Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Unsupervised Learning Algorithms for Intrusion Detection

Tesi di dottorato di:
Stefano Zanero

Relatore:
Prof. Giuseppe Serazzi
Tutore:
Prof. Marco Colombetti
Coordinatore del programma di dottorato:
Prof. Stefano Crespi Reghizzi

XVIII Ciclo

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci 32 I 20133 — Milano

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Unsupervised Learning Algorithms for Intrusion Detection

Doctoral Dissertation of:
Stefano Zanero

Advisor:
Prof. Giuseppe Serazzi
Tutor:
Prof. Marco Colombetti
Supervisor of the Doctoral Program:
Prof. Stefano Crespi Reghizzi

XVIII Edition

To my mother, who has dedicated her life to me.
To all of my friends, who are sharing it with me.
And in loving memory of my father
your blessing has been with me for the past 17 years.

Acknowledgments

This work has begun to shape into its current form as my master's graduation thesis, and then slowly evolved to become the focus of my research years as a doctoral candidate at the Politecnico di Milano university. Thus, it is really difficult to thank all of the people who directly or indirectly contributed to the development of this work.

A first acknowledgment is for my advisor, professor Giuseppe Serazzi, who has supported me during this three years without restraining me in any way, letting me explore what I really wanted most. Professor Sergio M. Savaresi, my former advisor during my master's thesis development, has also played an important role in making me understand what research is, and why it can be loved. Both constantly gave me support, ideas, and also corrections, whenever corrections were needed.

A huge number of people contributed ideas and suggestions, or critiques, to my work on unsupervised learning techniques for intrusion detection. It is impossible to thank each one of them, but some deserve a special acknowledgment:

- My colleague Giuliano Casale, who has been a tireless target for my questions and doubts, and who has peer reviewed most of my work, sometimes being duly harsh with me.

- My colleague Davide Balzarotti, with whom I discussed intrusion detection and security issues in a lot of informal sessions over a number of cups of coffee.

- My tireless students Matteo F. Zazzetta, Federico Maggi and Davide Veneziano, for their invaluable support in software development and lab testing, and for having found all the real-world bugs in my nice theories.

- William Robertson, a Ph.D. student at UCSB, for helping us out greatly with LibAnomaly and SyscallAnomaly.

- Dr. Matteo Matteucci for his help in understanding various critical points related to learning and neural algorithms.

- Prof. Salvatore J. Stolfo, for comments and critiques on an early paper on this work.

- Dr. Marcus J. Ranum and Dr. Tina Bird of the "Log Analysis" mailing list and workgroup, along with other participants, for helping me in understanding the real world needs for intrusion detection systems.

- Mr. Jeff Moss of Black Hat, Inc. and Mr. Dragos Ruiu, organizer of the CanSecWest symposium, for having invited me to speak of my work in two of the most important and recognized forums for applied security research; I also need to thank all the speakers and the attendees that during such events contributed suggestions, ideas, and corrections.

My tutor, professor Marco Colombetti, was indeed the first to help me study Intrusion Detection Systems under a behavioral point of view, which will be recalled also in this thesis. For this portion of the work, I need to thank Dr. Marzia Possenti, M.D., and professor George W. Barlow of the Department of Integrative Biology at U.C. Berkeley for their support.

Most of this work was supported by the Italian FIRB Project "Performance evaluation for complex systems", and I need to warmly thank professor M. C. Calzarossa, the principal investigator of the project, for this. Without her support, and the support of the project, most of what is described here would have never been done.

A warm "thank you" goes also to professor Giovanni Vigna, of U.C. Santa Barbara, for accepting to review this thesis work, and also for his earlier suggestions and support. He has been supportive, and at the same time constructively criticized many of my assumptions and outright mistakes. Claudio Criscione also helped me in reviewing this work. Part of the images contained in this thesis were carefully redesigned on my early sketches by Federico Maggi, improving them a lot. Thanks!

I need to thank also the friends, students and colleagues who supported me throughout this work, by handling work issues or rearranging schedules and meetings to help me going forth in my research: Alvise Biffi, Luca Carettoni, Claudio Merloni. A special thank you, for the same reason, goes to prof. Andrea Monti of the University of Chieti.

"Tower of Hanoi", the CTF (Capture the Flag) student team of the Politecnico di Milano university, also deserves a mention. Being your coach through the three editions of the International CTF contest has been a fun and challenging experience. And a very rewarding one, too.

To all the others (you know who you are) who have contributed to this work an idea, a suggestion, who have listened to me, or who simply still bear with me despite my flawed character, to all of you, thank you. This work, and my life, would not be the same without you all.

Abstract

This work summarizes our research on the topic of the application of unsupervised learning algorithms to the problem of intrusion detection.

We introduce briefly the key concepts and problems of information security, and we characterize the main types of attacks against networked computer systems. This analysis brings us naturally to the problem of tamper evidence in computer systems, and to the definition of intrusion detection.

We analyze the different technologies and types of intrusion detection systems, the problems and open issues to be solved, and the state of the art of the research in the field, focusing on earlier studies on the application of unsupervised learning algorithms to intrusion detection.

We introduce then our research results, both in network and host based intrusion detection. We propose a novel, two tier architecture for network intrusion detection, capable of clustering packet payloads and correlate anomalies in the packet stream. We show the experiments we conducted on such architecture and give performance results and compare our achievements with other comparable existing systems.

We also propose a framework for detecting anomalous system calls in an operating system, capable of tracing anomalies both in the parameters of system calls and in their sequence, through the use of statistical models, clustering and a Markov Chain model used for time correlation. We show the results such system obtains and compare them with earlier studies.

Sommario

Questo lavoro di tesi riassume i risultati del nostro lavoro di ricerca nel campo dell'applicazione di algoritmi di apprendimento non supervisionato per la creazione di strumenti di intrusion detection (individuazione delle intrusioni).

Vengono innanzitutto presentati alcuni concetti e problemi chiave nel campo della sicurezza dell'informazione, e vengono caratterizzati, in breve, i tipi principali di attacchi contro i sistemi informatici in rete. Da questa analisi deriva, in modo naturale, il problema dell'evidenza della compromissione di sistemi informatici, e il problema dell'individuazione delle intrusioni.

Vengono poi analizzate le diverse tecnologie e tipologie di sistemi di intrusion detection esistenti, le relative problematiche e gli argomenti di ricerca, e lo stato dell'arte del campo, focalizzandosi sugli studi precedenti riguardanti l'applicazione di algoritmi non supervisionati di apprendimento.

Vengono di seguito presentati i risultati della nostra ricerca, sia nel campo dei sistemi network based che nel campo dei sistemi host based. Proponiamo un'architettura innovativa a due livelli per la creazione di un network intrusion detection system, capace di applicare tecniche di clustering al payload dei pacchetti e di tracciare le anomalie nel flusso. Forniamo i risultati degli esperimenti da noi condotti su tale architettura, e ne paragoniamo le prestazioni ad altri sistemi dello stesso tipo esistenti in letteratura.

