastie, T., Tibshirani, R., Friedman, J.: The elements of statistical learning: Data mining, inference, and prediction. Springer-Verlag, New York, NY (2001)

[26] Fayyad, U.M., Irani, K.B.: Multi-interval discretization of continuous-valued attributes for classification learning. In: Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence. Morgan Kaufmann, San Francisco, CA (1993) 1022–1027

[27] Huang, C.C., Lee, H.M.: A grey-based nearest neighbor approach for missing attribute value prediction. Applied Intelligence **20** (2004) 239–252
[28] Somol, P., Pudil, P., Kittler, J.: Fast branch & bound algorithms for optimal feature selection. IEEE Transactions on Pattern Analysis and Machine Intelligence **26** (2004) 900–912
[29] Blum, A.L., Langley, P.: Selection of relevant features and examples in machine learning. Artificial Intelligence **97** (1997) 245–271
[30] Brodley, C.E., Friedl, M.A.: Identifying mislabeled training data. Journal of Artificial Intelligence Research **11** (1999) 131–167
[31] Pazzani, M., Merz, C., Murphy, P., Ali, K., Hume, T., Brunk, C.: Reducing misclassification costs. In: Proceedings of the Eleventh International Conference on Machine Learning. Morgan Kaufmann, San Francisco, CA (1994) 217–225
[32] Provost, F., Fawcett, T.: Robust classification for imprecise environments. Machine Learning **42** (2001) 203–231
[33] Schlimmer, J.C., Fisher, D.: A case study of incremental concept induction. In: Proceedings of the Fifth National Conference on Artificial Intelligence. AAAI Press, Menlo Park, CA (1986) 496–501
[34] Widmer, G., Kubat, M.: Learning in the presence of concept drift and hidden contexts. Machine Learning **23** (1996) 69–101
[35] Maloof, M.A., Michalski, R.S.: Selecting examples for partial memory learning. Machine Learning **41** (2000) 27–52
[36] Hulten, G., Spencer, L., Domingos, P.: Mining time-changing data streams. In: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM Press, New York, NY (2001) 97–106
[37] Maloof, M.A., Michalski, R.S.: Incremental learning with partial instance memory. Artificial Intelligence **154** (2004) 95–126
[38] Kolter, J.Z., Maloof, M.A.: Dynamic weighted majority: A new ensemble method for tracking concept drift. In: Proceedings of the Third IEEE International Conference on Data Mining. IEEE Press, Los Alamitos, CA (2003) 123–130
[39] Lane, T., Brodley, C.E.: Approaches to online learning and concept drift for user identification in computer security. In: Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining. AAAI Press, Menlo Park, CA (1998) 259–263
[40] Aha, D.W., Kibler, D., Albert, M.K.: Instance-based learning algorithms. Machine Learning **6** (1991) 37–66
[41] Friedman, J.H., Bentley, J.L., Finkel, R.A.: An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software **3** (1977) 209–226
[42] John, G.H., Langley, P.: Estimating continuous distributions in Bayesian classifiers. In: Proceedings of the Eleventh Conference on Uncertainty

in Artificial Intelligence. Morgan Kaufmann, San Francisco, CA (1995) 338–345
[43] Domingos, P., Pazzani, M.: Beyond independence: Conditions for the optimality of the simple Bayesian classifier. In: Proceedings of the Thirteenth International Conference on Machine Learning. Morgan Kaufmann, San Francisco, CA (1996) 105–112
[44] Langley, P., Sage, S.: Scaling to domains with many irrelevant features. In Greiner, R., ed.: Computational Learning Theory and Natural Learning Systems. Volume 4. MIT Press, Cambridge, MA (1997) 17–30
[45] Domingos, P., Pazzani, M.J.: On the optimality of the simple Bayesian classifier under zero-one loss. Machine Learning **29** (1997) 103–130
[46] Fisher, R.: The use of multiple measurements in taxonomic problems. Annals of Eugenics **7** (1936) 179–188
[47] Minsky, M., Papert, S.: Perceptrons: An introduction to computational geometry. MIT Press, Cambridge, MA (1969)
[48] Boser, B.E., Guyon, I., Vapnik, V.: A training algorithm for optimal margin classifiers. In: Proceedings of the Fourth Workshop on Computational Learning Theory. ACM Press, New York, NY (1992) 144–152
[49] Michalski, R.S., Kaufman, K.: The AQ-19 system for machine learning and pattern discovery: A general description and user's guide. Reports of the Machine Learning and Inference Laboratory MLI 01-4, Machine Learning and Inference Laboratory, George Mason University, Fairfax, VA (2001)
[50] Clark, P., Niblett, T.: The CN2 induction algorithm. Machine Learning **3** (1989) 261–284
[51] Cohen, W.W.: Fast effective rule induction. In: Proceedings of the Twelfth International Conference on Machine Learning. Morgan Kaufmann, San Francisco, CA (1995) 115–123
[52] Holte, R.C.: Very simple classification rules perform well on most commonly used datasets. Machine Learning **11** (1993) 63–91
[53] Quinlan, J.R.: C4.5: Programs for machine learning. Morgan Kaufmann, San Francisco, CA (1993)
[54] Iba, W.F., Langley, P.: Induction of one-level decision trees. In: Proceedings of the Ninth International Conference on Machine Learning. Morgan Kaufmann, San Francisco, CA (1992) 233–240
[55] Utgoff, P.E.: ID5: An incremental ID3. In: Proceedings of the Fifth International Conference on Machine Learning. Morgan Kaufmann, San Francisco, CA (1988) 107–120
[56] Utgoff, P.E., Berkman, N.C., Clouse, J.A.: Decision tree induction based on efficient tree restructuring. Machine Learning **29** (1997) 5–44
[57] Domingos, P., Hulten, G.: Mining high-speed data streams. In: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM Press, New York, NY (2000) 71–80
[58] Agrawal, R., Imielinski, T., Swami, A.: Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIG-