Proponiamo anche un framework per identificare chiamate di sistema anomale, capace di tracciare le anomalie sia nel contenuto dei parametri delle chiamate, sia nella loro sequenza, tramite l'uso di modelli statistici, di tecniche di clustering e di un modello a catena di Markov usato per la correlazione temporale. Descriviamo i risultati che questo sistema ottiene e li paragoniamo a quelli ottenuti in studi precedenti.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivations: the need for Information Security

In the modern world, broader and broader parts of our lives are based upon, managed by, or transmitted with networked computer systems. While this is bringing wonderful insights and unforeseen advancements to computer science, there is also a widespread and legitimate concern about the security of such systems, which has been heightened by the tragic events of 9/11/2001, and by the ensuing international developments. Even if computer technologies were not directly involved in any attack until today, there is a widespread consensus between military and intelligence analysts that digital warfare and cyberterrorism will have an increasing role in the future.

Information is today the most important business asset [1], along with the processes, systems, and networks that store, manage and retrieve it. Thus, achieving an appropriate level of "information security" can be viewed as essential in order to maintain competitive edge (a "business enabler" technology), besides compliance with legal requirements and corporate image issues.

Organizations and their information systems and networks are faced with security threats from a wide range of sources, including computer-assisted fraud, espionage, sabotage, vandalism, as well as acts of God. Attack techniques have become more common, more ambitious, and increasingly sophisticated. Information security is important to businesses, public organizations, and to protect critical infrastructures at a national or global level. The interconnection of public and private networks and the sharing of information resources increased the difficulty of achieving access control. The trend to distributed computing has also weakened the effectiveness of central, specialist control. Most information systems have not been designed from the ground up to be secure. Security cannot be achieved through technical means alone, as it is an inherently human and social problem, and therefore needs to be supported by appropriate management and procedures.

1.2 Our research focus: Unsupervised Learning for Intrusion Detection

Our research work focused on the analysis and development of technologies based on *unsupervised learning algorithm* for the class of problems known as *intrusion detection*. Intrusion detection systems are components designed for making an information system tamper-evident, i.e. to detect behaviors which violate the system's security policy (extensively denoted as "intrusions", including into this broad category also insider abuse or privilege escalation).

An Intrusion Detection System is the computer system equivalent of a burglar alarm. The concept was introduced in 1980 by J.P. Anderson [2], and has subsequently been the focus of a wide area of research.

Two major classes of Intrusion Detection systems exist, based on a different approach to the problem: *anomaly detection* systems, which try to create a model of normal behavior, and flags as suspicious any deviation; or *misuse detection* systems, which use a knowledge base to recognize directly the *signatures* of intrusion attempts.

These systems have symmetric strengths and weaknesses. Anomaly detection systems don't require "a priori" knowledge of the attacks, being theoretically able to detect any type of misbehavior in a statistical way. On the other hand, they proved difficult to build, they usually need a long training phase on the system, and are also traditionally known to be prone to errors and false positives.

Misuse based systems, vice versa, require an extensive study of attacks in order to build and keep up to date the knowledge base: this also implies they are powerless against new, "zero-day" attacks [3], and subject to a wide array of evasion techniques [4]. They are slightly more resistant against the false positive problem (but misconfiguration and a huge number of unwanted alerts can destroy this advantage), and are also much simpler to conceive and build.

Most commercial intrusion detection systems are substantially misuse based. Anomaly based systems have been mostly developed in academic environments, and mostly in an *host based* fashion, while today *network based* IDSs are dominant. However, the continuous evolution of the types of attacks against computer networks suggests that we need a paradigmatic shift from misuse based intrusion detection system to anomaly based ones.

While a number of earlier attempts in this direction have been quite unsuccessful in the commercial world, we think that the developments in learning algorithms, the availability of computational power, and the

new trends in information security suggest that new research in the field of anomaly detection is indeed promising.

Unsupervised learning algorithms are natural candidates for the task, for a number of reasons:

Outlier detection: unsupervised learning techniques are capable of identifying "strange" observations in a wide range of phenomena; this is a characteristic we definitely need in an anomaly based IDS.

Generalization: unsupervised learning techniques are also quite robust and therefore can show better resistance to polymorphic attacks.

Unsupervised learning: we wanted to create a model totally orthogonal to the misuse based model, which is dependent on the input of expert knowledge, so we tried to develop an IDS which needed no a priori knowledge inputs.

Adaptation: a learning algorithm can be tuned totally to the specific network or system it operates into, which is also an important feature to reduce the number of false positives and optimize the detection rate.

1.3 Research objectives and original contributions

As we will see in the following, while a vast literature exists on the application of supervised learning methods to intrusion detection, there are still no convincing uses of unsupervised learning techniques (particularly in the network based field). By "convincing" we mean extensively tested architectures, that have been designed by choosing, step by step, which algorithm fits best on a particular type of data, instead of trying to "force" an algorithm to work on suitably prepared data.

Our objectives were:

- To propose realistic architectures, efficient and properly structured to be really deployed.
- To demonstrate, step by step, our assumptions and assertions on realistic data.
- To let the domain knowledge of the computer security field guide us to the choice of proper algorithms, rather than trying to fit the problem into a particular algorithm class.
- To implement a working prototype of our systems.

In this thesis, we describe our implementation of a network based and a host based anomaly detection IDS. Both systems have been prototyped and tested on well known datasets and on real world data.

Our key original contributions (part of which have been published by us in international conferences, but which never before appeared in mainstream literature as of our knowledge) can be identified as follows:

- We propose a two-tier architecture to analyze network packets overcoming the dimensionality problems which arise in the application of unsupervised learning techniques to network based anomaly detection.
- We consider performance issues and propose improvements and heuristics to increase the throughput of Self Organizing Maps to a rate suitable for online Intrusion Detection purposes.
- We propose an innovative host based system, based on the analysis of both the arguments and the sequence of system calls in UNIX or Linux processes.
- We carefully evaluate the detection rate and the false positive rates of both systems against well known datasets, comparing the results with benchmarks of state of the art systems described in previous literature.

1.4 Structure of the work

The remainder of this work is organized as follows. In Chapter 2 we introduce briefly the key concepts and problems of information security, and we characterize the main types of attacks against networked computer systems. This analysis brings us naturally to introduce the problem of tamper evidence in computer systems, and thus the concept of "intrusion detection".

In Chapter 3 we analyze the different technologies and types of intrusion detection systems, the problems and open issues to be solve, and the state of the art of the field, focusing on earlier studies on the application of unsupervised learning algorithms to intrusion detection.

In Chapters 4 and 5 we introduce, respectively, our research in network and host based applications of unsupervised learning to intrusion detection. We show the experiments we conducted and the resulting architectures we propose. For each type of system, we give performance results, comparing our achievements with other comparable existing systems.

Finally, in Chapter 6 we draw our conclusions, outlining the future directions of this work.