MOD International Conference on Management of Data. ACM Press, New York, NY (1993) 207–216
[59] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proceedings of the Twentieth International Conference on Very Large Data Bases. Morgan Kaufmann, San Francisco, CA (1994) 487–499
[60] Michalski, R.S.: Inferential theory of learning: Developing foundations for multistrategy learning. In Michalski, R.S., Tecuci, G., eds.: Machine Learning: A Multistrategy Approach. Volume 4. Morgan Kaufmann, San Francisco, CA (1994) 3–61
[61] Cohen, L.J.: An introduction to the philosophy of induction and probability. Clarendon Press, Oxford (1989)
[62] Efron, B., Tibshirani, R.J.: Improvements on cross-validation: The .632+ bootstrap method. Journal of the American Statistical Association: Theory and Methods **92** (1997) 548–560
[63] Efron, B., Tibshirani, R.J.: An introduction to the bootstrap. Chapman & Hall, New York, NY (1993)
[64] Efron, B.: Estimating the error rate of a prediction rule: Improvement on cross-validation. Journal of the American Statistical Association **78** (1983) 316–331
[65] Beiden, S.V., Maloof, M.A., Wagner, R.F.: A general model for finite-sample effects in training and testing of competing classifiers. IEEE Transactions on Pattern Analysis and Machine Intelligence **25** (2003) 1561–1569
[66] Provost, F., Fawcett, T., Kohavi, R.: The case against accuracy estimation for comparing induction algorithms. In: Proceedings of the Fifteenth International Conference on Machine Learning. Morgan Kaufmann, San Francisco, CA (1998) 445–453
[67] Swets, J.A., Pickett, R.M.: Evaluation of diagnostic systems: Methods from signal detection theory. Academic Press, New York, NY (1982)
[68] Metz, C.E., Jiang, Y., MacMahon, H., Nishikawa, R.M., Pan, X.: ROC software. Web page, Kurt Rossmann Laboratories for Radiologic Image Research, University of Chicago, Chicago, IL (2003)
[69] Breiman, L.: Arcing classifiers. The Annals of Statistics **26** (1998) 801–849
[70] Dietterich, T.G.: Approximate statistical tests for comparing supervised classification learning algorithms. Neural Computation **10** (1998) 1895–1924
[71] Sahai, H., Ageel, M.I.: The analysis of variance: Fixed, random, and mixed models. Birkhäuser, Boston, MA (2000)
[72] Keppel, G., Saufley, W.H., Tokunaga, H.: Introduction to design and analysis. 2nd edn. W. H. Freeman, New York, NY (1992)
[73] Wolpert, D.H.: Stacked generalization. Neural Networks **5** (1992) 241–259

[74] Littlestone, N., Warmuth, M.K.: The weighted majority algorithm. Information and Computation **108** (1994) 212–261
[75] Breiman, L.: Bagging predictors. Machine Learning **24** (1996) 123–140
[76] Quinlan, J.R.: Bagging, boosting, and C4.5. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence. AAAI Press, Menlo Park, CA (1996) 725–730
[77] Maclin, R., Opitz, D.: An empirical evaluation of bagging and boosting. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence. AAAI Press, Menlo Park, CA (1997) 546–551
[78] Bauer, B., Kohavi, R.: An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. Machine Learning **36** (1999) 105–139
[79] Dietterich, T.G.: An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. Machine Learning **40** (2000) 139–158
[80] Schapire, R.E.: A brief introduction to boosting. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence. Morgan Kaufmann, San Francisco, CA (1999) 1401–1406
[81] Kolter, J.Z., Maloof, M.A.: Learning to detect malicious executables in the wild. In: Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM Press, New York, NY (2004) 470–478
[82] Freund, Y., Schapire, R.E.: Experiments with a new boosting algorithm. In: Proceedings of the Thirteenth International Conference on Machine Learning. Morgan Kaufmann, San Francisco, CA (1996) 148–156
[83] Dietterich, T.G.: Machine learning for sequential data: A review. In: Structural, syntactic, and statistical pattern recognition. Volume 2396 of Lecture Notes in Computer Science. Springer-Verlag, New York, NY (2002) 15–30
[84] Baum, L.E.: An inequality and associated maximization technique in statistical estimation for probabilistic functions of a Markov process. Inequalities **3** (1972) 1–8
[85] Lane, T., Brodley, C.E.: Temporal sequence learning and data reduction for anomaly detection. ACM Transactions on Information and System Security **2** (1999) 295–331
[86] Blake, C.L., Merz, C.J.: UCI repository of machine learning databases. Web site, Department of Information and Computer Sciences, University of California, Irvine (1998) `http://www.ics.uci.edu/~mlearn/MLRepository.html`
[87] Hettich, S., Bay, S.D.: The UCI KDD archive. Web site, Department of Information and Computer Sciences, University of California, Irvine (1999) `http://kdd.ics.uci.edu`
[88] Lippmann, R.P., Haines, J.W., Fried, D.J., Korba, J., Das, K.: The 1999 DARPA off-line intrusion detection evaluation. Computer Networks **34** (2000) 579–595