2 Computer and Information Security: an Overview

2.1 Formal definition of Information Security

2.1.1 The C.I.A. paradigm

Information is an asset that is essential to the operations of any organization, and consequently needs to be suitably protected. Information security [5] is the discipline that deals with ensuring three fundamental properties of information flowing through a system:

Confidentiality: the ability of a system to make its resources accessible only to the parties authorized to access them.

Integrity: the ability of a system to make it possible only to authorized parties to modify its resources and data, and only in authorized ways which are consistent with the functions performed by the system.

Availability: a rightful request to access information must never be denied, and must be satisfied in a timely manner.

This paradigm is known as the C.I.A. paradigm. Some people add other goals in their definition of Information Security, such as authenticity, accountability, non-repudiation, safety and reliability. However, the general consensus is that these are either a consequence of the three properties defined above, or a mean to attain them.

Also, as currently most information in the world is processed through computer systems, it is common to use the term "information security" also to denote "computer security"; but academically, information security spans to all the processes of handling and storing information. Information can be printed on paper, stored electronically, transmitted by post or by using electronic means, shown on films, or spoken in conversation. The U.S. National Information Systems Security Glossary defines Information systems security (INFOSEC) as:

> "the protection of information systems against unauthorized access to or modification of information, whether in storage, processing or transit, and against the denial of service to authorized users or the provision of service to unauthorized users, including those measures necessary to detect, document, and counter such threats."

This observation on information pervasiveness is especially important in the increasingly interconnected business environment. As a result of it, information is exposed to a growing number and a wider variety of threats and vulnerabilities, which often have nothing to do with computer systems at all.

In this work, however, we will deal mostly with *computer security*, i.e. the security of information handled by *computer systems*, and not information systems in general.

2.1.2 The A.A.A. architecture: access control methods and paradigms

The logical paradigm of confidentiality, integrity and availability of data and information contained in computer system is usually implemented with what is known as an A.A.A. architecture:

Authentication: the user is properly identified in some manner, and an access profile is associated with him.

Authorization: each operation and task activated by the user is subject to a set of constraints, given by the privileges he has to access system assets.

Accounting: operations are logged and reviewed with a proper process, in order to ensure that no violations of the C.I.A. paradigm have happened.

The A.A.A. conceptual taxonomy applies to networked operating systems and network services, but also to network control systems such as firewalls and VPN architectures (which are ways to allow or deny access to certain network services to certain hosts). This happens because the idea of authentication and authorization is orthogonal to most business processes and network services.

Authentication can be performed through various techniques, often divided into using something the user knows (such as a password), something the user has (a token, a smart card or any other sort of key object),

or what the user is (through biometric techniques such as fingerprint or iris scans).

Authentication and authorization are both concerned with the proper *identification* of users and the attribution of appropriate *privileges* for access to system assets.

There are two main paradigms for the management of this association: DAC (Discretionary Access Control) and MAC (Mandatory Access Control). A third paradigm is called RBAC (Role Based Access Control). While a complete review of these systems is beyond the scope of this work, we would like to briefly recall the key concepts.

DAC systems are simple: each object has an *owner*, and the owner can fully control the access privileges, granting and revoking rights to other users inside the system, up to the possibility of transferring the ownership privileges themselves to someone else. The *system administrator* manages system objects, and can usually preempt any privilege restriction on user objects.

Most commercial and free operating system in use nowadays (including all the Windows family systems, and most Linux and *BSD flavors) are DAC-based.

In a MAC system instead objects have a secrecy level (or more secrecy levels in various categories). Users instead have an access level (or a set of access levels). A "security officer" sets secrecy levels and access levels system-wide. One of the most famous MAC models, the Bell-LaPadula model [6] uses the following rules for granting or denying access:

1. Security rule: an user cannot read informations which have an higher secrecy level than his access level ("no read up").

2. *-property: an user cannot move information from an higher access level to a lower access level, but can move them upwards ("no write down").

In fact, this is a simplification, since the existence of set of access levels introduces a lattice of privileges where the concept of "above" and "below" (i.e., a total order relation) must be substituted by the concept of dominance (i.e. a partial order relation).

Since the natural entropy of this system would lead most information to be escalated towards more secret levels, an additional concept is needed: a *trusted subject* such as the security officer can, in fact, violate the rules and "declassify" information towards lower secrecy levels. These types of systems, however, tend to be very complex to manage: in fact, usually, operating systems use MAC just for data and documents rather than for the whole of the system files.

Another widely studied type of systems is RBAC (Role Based Access Control). In these systems, access privileges are assigned to different *roles*, and each user fits in one or more roles based on their occupation. See [7] for a detailed review of literature in this area.

The two fundamental means for expressing privileges are access control lists (ACLs) and capabilities. Access Control Lists are inherently more affine to the DAC model, and are expressed as a list of all the entities permitted access to each object in the system. On the other hand, Capabilities assert, for each user, the type of operations he is allowed to perform, and are inherently more useful in a MAC environment. The semantics of ACLs have been proven to be insecure in many situations, causing uncertainty and faults. Unfortunately, for historical reasons, capabilities have been mostly restricted to research or customized operating systems, while the commercial ones still use ACLs. A reason for the lack of adoption of capabilities is that ACLs appeared to offer a quick way to "add" security enforcement features without pervasive redesign of the existing operating system and hardware. However, TrustedSolaris or SELinux are examples of existing standard operating systems which support a MAC model as well as capabilities.

2.2 Applied Computer Security: Vulnerabilities, Risk and Assurance

2.2.1 Requirements, Specifications, Implementation: where vulnerabilities appear

In software engineering terms, we could say that the C.I.A. paradigm belongs to the world of *requirements*, stating the high-level goals related with security of information; the A.A.A. architecture and components are *specifications* of a software and hardware system architecture which strives to implement those requirements. Then, of course, security systems are the real world *implementations* of these specifications.

The trust we can place in this process can be expressed in terms of "assurance" [8]. Assurance can be defined as the basis for confidence that the security measures, both technical and operational, work as intended to protect the system and the information in processes and that the security objectives of integrity, availability and confidentiality have been "adequately" met by a specific implementation.

In a perfect world, implementation would be perfectly respondent to specifications, and specifications would meet and exceed requirements. However, as it is widely evident, we do not live in a perfect world [9].

Therefore, several orders of weaknesses afflict the path between requirements and implementation:

1. Analysis weaknesses in stating the requirements of confidentiality, integrity and availability for the information assets;
2. Design weaknesses while translating such high-level requirements into specifications in term of policies and architectures for authentication, authorization and auditing;
3. Implementation weaknesses while coding, deploying and configuring security systems;

In addition, since security requirements and their interaction with a forever changing environment are not stable, there is a need for a proper cyclic *development model* for maintaining the security system adherent to the changing business needs.

All these elements are well known in the software engineering field, and a lot of experience and best practices coming from that area can be applied. As we know, formal specification languages are needed to correctly design systems, but security practitioners have traditionally denied such need, preferring a more hacker-like approach (often denoted as "penetrate and patch").

However, as systems have grown larger, and more interconnected, this is no longer acceptable: complex systems need an appropriate, complete security policy specification, and this can be done fairly using common formal specification languages [10]. Other works have explored the use of different logical structures to express specification (as an example, lattice-based policies [11]).