[89] Elkan, C.: Results of the KDD'99 classifier learning. SIGKDD Explorations: Newsletter of the ACM Special Interest Group on Knowledge Discovery and Data Mining **1** (2000) 63–64
[90] Mena, J.: Investigative data mining for security and criminal detection. Butterworth Heinemann, New York, NY (2003)
[91] Barbará, D., Jajodia, S., eds.: Applications of Data Mining in Computer Security. Volume 6 of Advances in Information Security. Kluwer Academic Publishers, Boston, MA (2002)
[92] Cristianini, N., Shawe-Taylor, J.: An introduction to support vector machines and other kernel-based learning methods. Cambridge University Press, Cambridge (2000)
[93] Schölkopf, B., Burges, C.J.C., Mika, S., eds.: Advances in Kernel Methods – Support Vector Learning. MIT Press, Cambridge, MA (1998)
[94] Bishop, C.M.: Neural networks for pattern recognition. Clarendon Press, Oxford (1995)
[95] McGraw, G., Morisett, G.: Attacking malicious code: A report to the Infosec Research Council. IEEE Software (2000) 33–41
[96] Anonymous: Maximum security. 4th edn. Sams Publishing, Indianapolis, IN (2003)
[97] Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: Proceedings of the Twelfth USENIX Security Symposium. The USENIX Association, Berkeley, CA (2003) 169–186
[98] Lo, R.W., Levitt, K.N., Olsson, R.A.: MCF: A malicious code filter. Computers & Security **14** (1995) 541–566
[99] Kephart, J.O., Sorkin, G.B., Arnold, W.C., Chess, D.M., Tesauro, G.J., White, S.R.: Biologically inspired defenses against computer viruses. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence. Morgan Kaufmann, San Francisco, CA (1995) 985–996
[100] Tesauro, G., Kephart, J.O., Sorkin, G.B.: Neural networks for computer virus recognition. IEEE Expert **11** (1996) 5–6
[101] Schultz, M.G., Eskin, E., Zadok, E., Stolfo, S.J.: Data mining methods for detection of new malicious executables. In: Proceedings of the IEEE Symposium on Security and Privacy. IEEE Press, Los Alamitos, CA (2001) 38–49
[102] Miller, P.: hexdump 1.4. Software, `http://gd.tuwien.ac.at/softeng/Aegis/hexdump.html` (1999)
[103] Soman, S., Krintz, C., Vigna, G.: Detecting malicious Java code using virtual machine auditing. In: Proceedings of the Twelfth USENIX Security Symposium. The USENIX Association, Berkeley, CA (2003) 153–168
[104] Durning-Lawrence, E.: Bacon is Shake-speare. The John McBride Company, New York, NY (1910)

[105] Kjell, B., Woods, W.A., Frieder, O.: Discrimination of authorship using visualization. Information Processing and Management **30** (1994) 141–150
[106] Spafford, E.H., Weeber, S.A.: Software forensics: Can we track code to its authors? Computers & Security **12** (1993) 585–595
[107] Gray, A.R., Sallis, P.J., MacDonell, S.G.: Software forensics: Extending authorship analysis techniques to computer programs. In: Proceedings of the Third Biannual Conference of the International Association of Forensic Linguists. International Association of Forensic Linguists, Birmingham, UK (1997) 1–8
[108] Krsul, I.: Authorship analysis: Identifying the author of a program. Master's thesis, Purdue University, West Lafayette, IN (1994)
[109] Aiken, A.: MOSS: A system for detecting software plagiarism. Software, Department of Computer Science, University of California, Berkeley, `http://www.cs.berkeley.edu/~aiken/moss.html` (1994)
[110] Jankowitz, H.T.: Detecting plagiarism in student Pascal programs. Computer Journal **31** (1988) 1–8
[111] Krsul, I., Spafford, E.: Authorship analysis: Identifying the authors of a program. In: Proceedings of the Eighteenth National Information Systems Security Conference. National Institute of Standards and Technology, Gaithersburg, MD (1995) 514–524
[112] Jain, A.K., Duin, R.P.W., Mao, J.: Statistical pattern recognition: A review. IEEE Transactions on Pattern Analysis and Machine Intelligence **22** (2000) 4–37
[113] Dumais, S., Platt, J., Heckerman, D., Sahami, M.: Inductive learning algorithms and representations for text categorization. In: Proceedings of the Seventh International Conference on Information and Knowledge Management. ACM Press, New York, NY (1998) 148–155
[114] Sahami, M., Dumais, S., Heckerman, D., Horvitz, E.: A Bayesian approach to filtering junk e-mail. In: Learning for Text Categorization: Papers from the 1998 AAAI Workshop. AAAI Press, Menlo Park, CA (1998) Technical Report WS-98-05
[115] Yang, Y., Pederson, J.O.: A comparative study on feature selection in text categorization. In: Proceedings of the Fourteenth International Conference on Machine Learning. Morgan Kaufmann, San Francisco, CA (1997) 412–420
[116] Maron, M.E., Kuhns, J.L.: On relevance, probabilistic indexing and information retrieval. Journal of the ACM **7** (1960) 216–244
[117] Joachims, T.: Text categorization with support vector machines: Learning with many relevant features. In: Proceedings of the Tenth European Conference on Machine Learning. Springer-Verlag, Berlin (1998) 487–494
[118] Platt, J.: Fast training of support vector machines using sequential minimal optimization. In Schölkopf, B., Burges, C.J.C., Mika, S., eds.:

Advances in Kernel Methods – Support Vector Learning. MIT Press, Cambridge, MA (1998)
[119] Platt, J.: Probabilities for SV machines. In Bartlett, P.J., Schölkopf, B., Schuurmans, D., Smola, A.J., eds.: Advances in Large-Margin Classifiers. MIT Press, Cambridge, MA (2000) 61–74
[120] Opitz, D., Maclin, R.: Popular ensemble methods: An empirical study. Journal of Artificial Intelligence Research **11** (1999) 169–198
[121] Drummond, C., Holte, R.C.: Explicitly representing expected cost: An alternative to ROC representation. In: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM Press, New York, NY (2000) 198–207
[122] Lee, W., Stolfo, S.: Data mining approaches for intrusion detection. In: Proceedings of the Seventh USENIX Security Symposium. The USENIX Association, Berkeley, CA (1998) 79–94
[123] Stolfo, S.J., Lee, W., Chan, P.K., Fan, W., Eskin, E.: Data mining-based intrusion detectors: An overview of the Columbia IDS project. ACM SIGMOD Record **30** (2001) 5–14
[124] Lee, W., Park, C.T., Stolfo, S.J.: Automated intrusion detection using NFR: methods and experiences. In: Proceedings of the First Workshop on Intrusion Detection and Network Monitoring. The USENIX Association, Berkeley, CA (1999)
[125] Manganaris, S., Christensen, M., Zerkle, D., Hermiz, K.: A data mining analysis of RTID alarms. Computer Networks **34** (2000) 571–577
[126] Halme, L.R., Bauer, K.R.: AINT misbehaving – A taxonomy of anti-intrusion techniques. In: Proceedings of the Eighteenth National Information Systems Security Conference. National Institute of Standards and Technology, Gaithersburg, MD (1995) 163–172
[127] Bloedorn, E., Christiansen, A.D., Hill, W., Skorupka, C., Talbot, L.M., Tivel, J.: Data mining for network intrusion detection: How to get started. In: AFCEA Federal Database Exposition and Colloquium. Armed Forces Communications and Electronics Association (2001) 31–39
[128] Skorupka, C., Tivel, J., Talbot, L., DeBarr, D., Bloedorn, E., Christiansen, A.: Surf the flood: Reducing high-volume intrusion detection data by automated record aggregation. In: Proceedings of the SANS 2001 Technical Conference. The SANS Institute, Bethesda, MD (2001)
[129] Axelsson, S.: Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden (2000)
[130] Collins, A., Paller, A., Lucas, A., Kotkov, A., Chuvakin, A., Bellamy, B., Peterson, B., Booth, C., Benjes, C., Misra, C., Dobrotka, D., Beecher, D., Fisher, E., Skoudis, E., Ray, E.W., Kamerling, E., Eschelbeck, G., Campione, J., Ito, J., Li, J., Thacker, K., Tan, K.Y., Bueno, P.P.F., Beck, P., Wanner, R., Lascola, R.M., Patel, R., Morrison, R., Lawler, S.A., Northcutt, S., Kletnieks, V., Eckroade, W.: The twenty most criti-

cal security vulnerabilities (updated): The experts consensus. Technical Report Version 4.0, The SANS Institute, http://www.sans.org/top20 (Oct. 8, 2003)
[131] Ramaswamy, S., Rastogi, R., Shim, K.: Efficient algorithms for mining outliers from large data sets. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. ACM Press, New York, NY (2000) 427–438
[132] Breunig, M., Kriegel, H.P., Ng, R., Sander, J.: LOF: Identifying density-based local outliers. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. ACM Press, New York, NY (2000) 93–104
[133] SPSS Clementine: http://www.spss.com/clementine (2004)
[134] Allen, J., Christie, A., Fithen, W., McHugh, J., Pickel, J., Stoner, E.: State of the practice of intrusion detection technologies. Technical Report CMU/SEI-99-TR-028, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2000)
[135] Bace, R.: Intrusion detection. Sams Publishing, Indianapolis, IN (1999)
[136] Debar, H., Dacier, M., Wespi, A.: A revised taxonomy for intrusion detection systems. Annales des Télécommunications **55** (2000) 361–378
[137] Broderick, J.: IBM outsourced solution (1998) http://www.infoworld.com/cgi-bin/displayTC.pl?/980504sb3-ibm.htm
[138] Axelsson, S.: The base-rate fallacy and the difficulty of intrusion detection. ACM Transactions on Information and System Security **3** (2000) 186–205
[139] Julisch, K.: Mining alarm clusters to improve alarm handling efficiency. In: Proceedings of the Seventeenth Annual Computer Security Applications Conference. IEEE Computer Society Press, Los Alamitos, CA (2001) 12–21
[140] Dain, O., Cunningham, R.K.: Fusing heterogeneous alert streams into scenarios. In Barbará, D., Jajodia, S., eds.: Applications of Data Mining in Computer Security. Volume 6 of Advances in Information Security. Kluwer Academic Publishers, Boston, MA (2002) 103–122
[141] Debar, H., Wespi, A.: Aggregation and correlation of intrusion-detection alerts. In: Recent Advances in Intrusion Detection. Volume 2212 of Lecture Notes in Computer Science. Springer-Verlag, New York, NY (2001) 85–103. Fourth International Symposium, RAID 2001, Davis, CA, October 10–12, 2001. Proceedings
[142] Valdes, A., Skinner, K.: Probabilistic alert correlation. In: Recent Advances in Intrusion Detection. Volume 2212 of Lecture Notes in Computer Science. Springer-Verlag, New York, NY (2001) 54–68. Fourth International Symposium, RAID 2001, Davis, CA, October 10–12, 2001. Proceedings
[143] Julisch, K., Dacier, M.: Mining intrusion detection alarms for actionable knowledge. In: Proceedings of the Eighth ACM SIGKDD International

Conference on Knowledge Discovery and Data Mining. ACM Press, New York, NY (2002) 366–375
[144] Julisch, K.: Clustering intrusion detection alarms to support root cause analysis. ACM Transactions on Information and System Security **6** (2003) 443–471
[145] Julisch, K.: Using root cause analysis to handle intrusion detection Alarms. PhD thesis, University of Dortmund, Germany (2003)
[146] Julisch, K.: Data mining for intrusion detection: A critical review. In Barbará, D., Jajodia, S., eds.: Applications of Data Mining in Computer Security. Volume 6 of Advances in Information Security. Kluwer Academic Publishers, Boston, MA (2002) 33–62
[147] Laprie, J.C., ed.: Dependability: Basic Concepts and Terminology. Volume 5 of Dependable Computing and Fault-Tolerant Systems. Springer-Verlag, New York, NY (1992)
[148] Powell, D., Stroud, R.: Architecture and revised model of MAFTIA. Technical Report CS-TR-749, University of Newcastle upon Tyne, United Kingdom (2001)
[149] CERT: Advisory CA-2001-19: "Code Red" worm exploiting buffer overflow in IIS Indexing Service DLL (2001) `http://www.cert.org/advisories/CA-2001-19.html`
[150] Ohsie, D.A.: Modeled abductive inference for event management and correlation. PhD thesis, Columbia University, New York, NY (1998)
[151] Peng, Y., Reggia, J.A.: A probabilistic causal model for diagnostic problem solving – Part II: Diagnostic strategy. IEEE Transactions on Systems, Man, and Cybernetics **17** (1987) 395–406
[152] Peng, Y., Reggia, J.A.: A probabilistic causal model for diagnostic problem solving – Part I: Integrating symbolic clausal inference with numeric probabilistic inference. IEEE Transactions on Systems, Man, and Cybernetics **17** (1987) 146–162
[153] Han, J., Cai, Y., Cercone, N.: Knowledge discovery in databases: An attribute-oriented approach. In: Proceedings of the Eighteenth ACM International Conference on Very Large Databases. ACM Press, New York, NY (1992) 547–559
[154] Han, J., Cai, Y., Cercone, N.: Data-driven discovery of quantitative rules in relational databases. IEEE Transactions on Knowledge and Data Engineering **5** (1993) 29–40
[155] Jain, A.K., Dubes, R.C.: Algorithms for clustering data. Prentice-Hall, Englewood Cliffs, NJ (1988)
[156] Bugtraq ID 1846, CVE-2000-0945: Cisco Catalyst 3500 XL remote arbitrary command execution vulnerability (2000) `http://www.securityfocus.com/bid/1846`
[157] Aldenderfer, M.S., Blashfield, R.K.: Cluster analysis. Volume 44 of Quantitative Applications in the Social Sciences. Sage Publications, London (1985)