2.2.2 Finding Vulnerabilities: Program Testing

Real systems are not perfectly respondent to the three requirements of security. The difficulty of ensuring the adherence of a program to its specifications is well known, and well expressed by Dijkstra [12]:

> Program testing can be used to show the presence of bugs, but never to show their absence

This trivially descends from well-known results of theoretical computer science, such as the halting theorem [13].

Additionally, while normal specifications are validated thinking of "cooperative" users and environment (that is, we do not assume that a user actively *wants* to break our specifications, or that he would try to cause

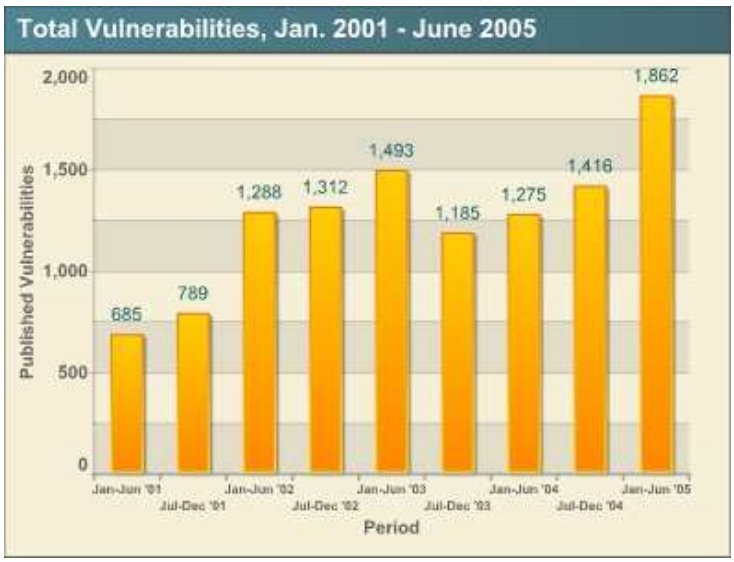


Figure 2.1: The number of vulnerabilities discovered in years 2001–2006

the program to fail unexpectedly), security specifications often need to prove their strength against a user who willingly tries to break or bend them, and who is often as skilled as (or even more skilled than) the person who designed them.

There are interesting proofs that most softwares, even if well designed, behaves anomalously under stress conditions: an interesting example is the utility FUZZ [14], which can be used to send the equivalent of white noise to applications; a tool called FIST (Fault Injection Security Tool) [15] uses instead a more evolved methodology called AVA (Adaptive vulnerability Analysis), which integrates white-box and black-box testing concepts to show even more weaknesses.

Even outside academical frames, the sheer number of vulnerabilities reported to public vulnerability forums (see Figure 2.1, which has been drawn from [16]) demonstrates that security vulnerabilities are indeed present, and that the discovery of such problems is in continuous growth.

Thanks to the principle of *full disclosure* this has become increasingly evident. Applying full disclosure [17] means to fully disclose to the public all the details of security problems when they arise, including a complete description of the vulnerability and how to detect and exploit it.

It is a philosophy of security management, which opposes the idea of "security through obscurity". The theory is that releasing vulnerability information results in quicker reaction by vendors and generally improves over time the security of systems. Security is improved be-

cause the window of exposure, the amount of time the vulnerability is known at least by some, and still unpatched, is reduced. Most researchers apply a so-called "responsible disclosure" policy, pre-alerting the vendor and coordinating vulnerability disclosure with a patch release. Disclosure is often achieved via public mailing lists such as Bugtraq (`www.securityfocus.com`).

Full disclosure was developed in reaction to the laziness of vendors in patching security vulnerabilities, relying on security through obscurity. Disclosure has often been the subject of heathed debates [18], but it's not new to the computer security scene. The issue of full disclosure was first raised in the context of locksmithing, in a 19th century controversy regarding whether weaknesses in lock systems should be kept secret in the locksmithing community, or revealed to the public. However, it is beyond the scope of this work to discuss such issues.

Getting back to our point, we do not wish to underline the shortcomings of traditional information security and software design practices, but to stress that every software, in the day to day practice, shows anomalous behaviors. In the security domain, anomalous behavior means that the access controls do not ensure anymore that the properties of confidentiality, integrity and availability are correctly respected.

2.2.3 Distinction between threats, vulnerabilities and risk

It should be noted that information security is, by itself, a science of uncertainty. As Bruce Schneier has it, "Information security is all about risk management": in other words, absolute security does not exist. The level of information security sought in any particular situation should therefore be commensurate with the value of the information and the loss, financial or otherwise, that might accrue from its improper use, disclosure, degradation or denial.

Various ISO standards [1, 19, 20], clearly defines the difference between risk, threats and vulnerabilities:

Risk: combination of the probability of an event and its consequence;

Threat: a potential cause of an unwanted incident, which may result in harm to a system or organization;

Vulnerability: a weakness of an asset or group of assets that can be exploited by one or more threats.

Information security tasks are all related to managing and reducing the risks related to information usage in an organization, usually, but

not always, by reducing or handling vulnerabilities or threats. So, it is wrong to think of security in terms of vulnerability reduction. Security is a component of the organizational *risk management* process (*a set of coordinated activities to direct and control an organization with regard to risk* [19]). In other words, information security is the protection of information from a wide range of threats in order to ensure continuity, minimize risk, and maximize return on investments and business opportunities.

Components of a proper risk management process are:

Risk analysis: the systematic use of information to identify risk sources and to estimate the risk;

Risk evaluation: the process of comparing the estimated risk against given risk criteria to determine the significance of the risk;

Risk assessment: the overall process of risk analysis and risk evaluation;

Risk treatment: process of selection and implementation of measures to reduce risks.

2.2.4 The Unpatchable Vulnerability: People

In the previous sections, we have voluntarily omitted the weakest link in the security of information systems, the unpatchable vulnerability of any security measure: the people, the users of our information system.

As we have seen, the traditional A.A.A. paradigm heavily relies on a proper authentication of users. We have also briefly seen the three classes of authentication methods (something you know, something you have, something you are). Passwords are still the most common, albeit insecure, form of authentication. Passwords can be shared (even innocently, just think about any university computer lab); they are usually badly chosen (short and easily guessable passwords are more comfortable to recall); they can be written down, often near the computer itself.

All these dramas have to do with user education to security, which has been described as "pointless" by Marcus J. Ranum [21]. There is a widespread problem of perception of security measures as being useful to avoid or reduce a real risk: for instance, no one complains for the fact that ATM withdrawal cards are PIN protected, but everyone complains every time a new password is introduced for accessing a computer system or application.

But apart from "distractions", experts agree that users often are willingly violating security measures [22]. Sometimes they do so with a criminal intent, either because they are disgruntled by the organization

or because they intend to commit some sort of white collar crime. Other times, security measures are perceived as an annoyance, a distraction from work, an intolerable intrusion into the personal workspace.

Finally, in many cases attackers abuse of user credulity in order to bypass security measures: this is what is known as a "social engineering" technique. Among the greatest social engineers of all times we can remind Kevin Mitnick, also known as "Condor", one of the most famous American digital criminals of all times, who was the subject of a nationwide manhunt by the FBI which became famous. Kevin Mitnick confessed that his core skills were not technical, he simply contacted people and convinced them to hand over their credentials.

2.3 Components of a Secure System Architecture

2.3.1 Planning a security system: policies and controls

Information security is achieved by implementing a suitable set of controls, including policies, processes, procedures, organizational structures and software and hardware functions. These controls need to be established, implemented, monitored, reviewed and improved, where necessary, to ensure that the specific security and business objectives of the organization are met. This should be done in conjunction with other business management processes [1].

A security policy is the overall intention and direction related to security issues, as formally expressed by management [23]. A correct security policy is the foundation of any secure system.

A security policy states, often formally, the high level requirements and specifications of an organization security system.

2.3.2 Security by design

A system should be securely designed from the ground up, rather than "secured" as an afterthought as often happens.

The first principle is enforcing privilege separation, i.e. giving any entity in the system only the privileges that are needed for its function. In this way, even if an attacker subverts one part of the system, fine-grained security ensures that it is just as difficult for them to subvert the rest.

Also, by breaking up the system into smaller components, the complexity of each components is reduced, thus opening up the possibility of formally proving the correctness of crucial software subsystems, e.g. through model checking techniques. Where formal correctness proofs

are not possible, a rigorous use of code review and testing can still be helpful in making modules as secure as possible.

This also enforces "defense in depth", since more than one subsystem needs to be compromised to compromise the security of the system and the information it holds. Subsystems also should ideally be failsafe, and the performance of the system should degrade gracefully while it is being compromised or damaged.

Subsystems and systems should default to secure settings as much as possible: it should take a deliberate decision on the part of legitimate authorities in order to make them behave insecurely.

A wise usage of cryptographic techniques can also help to ensure confidentiality and integrity in transmission or storage of information, as well as source authentication and non-repudiation. However, it should be kept in mind that cryptography is not, ultimately, the panacea for solving any security problem [24, 25].

2.3.3 Architectural Security

Architectural security means designing a network in such a way that makes it possible to correctly enforce a security policy on it. A firewall is the system that can enforce an access control policy between two or more networks. Various types and technologies of firewalls perform this access control mechanism in different ways. Usually, firewalls by default block all traffic which has not been specifically authorized. It is evident, then, that a proper configuration is really what makes a firewall effective or totally ineffective.

There are some threats that firewalls cannot protect against: for one, firewalls cannot protect against attacks that don't actually go through them. This means, for instance, that dial-up connections from the inside of the network, or towards the inside, cannot be controlled by a firewall. All the same, a firewall does not protect against malicious users inside the network, or against careless behavior by users.

Also, firewalls cannot usually distinguish between legitimate connections and connections carrying malware or exploit attempts.

2.3.4 Disaster Recovery and Business Continuity

Ensuring the availability of information also requires to plan for *disaster recovery* options. This is usually obtained through the creation of copies of data, called "backups", on various type of media and technologies. Data are usually also shipped off-site, to account for geographic disasters.

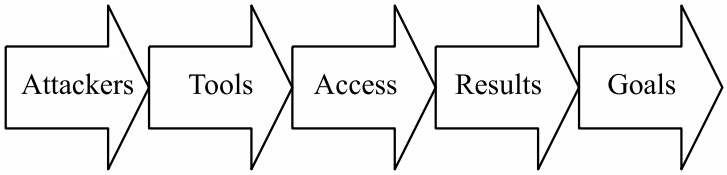


Figure 2.2: The CERT/CC Intrusion Process Taxonomy

Business continuity planning means taking this preparedness effort one step further, and creating an architecture which besides saving data ensures that applications and systems continue to function (usually in an alternate site) should the primary site be destroyed or damaged (physically or electronically).

Usually disaster recovery and continuity strategies are evaluated on the basis of two indicators:

Recovery Point Objective (RPO): the minimum target level of operativity that can be restored with the strategies in place. For instance, in case of backups, how "old" the restored data will be in the worst case;

Recovery Time Objective (RTO): the maximum time needed to restore the applications and the data to the operational level appointed by the RPO;

2.4 A taxonomy of threats: attackers and attacks

2.4.1 Motivations and former studies

To create a complete taxonomy of threats, attacks, and attackers of information systems is well beyond the scope of this work, and an open field of research. We gather here together our observations on the matter, along with some contributions from published research [26, 27, 28, 29]. We integrate them into the framework of what seems to be one of the most interesting attempts to describe a taxonomy for computer attacks, developed at CERT/CC [30].

In this taxonomy, the security incident is correctly modeled as a process with multiple stages and components, as shown in Figure 2.2. We modify this framework to express our own observations and opinions on the matter.

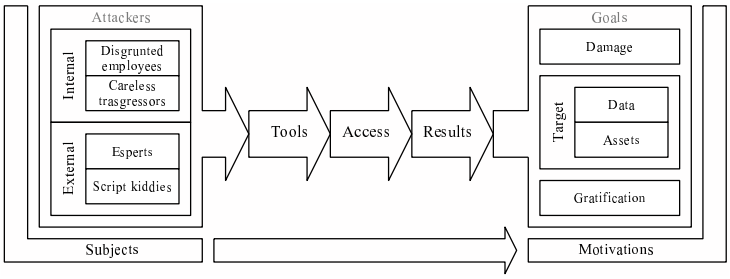


Figure 2.3: The relationship among attackers' motivations and goals

2.4.2 Attackers and their targets

As Sun-Tzu, the Chinese master of the art of war, would have it: "Hence the saying: If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle" [31]. In computer security, knowing ourselves mean knowing our systems, but even more knowing the assets we need to defend. Knowing the enemy means being able to understand the attacks, but even more being able to understand attackers, as well as their motivations and objectives.

In fact, since in most cases computer systems are attacked by people, analyzing and understanding the motivations is of foremost importance, and has inspired a number of works (notably, [32, 33]). As we already noticed, information security is an interdisciplinary environment, with a unique mix of computer science and social science problems to solve.

A first macro-distinction is between external and internal threats. We have already hinted that many attacks come from inside an organization (and indeed, nowadays it is difficult to even define what is "inside" or "outside" the perimeter). An internal aggressor has immense advantages, he usually knows the strengths and the weaknesses of the systems, and also knows where the crown jewels of the organization are. Internal aggressor can be consciously attacking the organization (often because they are disgruntled or corrupted), or unconsciously helping an external aggressor.

The technical skill level of the aggressor is another key differentiator: it may vary widely, from the more expert and skilled professionals to the script-kiddies, who use in a very dumb way automated scanners and tools created by others.