[158] Anderberg, M.R.: Cluster analysis for applications. Academic Press, New York, NY (1973)
[159] Gordon, A.D.: Classification. 2nd edn. Volume 82 of Monographs on Statistics and Applied Probability. CRC Press, Boca Raton, FL (1999)
[160] Guha, S., Rastogi, R., Shim, K.: CURE: an efficient clustering algorithm for large databases. In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data. (1998) 73–84
[161] Dubes, R.C., Jain, A.K.: Validity studies in clustering methodologies. Pattern Recognition **11** (1979) 235–254
[162] Dubes, R.C.: Cluster analysis and related issues. In Chen, C.H., Pau, L.F., Wang, P.S.P., eds.: Handbook of Pattern Recognition and Computer Vision. 2nd edn. World Scientific, Hackensack, NJ (1998) 3–32
[163] Gordon, A.D.: Cluster validation. In Hayashi, C., Ohsumi, N., Yajima, K., Tanaka, Y., Bock, H.H., Baba, Y., eds.: Data Science, Classification, and Related Methods. Springer-Verlag, New York, NY (1998) 22–39
[164] Dubes, R.C., Jain, A.K.: Clustering methodologies in exploratory data analysis. In Yovits, M.C., ed.: Advances in Computers. Volume 19. Academic Press, New York, NY (1980) 113–228
[165] Halkidi, M., Batistakis, Y., Vazirgiannis, M.: On clustering validation techniques. Journal of Intelligent Information Systems **17** (2001) 107–145
[166] Breckenridge, J.N.: Replication cluster analysis: Method, consistency, and validity. Multivariate Behavioral Research **24** (1989) 147–161
[167] McIntyre, R.M., Blashfield, R.K.: A nearest-centroid technique for evaluating the minimum-variance clustering procedure. Multivariate Behavioral Research **15** (1980) 225–238
[168] Morey, L.C., Blashfield, R.K., Skinner, H.A.: A comparison of cluster analysis techniques within a sequential validation framework. Multivariate Behavioral Research **18** (1983) 309–329
[169] Hjorth, J.S.U.: Computer intensive statistical methods: Validation, model selection, and bootstrap. CRC Press, Boca Raton, FL (1993)
[170] Bock, H.H.: Probability models and hypotheses testing in partitioning cluster analysis. In Arabie, P., Hubert, L.J., De Soete, G., eds.: Clustering and Classification. World Scientific, Hackensack, NJ (1996) 377–453
[171] Heberlein, L.T., Dias, G.V., Levitt, K.N., Mukherjee, B., Wood, J., Wolber, D.: A network security monitor. In: Proceedings of the IEEE Symposium on Research in Security and Privacy. IEEE Press, Los Alamitos, CA (1990) 296–304
[172] Roesch, M., Green, C.: Snort – The open source network intrusion detection system. `http://www.snort.org/` (2003)
[173] NIST: Ping of Death – DoS attack. `http://icat.nist.gov/icat.cfm?cvename=CVE-1999-0128` (1999)
[174] CERT: CERT Advisory CA-1996-21 TCP SYN flooding and IP spoofing attacks (1996)

[175] NIST: Teardrop – DoS attack. http://icat.nist.gov/icat.cfm?cvename=CAN-1999-0015 (1999)
[176] Ptacek, T.H., Newsham, T.N.: Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical Report T2R-0Y6, Secure Networks, Inc., Suite 330, 1201 5th Street SW, Calgary, Alberta, Canada (1998)
[177] Postel, J.: RFC 793: Transmission Control Protocol (1981)
[178] Postel, J.: RFC 768: User Datagram Protocol (1980)
[179] Postel, J.: RFC 791: Internet Protocol (1981)
[180] Mahoney, M.V.: Network traffic anomaly detection based on packet bytes. In: Proceedings of the 2003 ACM Symposium on Applied Computing. ACM Press, New York, NY (2003) 346–350
[181] Lee, W., Stolfo, S.J., Mok, K.W.: A data mining framework for building intrusion detection models. In: Proceedings of the 1999 IEEE Symposium on Security and Privacy. IEEE Press, Los Alamitos, CA (1999) 120–132
[182] NIST: Land IP DoS attack. http://icat.nist.gov/icat.cfm?cvename=CVE-1999-0016 (1999)
[183] MIT Lincoln Labs: 1999 DARPA intrusion detection evaluation data set. http://www.ll.mit.edu/IST/ideval/data/1999/1999_data_index.html (1999)
[184] Webb, G.I.: Magnum Opus rule mining tools. http://www.rulequest.com/MagnumOpus-info.html (2004)
[185] Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proceedings of the Eleventh International Conference on Data Engineering. IEEE Press, Los Alamitos, CA (1995) 3–14
[186] Barbará, D., Couto, J., Jajodia, S., Wu, N.: ADAM: A testbed for exploring the use of data mining in intrusion detection. ACM SIGMOD Record **30** (2001) 15–24
[187] Lee, W., Stolfo, S.: A framework for constructing features and models for intrusion detection systems. ACM Transactions on Information and System Security **3** (2000) 227–261
[188] Reynolds, J., Postel, J.: RFC 1700: Assigned numbers (1994)
[189] Porras, P.A., Neumann, P.G.: EMERALD: Event monitoring enabling responses to anomalous live disturbances. In: Proceedings of the Twentieth NIST-NCSC National Information Systems Security Conference. National Institute of Standards and Technology, Gaithersburg, MD (1997) 353–365
[190] iNetPrivacy Software Inc.: Antifirewall. http://www.antifirewall.com/intro.htm (2003)
[191] Markatos, E.P.: Tracing a large-scale peer to peer system : An hour in the life of Gnutella. In: Proceedings of the Second IEEE International Symposium on Cluster Computing and the Grid. IEEE Press, Los Alamitos, CA (2002) 56–65