The objective of the aggressor is also important: some aggressors have a specific target (data, information, code,...); others may want to cause

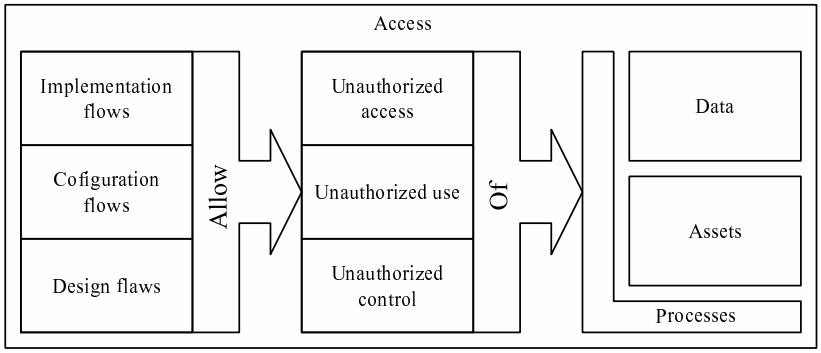


Figure 2.4: Attack Methodologies: a graphical representation

damage, e.g. as a form of protest or to wreak havoc on a competitor; others are just playing, stretching the security measures, exploring systems for fun. All of them have a varying grade of dangerousness, and the type of damages they can cause varies widely.

Even the target may vary: sometimes confidential data and information are the target, and the attacker tries to gain unauthorized access, or to modify or delete them. Sometimes the resources, the systems themselves, are the target: bandwidth and disk space are sought by script kiddies, MP3 traders, spammers, and so on. A flourishing black market of these resources is present [34].

The tie between aggressors, objectives and targets is the motivation for action, the reason for which someone attacks a system. This is the fundamental component of this problem, but often it is not properly taken into account. This concept is summarized in Figure 2.3.

2.4.3 Attacks: methods, tools and results

The heart of an attack is the subversion of the security policy of a system. In other words, the attacker somehow obtains the capability of performing an operation he should not be allowed to do. To do this, he usually starts, controls, hijacks or aborts system processes in an unforeseen manner, thus gaining access, performing commands, or making alterations to system data and resources beyond what the security policy would allow. This is usually accomplished by exploiting a security vulnerability, as we described above. The mechanism may wildly vary, but the core elements are shown in Figure 2.4.

Security vulnerabilities may exist in implementation of software: examples are input validation errors, such as SQL injections; buffer overflows [35]; wrong management of permissions; time of check to time of

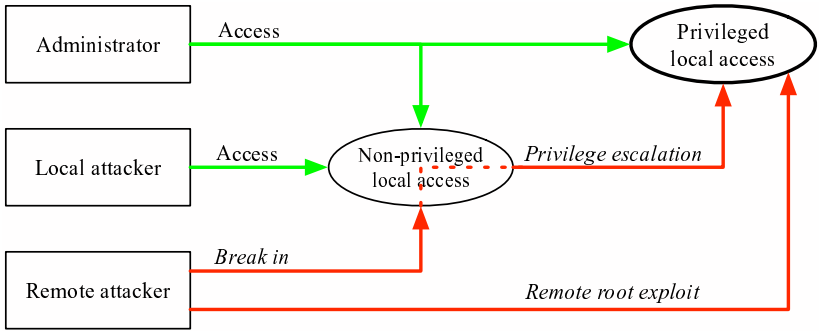


Figure 2.5: Escalation paths for an aggressor and for a normal user

use attacks and other attack paths. Vulnerabilities are also caused by misconfiguration of systems and programs [36]. Sometimes, vulnerabilities are inherent in a system or protocol design (for instance the WEP, Wired Equivalent Privacy protocol, is intrinsically flawed). We cannot really go in depth analyzing security vulnerabilities, as this would be far beyond the scope of this work. We refer the reader to [37, 38, 30, 5, 39] for further details.

We must, however, distinguish remote and local attacks. These terms do not denote the physical presence of the attacker in front of the machine, or the use of the system console. They make a distinction of privilege: a local abuser has some kind of access privileges to the system, a remote attacker doesn't. Usually, a remote attacker will penetrate the system first (break in) and then perform other attacks to obtain privilege escalation, usually to administrative powers (see the diagram in Figure 2.5). This type of compromise is called in slang "root-ing" a machine. Some attacks bring a remote user directly to root privileges (remote root attacks), and they are obviously the most dangerous type of vulnerabilities a system can show.

2.4.4 Conclusive remarks on attack taxonomies

An important missing factor in the CERT/CC taxonomy is the fact that an attack is not usually linear. Aggressions are usually perpetrated with a circular pattern: each successful exploit brings new power to the attacker, that can be used to further penetrate other systems. Another element which is not duly taken into account is the use of social engineering techniques to directly obtain access.

The complete scheme of the process is thus similar to the one in Figure 2.6

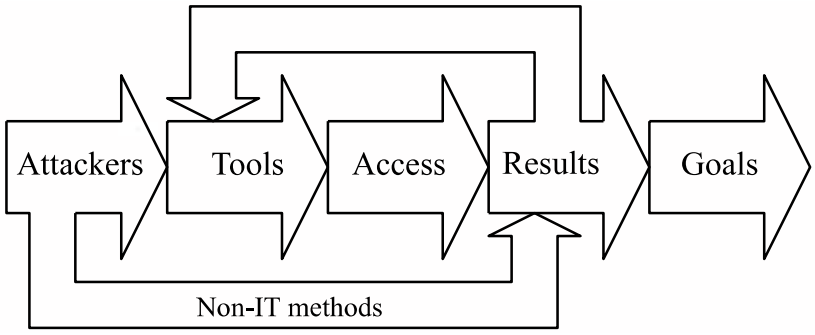


Figure 2.6: The complete diagram of the intrusion process

2.5 Intrusion Detection and Tamper Evidence

A well known problem in the art of war is the fact that the defender needs to plan for everything, while the attacker just needs to hit one weak spot.

As Baker notes:

> [the] philosophy of protection [...] depends upon systems to: behave predictably (they should do what we think they will do); be available when we need them; be safe (they should not do what we don't want them to do); be capable of protecting our data from unwanted disclosure,modification,and destruction; respond quickly. In other words, systems should be trustworthy".

Significantly, the title of the essay is "Fortresses built upon sand" [40]. As we have seen, almost none of these conditions are respected. This means that we must realistically consider information systems as being inherently insecure: software and hardware are not trustworthy, and people are willingly or unwillingly violating security policies. Furthermore, policy specifications can be incorrect, or incomplete, or incorrectly implemented.

As one of the Murphy's laws would have it: "The only difference between systems that can fail and systems that cannot possibly fail is that, when the latter actually fail, they fail in a totally devastating and unforeseen manner that is usually also impossible to repair". The lesson here is that every defensive system will, at some time, fail, so we must plan for failure. As we plan for disaster recovery and continuity, because disasters will happen at some point, we must design systems to withstand attacks, and fail gracefully. We must design them in a way

which makes it possible to recover them from attacks without losing data.