[192] Hansell, S.: E-mail's backdoor open to spammers. New York Times (May 20, 2003)
[193] Taylor, C., Alves-Foss, J.: NATE: Network analysis of anomalous traffic events, a low-cost approach. In: Proceedings of the 2001 Workshop on New Security Paradigms. ACM Press, New York, NY (2001) 89–96
[194] Mahoney, M.V., Chan, P.K.: PHAD: Packet header anomaly detection for identifying hostile network traffic. Technical Report CS-2001-4, Department of Computer Sciences, Florida Institute of Technology, Melbourne, FL (2001)
[195] Sharman Networks Ltd. : Kazaa Media. `http://www.kazaa.com/us/` (2003)
[196] St. Sauver, J.: Percentage of total Internet2 traffic consisting of Kazaa/Morpheus/FastTrack – University of Oregon. In: Collaborative Computing in Higher Education: Peer-to-Peer and Beyond Workshop. (2002)
[197] Cornell University Student Assembly Committee on Information and Technologies and ResNet: Cornell Internet usage statistics. `http://www.cit.cornell.edu/computer/students/bandwidth/charts.html` (2001)
[198] Anderson, J.P.: Computer security threat monitoring and surveillance. Technical Report 79F296400, James P. Anderson Co., Fort Washington, PA (1980)
[199] Denning, D.E.: An intrusion-detection model. IEEE Transactions on Software Engineering **SE-13** (1987) 222–232
[200] MIT Lincoln Labs: 1998 DARPA intrusion detection evaluation data set. `http://www.ll.mit.edu/IST/ideval/data/1998/1998_data_index.html` (1998)
[201] Cabrera, J.B.D., Ravichandran, B., Mehra, R.K.: Statistical traffic modeling for network intrusion detection. In: Proceedings of the Eighth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. IEEE Computer Society Press, Los Alamitos, CA (2000) 466–473
[202] Lee, W.: A data mining framework for constructing features and models for intrusion detection systems. PhD thesis, Columbia University (1999)
[203] Jacobson, V., Leres, C., McCanne, S.: `tcpdump`. Available via anonymous ftp from `ftp.ee.lbl.gov` (1989)
[204] Domingos, P.: MetaCost: A general method for making classifiers cost-sensitive. In: Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM Press, New York, NY (1999) 155–164
[205] Lee, W., Fan, W., Miller, M., Stolfo, S.J., Zadok, E.: Toward cost-sensitive modeling for intrusion detection and response. Journal of Computer Security **10** (2002) 5–22
[206] Network Flight Recorder Inc.: Network flight recorder. `http://www.nfr.com` (1997)

[207] Lippmann, R.P., Fried, D.J., Graf, I., Haines, J., Kendall, K.R., McClung, D., Weber, D., Webster, S., Wyschogrod, D., Cunningham, R.K., Zissman, M.A.: Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. In: Proceedings of the DARPA Information Survivability Conference and Exposition. Volume 2. IEEE Press, Los Alamitos, CA (2000) 12–26

[208] Miller, M.: Learning cost-sensitive classification rules for network intrusion detection using RIPPER. Technical Report CUCS-035-99, Computer Science Department, Columbia University, New York, NY (1999)

[209] Turney, P.: Cost-sensitive learning bibliography. Web site, `http://purl.org/peter.turney/bibliographies/cost-sensitive.html` (2001)

[210] Chan, P., Stolfo, S.: Towards scalable learning with non-uniform class and cost distribution: A case study in credit card fraud detection. In: Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining. AAAI Press, Menlo Park, CA (1998) 164–168

[211] Fan, W., Stolfo, S., Zhang, J.X., Chan, P.K.: AdaCost: Misclassification cost-sensitive learning. In: Proceedings of the Sixteenth International Conference on Machine Learning. Morgan Kaufmann, San Francisco, CA (1999) 97–105

[212] Lavrac, N., Gamberger, D., Turney, P.: Cost-sensitive feature reduction applied to a hybrid genetic algorithm. In: Algorithmic Learning Theory. Volume 1160 of Lecture Notes in Artificial Intelligence. Springer-Verlag, New York, NY (1996) 127–134. Seventh International Workshop, ALT'96, Sydney, Australia, October 23–25, 1996. Proceedings

[213] Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proceedings of the Ninth ACM Conference on Computer and Communications Security. ACM Press, New York, NY (2002) 255–264

[214] Tan, K., Maxion, R.: "Why 6?" Defining the operational limits of stide, an anomaly-based intrusion detector. In: Proceedings of the IEEE Symposium on Security and Privacy. IEEE Press, Los Alamitos, CA (2002) 188–201

[215] Forrest, S., Hofmeyr, S., Somayaji, A., Longstaff, T.: A sense of self for Unix processes. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy. IEEE Press, Los Alamitos, CA (1996) 120–128

[216] Lane, T., Brodley, C.E.: Detecting the abnormal: Machine learning in computer security. Technical Report ECE 97-1, Department of Electrical and Computer Engineering, Purdue University, West Lafayette, IN (1997)

[217] Lane, T., Brodley, C.E.: Sequence matching and learning in anomaly detection for computer security. In: AI Approaches to Fraud Detection and Risk Management: Papers from the 1997 AAAI Workshop. AAAI Press, Menlo Park, CA (1997) 43–49. Technical Report WS-97-07

[218] Warrender, C., Forrest, S., Pearlmutter, B.: Detecting intrusions using system calls: Alternative data models. In: Proceedings of the 1999 IEEE Symposium on Security and Privacy. IEEE Press, Los Alamitos, CA (1999) 133–145

[219] Wespi, A., Dacier, M., Debar, H.: An intrusion-detection system based on the TEIRESIAS pattern-discovery algorithm. In: Proceedings of the European Institute for Computer Anti-Virus Research Conference. (1999)

[220] Wespi, A., Dacier, M., Debar, H.: Intrusion detection using variable-length audit trail patterns. In: Recent Advances in Intrusion Detection. Volume 1907 of Lecture Notes in Computer Science. Springer-Verlag, New York, NY (2000) 110–129. Third International Workshop, RAID 2000, Toulouse, France, October 2–4, 2000. Proceedings

[221] Rigoutsos, I., Floratos, A.: Combinatorial pattern discovery in biological sequences: The TEIRESIAS algorithm. Bioinformatics **14** (1998) 55–67

[222] Ghosh, A., Schwartzbard, A.: A study in using neural networks for anomaly and misuse detection. In: Proceedings of the Eighth USENIX Security Symposium. The USENIX Association, Berkeley, CA (1999) 141–152

[223] Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy. IEEE Press, Los Alamitos, CA (2001) 144–155

[224] Jiang, N., Hua, K., Sheu, S.: Considering both intra-pattern and inter-pattern anomalies in intrusion detection. In: Proceedings of the 2002 IEEE International Conference on Data Mining. IEEE Press, Los Alamitos, CA (2002) 637–640

[225] Liao, Y., Vemuri, R.: Use of text categorization techniques for intrusion detection. In: Proceedings of the Eleventh USENIX Security Symposium. The USENIX Association, Berkeley, CA (2002) 51–59

[226] Jones, A., Li, S.: Temporal signatures for intrusion detection. In: Proceedings of the Seventeenth Annual Computer Security Applications Conference. IEEE Press, Los Alamitos, CA (2001) 252–261

[227] Coull, S., Branch, J., Szymanski, B., Breimer, E.: Intrusion detection: A bioinformatics approach. In: Proceedings of the Ninteenth Annual Computer Security Applications Conference. IEEE Press, Los Alamitos, CA (2003) 24–33

[228] Mazeroff, G., De Cerqueira, V., Gregor, J., Thomason, M.: Probabilistic trees and automata for application behavior modeling. In: Proceedings of the Forty-first ACM Southeast Regional Conference. ACM Press, New York, NY (2003)

[229] Lee, W., Stolfo, S., Chan, P.: Learning patterns from UNIX process execution traces for intrusion detection. In: AI Approaches to Fraud Detection and Risk Management: Papers from the 1997 AAAI Work-

shop. AAAI Press, Menlo Park, CA (1997) 50–56. Technical Report WS-97-07
[230] Portnoy, L.: Intrusion detection with unlabeled data using clustering. Undergraduate thesis, Department of Computer Science, Columbia University, New York, NY (2000) http://www1.cs.columbia.edu/ids/publications/cluster-thesis00.pdf
[231] Eskin, E., Arnold, A., Prerau, M., Portnoy, L., Stolfo, S.: A geometric framework for unsupervised anomaly detection: Detecting intrusions in unlabeled data. In Barbará, D., Jajodia, S., eds.: Applications of Data Mining in Computer Security. Volume 6 of Advances in Information Security. Kluwer Academic Publishers, Boston, MA (2002) 77–101
[232] Chan, P.K., Mahoney, M.V., Arshad, M.: Learning rules and clusters for anomaly detection in network traffic. In Kumar, V., Srivastava, J., Lazarevic, A., eds.: Managing Cyber Threats: Issues, Approaches and Challenges. Kluwer Academic Publishers, Boston, MA (2003)
[233] Lazarevic, A., Ertöz, L., Ozgur, A., Srivastava, J., Kumar, V.: A comparative study of anomaly detection schemes in network intrusion detection. In: Proceedings of the Third SIAM International Conference on Data Mining. Society of Industrial and Applied Mathematics, Philadelphia, PA (2003)
[234] Bernaschi, M., Gabrielli, E., Mancini, L.V.: Operating system enhancement to prevent the misuse of system calls. In: Proceedings of the Seventh ACM Conference on Computer and Communications Security. ACM Press, New York, NY (2000) 174–183
[235] Gibbs, A.J., McIntyre, G.A.: The diagram, a method for comparing sequences. Its use with amino acid and nucleotide sequences. European Journal of Biochemistry **16** (1970) 1–11
[236] Mahalanobis, P.C.: On tests and measures of groups divergence. International Journal of the Asiatic Society of Bengal **26** (1930)
[237] Knorr, E., Ng, R.: Algorithms for mining distance-based outliers in large data sets. In: Proceedings of the Twenty-fourth International Conference on Very Large Databases. Morgan Kaufmann, San Francisco, CA (1998) 392–403
[238] Aggarwal, C., Yu, P.: Outlier detection for high dimensional data. In: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data. ACM Press, New York, NY (2001) 37–46
[239] Salvador, S., Chan, P., Brodie, J.: Learning states and rules for time series anomaly detection. In: Proceedings of the Seventeenth International Florida AI Research Society Conference. AAAI Press, Menlo Park, CA (2004) 306–311
[240] Mahoney, M.V., Chan, P.K.: Learning rules for anomaly detection of hostile network traffic. In: Proceedings of the Third IEEE International Conference on Data Mining. IEEE Press, Los Alamitos, CA (2003) 601–604