But even more importantly, any secure information system must be designed for being tamper-evident, because when it will be broken into, we want to be able to detect the intrusion attempt, in order to react to it. Since information systems are not, usually, tamper evident, we call Intrusion Detection Systems all the systems that can detect intrusion attempts, and possibly assist in post-attack forensics and recovery.

Relatively few organizations maintain computer systems with effective detection systems, and fewer still have organized response mechanisms in place.

3 Learning Algorithms for Intrusion Detection Systems: State of the Art

3.1 What is an Intrusion Detection System ?

As we stated in the previous chapter, we need a complementary approach to help us make computer systems tamper evident, i.e. to help us detect intrusion events, alerting security personnel for reaction.

What we need is an Intrusion Detection System or IDS, which is the computer system equivalent of a burglar alarm. The concept of a system capable of detecting intrusions was introduced in 1980 by J.P. Anderson [2].

The idea behind any type of IDS is that any information system is designed to serve some goals, and the three properties of security are aimed to ensure that the information system is not abused to do something else. Thus, when someone willingly violates the security paradigm of an information system, his behavior and/or the behavior of the system will somehow differ from the "normal" behavior. Ideally an IDS would detect these behavioral anomalies and tag them as suspicious.

3.2 A taxonomy of Intrusion Detection System approaches

3.2.1 Anomaly based vs. Misuse based

Intrusion Detection Systems can be broadly divided in two main categories, based on two different approaches: anomaly detection or misuse detection.

Definition 3.1 *An anomaly detection IDS tries to create a model of normal behavior for the monitored system(s) or for their users, and flags as suspicious any deviation from this "normal" behavior which exceeds*

Misuse Based	**Anomaly Based**
Require continuous updates	Do not require updates
No initial training	Long and complex training
Need tuning	Tuning included in training
Cannot detect new attacks	Can detect new attacks
Precise alerts	Vague alerts
Almost no false positives	Huge numbers of false positives
Lots of non contextual alerts	No non contextual alerts
Easier to design	More difficult to design

Table 3.1: Comparison between strengths and weaknesses of anomaly based and misuse based IDSs

carefully tuned thresholds. This is surprisingly similar to the earliest conceptions of what an IDS should do [2].

Definition 3.2 *A misuse detection IDS uses a knowledge base (often called a set of signatures) in order to recognize directly the intrusion attempts, which means that instead of trying to describe the normal behavior of a system it tries to describe the anomalous behaviors.*

These systems have symmetric strengths and weaknesses that can be summed up as in Table 3.1. The great strength of anomaly detection systems is that they do not require a continuously updated knowledge base to be created "a priori", since they model the normal behavior of the system they act upon. This makes them capable of detecting new and modified attacks, a very desirable property.

However, in order to build a model of "normal behavior" we need to define both what variables we are going to monitor, and what kind of model we are going to use to model them: this translates into anomaly detection systems being much more difficult to design and test than misuse detection systems.

Another negative point is that most anomaly detection systems need a rather long training phase, during which the IDS is not effective and can, in some cases, be sensitive to the insertion of attacks into training data, since they would be built "into" the system profile as normal behaviors. As a final note, these system are known to be prone to errors and false positives.

Misuse based systems, on the other hand, require an extensive study of the attacks in order to build and keep up to date a base of signatures. The efficiency of these systems is directly dependent on the quality of the knowledge base (a parallel can be drawn here with the world of

antivirus software). These systems are less prone to false positives, but if misconfigured they can generate a huge number of unwanted, non contextual alerts that can make them useless. A key advantage is that misuse detectors are precise: when an alert is triggered, we know what attack signature was activated and we can look for that specific attack and even activate an automatic response (a model known as "Intrusion Prevention System" or IPS). An anomaly based system would instead just flag a connection or event as anomalous, without specifically saying what is out of ordinary.

However, these features of misuse based systems are paid dearly and in advance: maintaining a wide, up to date knowledge base of the attacks is an impossible task, for at least two reasons:

1. As we saw when speaking of disclosure, not all the new attacks are immediately released to the experts for analysis; many vulnerabilities are undisclosed, and attacks for these vulnerabilities (the so-called "zero days") cannot be catched by misuse detectors, except in the lucky case that these attacks closely resemble earlier attacks against different software, or that a part of these attacks, alone, is recognizable.

2. Some forms of attack (in particular attacks against web applications) can be studied by a skilled attacker right on the spot, just to hit a single or a few systems: in this case, no suitable signatures can possibly exist, except once again if the attacks closely resemble other classes of attack and can be detected by the same signatures.

Additionally, most computer attacks are inherently polymorphic. If multiple sequences of actions can lead to a compromise, it is correspondingly more difficult to develop appropriate signatures: either we develop more and more signatures for each possible variation of the attack, or we try to generalize the signatures. In the first case, we may have a complexity problem, while in the second case we expose ourselves to the risk of triggering false positives.

Let us analyze three classical examples of this problem:

Example 3.1 *All the "Unicode" related bugs are inherently polymorphic since in the Unicode character set there are multiple possible codes for each character. This means that either we build an exponential number of signatures, or we must apply a decoder/canonicalizer before matching against the signatures of the attack. Either way, we evidently incur in performance problems.*

Example 3.2 *The ADMmutate tool [41] enables an aggressor to encrypt the shellcode of a stack-smashing buffer overflow attack [35]. Instead of using the binary sequence of commands that he wants to run on a target machine (let it be sequence "A"), the aggressor encrypts this code (let it be $C(A)$), and puts a decryption wrapper around it (the function C^{-1}), thus fooling any IDS which blindly looks for string A, which is completely different than the resulting $C^{-1}(C(A))$; however, when executed, the latter generates (and executes) A. Even if most IDSs nowadays have a specific signature for the function C^{-1} it's easy to understand that this principle can be indefinitely applied, in many forms.*

Example 3.3 *Another example based on buffer overflow attacks is the so-called "NOP signature". Most attacks of this kind use "no operation" instructions (on Intel x86 assembler, "NOP", hexadecimal code $0x90$) as "padding", because it's sometimes difficult to understand where, exactly, code execution will begin. In this way, wherever the code happens to begin execution, the processor finds a series of NOP instructions ("NOP sled") and arrives safely at the beginning of the real attack code. Most IDSs thus detect a long sequence of NOP as a possible shellcode. Sneaky attackers thus use a jump to the following address (JMP 0, $0xEB0x00$) instead of a NOP. If such a signature is also added to the IDS knowledge base, they could jump ahead of an arbitrary number of positions instead, and always find a way to fool the IDS.*

Drawing an analogy, if security is an eternal chess game, most chess players can tell you that a "mirror game" in which black continuously mirror white's move is invariably won by white. And aggressors, with this approach, always have the first move.

In fact, misuse based IDSs are known to be particularly effective against the so-called "script kiddies", unskilled attacker that rely on commonly known attack tools, for which a signature is usually wide available.

Most commercial systems are substantially misuse based. Anomaly based systems have been mostly developed in academic environments. Some systems try to integrate the approaches, but there's a difficult problem of metrics to compare signature based alerts and anomaly alerts.