[241] Witten, I.H., Bell, T.C.: The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. IEEE Transactions on Information Theory **37** (1991) 1085–1094
[242] Whittaker, J.A., De Vivanco, A.: Neutralizing Windows-based malicious mobile code. In: Proceedings of the 2002 ACM Symposium on Applied Computing. ACM Press, New York, NY (2002) 242–246
[243] Weiner, P.: Linear pattern matching algorithms. In: Proceedings of the Fourteenth Annual IEEE Symposium on Switching and Automata Theory. IEEE Press, Los Alamitos, CA (1973)
[244] McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of the ACM **23** (1976) 262–272
[245] Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14** (1995) 249–260
[246] Gusfield, D.: Algorithms on strings, trees, and sequences: Computer science and computational biology. Cambridge University Press, Cambridge (1997)
[247] Lane, T.: Machine learning techniques for the computer security domain of anomaly detection. PhD thesis, Purdue University, West Lafayette, IN (2000)
[248] Lane, T., Brodley, C.E.: An empirical study of two approaches to sequence learning for anomaly detection. Machine Learning **51** (2003) 73–107
[249] Fawcett, T., Provost, F.: Activity monitoring: Noticing interesting changes in behavior. In: Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM Press, New York, NY (1999)
[250] Lunt, T.F., Jagannathan, R.: A prototype real-time intrusion-detection expert system. In: Proceedings of the IEEE Symposium on Security and Privacy, New York, NY, IEEE Press (1988) 59–66
[251] Lunt, T.F.: IDES: An intelligent system for detecting intruders. In: Proceedings of the Symposium: Computer Security, Threat and Countermeasures, Rome, Italy (1990)
[252] Anderson, D., Lunt, T., Javitz, H., Tamaru, A., Valdes, A.: Safeguard final report: Detecting unusual program behavior using the NIDES statistical component. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA (1993)
[253] Mukherjee, B., Heberlein, L.T., Levitt, K.N.: Network intrusion detection. IEEE Network **8** (1994) 26–41
[254] Davison, B.D., Hirsh, H.: Predicting sequences of user actions. In: Predicting the Future: AI Approaches to Time-series Problems: Papers from the AAAI Workshop. AAAI Press, Menlo Park, CA (1998) 5–12. Technical Report WS-98-07
[255] Lee, W., Stolfo, S.J., Mok, K.W.: Mining audit data to build intrusion detection models. In: Proceedings of the Fourth International Confer-

ence on Knowledge Discovery and Data Mining. AAAI Press, Menlo Park, CA (1998) 66–72
[256] Mahoney, M.V., Chan, P.: Learning rules for anomaly detection of hostile network traffic. In: Proceedings of the Third IEEE International Conference on Data Mining, Los Alamitos, CA (2003) 601–604
[257] Forrest, S., Perelson, A.S., Allen, L., Cherukuri, R.: Self-nonself discrimination in a computer. In: Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy, Los Alamitos, CA, IEEE Computer Society Press (1994)
[258] Balthrop, J., Esponda, F., Forrest, S., Glickman, M.: Coverage and generalization in an artificial immune system. In: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference. Morgan Kaufmann, San Francisco, CA (2002) 3–10
[259] Dasgupta, D., González, F.: An immunity-based technique to characterize intrusions in computer networks. IEEE Transactions on Evolutionary Computation **6** (2002) 1081–1088
[260] Kim, J., Bentley, P.J.: Towards an artificial immune system for network intrusion detection: An investigation of clonal selection with a negative selection operator. In: Proceedings of the 2002 Congress on Evolutionary Computation. IEEE Press, Los Alamitos, CA (2001) 1244–1252
[261] Haines, J.W., Lippmann, R.P., Fried, D.J., Tran, E., Boswell, S., Zissman, M.A.: 1999 DARPA intrusion detection system evaluation: Design and procedures. Technical Report 1062, MIT Lincoln Labs, Lexington, MA (2001)
[262] Schonlau, M., DuMouchel, W., Ju, W.H., Karr, A.F., Theus, M., Vardi, Y.: Computer intrusion: Detecting masquerades. Statistical Science **16** (2001) 58–74
[263] DuMouchel, W.: Computer intrusion detection based on Bayes factors for comparing command transition probabilities. Technical Report 91, National Institute of Statistical Sciences, Research Triangle Park, NC (1999)
[264] Schonlau, M., Theus, M.: Detecting masquerades in intrusion detection based on unpopular commands. Information Processing Letters **76** (2000) 33–38
[265] Ju, W.H., Vardi, Y.: A hybrid high-order Markov chain model for computer intrusion detection. Technical Report 92, National Institute of Statistical Sciences, Research Triangle Park, NC (1999)
[266] Lane, T., Brodley, C.E.: Approaches to online learning and concept drift for user identification in computer security. In: Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining. AAAI Press, Menlo Park, CA (1998) 259–263
[267] Smyth, P., Heckerman, D., Jordan, M.: Probabilistic independence networks for hidden Markov models. Neural Computation **9** (1997) 227–269
[268] Jensen, F.V.: Bayesian networks and decision graphs. Springer-Verlag, New York, NY (2001)

[269] Murphy, K.: Dynamic Bayesian networks: Representation, inference and learning. PhD thesis, University of California, Berkeley (2002)
[270] Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the EM algorithm. Journal of the Royal Statistical Society Series B (Methodological) **39** (1977) 1–38
[271] Moon, T.K.: The expectation-maximization algorithm. IEEE Signal Processing Magazine **13** (1996) 47–60
[272] Puterman, M.L.: Markov decision processes: Discrete stochastic dynamic programming. John Wiley & Sons, New York, NY (1994)
[273] Pearl, J.: Probabilistic reasoning in intelligent systems: Networks of plausible inference. Morgan Kaufmann, San Francisco, CA (1988)
[274] Charniak, E.: Bayesian networks without tears. AI Magazine **12** (1991) 50–63
[275] Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artificial Intelligence **101** (1998) 99–134
[276] Lusena, C., Mundhenk, M., Goldsmith, J.: Nonapproximability results for partially observable Markov decision processes. Journal of Artificial Intelligence Research **14** (2001) 83–103
[277] Theocharous, G.: Hierarchical learning and planning in partially observable Markov decision processes. PhD thesis, Michigan State University, East Lansing (2002)
[278] Huang, C., Darwiche, A.: Inference in belief networks: A procedural guide. International Journal of Approximate Reasoning **15** (1996) 225–263

Index