3.2.2 Network based vs. Host based

Another distinction can be drawn on the base of the source of data being audited by the intrusion detection system, between *network* based and *host* based systems.

Definition 3.3 *A host based IDS controls a single machine, sometimes even a single application, and depends on data which can be traced by the operating system of the monitored host, e.g. system calls, resources usage, privilege escalations, and/or system logs.*

Definition 3.4 *A network based IDS is connected to a network segment and tries to analyze all the traffic which flows through the segment (usually, by the means of a network sniffer), trying to detect packets which could be part of an attack.*

Both methods have their own advantages and disadvantages. Network based IDSs can control networks of arbitrary size with a small number of (almost) invisible sensors (using unnumbered sniffing network interfaces in promiscuous mode), but they obviously can detect only attacks which come through the network (which, of course, is nowadays the main vector of aggression). In addition, some network communications are encrypted, effectively blinding the IDS. If an IDS has signatures for an attack against a web application, but this attack happens in an SSL communication, no alert will ever be triggered.

Network based IDSs also suffer from the need to reconstruct the effect of the packet flow on every single host from a central point, taking also into account the variability of the implementations of the TCP/IP standards among different operating systems. While this would be theoretically possible for an arbitrarily powerful system, in real case implementations there are performance constraints, so an IDS must use simplified approaches that lead to attacks such as insertion and evasion, in which the IDS is unable to correctly reconstruct the packet flow observed by a system [4]. Similar difficulties arise from the fragmentation of TCP/IP packets, which different stack implementations handle differently.

Host based systems were the first ones to be developed and deployed, primarily drawing information from system logs, as in [2]. However, they presented a number of drawbacks: the peer-to-peer structure of nowadays information systems is much less suited to host based detectors than the server-and-terminals structure of the 80's. The widespread use of the network computing paradigm and the explosive growth of the Internet made the network one of the most sensitive attack points (even for internal attacks). The additional perceived and illusionary simplicity of a "connect and forget" system also helped. So, if the first really impressive example of a network based IDS, Network Security Monitor [42], was developed much later than the first examples of host based IDS, nowadays such systems are fundamentally network based.

It is important to note, however, that since failures and strengths of such approaches are symmetric, some systems try to integrate them [43], but there are difficult and intriguing problems of metrics, fusion and normalization when working on data coming from different sources, somehow tied to the "multi-sensor data fusion" problems already under consideration in the field of robotics [44]. We will not try to address such problems in this thesis, however, because this would be enough for an entirely new work of this same size.

3.2.3 Centralized vs. Distributed Architectures

Even if, for simplicity, we talk of IDS systems as monolithic entities, they are often composed of distributed components, sometimes of different types. As we stated before, network and host based probes are complementary, and also anomaly and misuse based systems can be combined. For such a distributed monitoring network, an appropriate collaboration infrastructure is needed in order to help correlation engines cross system boundaries.

Beyond the already mentioned problem in data normalization and fusion, appropriate decisions must be taken and alerts must be generated. In [45] various inference engines for coordination are proposed, based on Bayesian rules and on graph analysis. Management and correlation can happen in a centralized engine, often with a hierarchical structure where "lower layer" components detect atomic events and "higher layer" components correlate and build scenarios on them [46]. Otherwise, the system reasoning can be distributed among the various probes in a multi-agent fashion. Examples of distributed, agent based infrastructure are HIDE, Hierarchical Intrusion DEtection [47], AAAFID, Autonomous Agents for Intrusion Detection [48, 49] or EMERALD [43]. These architectures allow for great flexibility and extensibility, as well as greater survivability in the face of overload or attack.

3.3 Main Issues in Intrusion Detection

3.3.1 Comprehensiveness of model

As we stated above, the types of attacks are continuously evolving. If the micro-evolution poses just updating problems, macro-evolution steps, such as the discovery of new types of attack, may make the IDS base model outdated.

IDS models must be as comprehensive as possible. For instance, a sensor operating at layer 3 of the ISO/OSI stack will never detect attacks

at layer 2, no matter the knowledge base updates. The engine itself must be rewritten or patched to go below the original layers.

Anomaly detection and misuse detection systems suffer of this problem in a similar manner. If an attack shows up only in the variables that an anomaly detection system does not measure, then the IDS is blind to it. It is easy to imagine forms of attack specifically studied to find and exploit these "dead spots". An interesting example is in [50].

3.3.2 Zero-day attack recognition

As we already stated in Section 3.2.1, by their own nature, misuse based systems are unable to deal with unknown attacks: thus, the growing number of vulnerabilities discovered every day requires a continuous update of their knowledge base. In addition, there is also an unknown number of discovered but undisclosed vulnerabilities (the so called "zero-days" [3]) that are not available for analysis and inclusion in the knowledge base. If we add attacks that are specifically studied to compromise a custom application, or a specific system, we can see that misuse detectors have a severely limited *coverage* of attacks. In fact, misuse-based IDSs are mostly effective against unskilled attackers that rely on commonly known attack tools, for which a signature is usually widely available. This evidently excludes the most dangerous attackers.

Since Intrusion Detection Systems are intended to be a complementary security measure, which can detect the failures of other measures, the inability to detect unknown attacks (or new ways to exploit an old vulnerability) is an unacceptable limitation. For this reason, some vendors in their literature have re-defined a zero day not as an undisclosed vulnerability, but as a new exploit for an already known vulnerability. In this sense, since well-written signatures can catch new exploits for some vulnerabilities, misuse detectors can claim some sort of zero day recognition, but this should be recognized as just a marketing hype.

In truth, zero-days are by definition beyond the grasp of misuse based systems, and one of the key reasons

3.3.3 Intrinsic security and survivability

An IDS should be designed to be as secure as possible, and also designed for survivability, since it has a similar function as an aircraft emergency flight recorder. Since the IDS logs can quickly become the only valuable source of information on a security breach, it is important that the IDS system itself is not compromised.

Subversion of an IDS can disable the intrusion alerts, generating a

false sense of security, and can lead to irreversible alterations in logs and traces. A distributed IDS is also vulnerable to common attacks against communication between components, and must adopt all the techniques to ensure end-to-end communication security and confidentiality.

3.3.4 Flexibility and Usability Issues

Security personnel can be trained to use an IDS system, but really knowledgeable people are a scarce resource worldwide. As a result, IDS alerts must be as clear as possible (this is typically an issue for anomaly detection systems), and management consoles must be designed for usability.

Additionally, flexibility is required to adapt the systems (particularly commercial, off-the-shelf IDS softwares) to different usage scenarios.

3.3.5 Scalability and Throughput

An IDS system must scale to handle the ever-increasing throughput of today's networks and computer systems. For instance, network based systems must be specifically redesigned for high performance, e.g. the BRO project [51]. In the case of in-line systems such as intrusion prevention devices, response time (which is not the same as the throughput) is also of foremost importance.

Network IDSs, when overloaded, typically enter a random discard phase, loosing part of the incoming packets. This problem can be modeled with well known techniques of performance theory, such as blocking queueing networks [52], and we even performed some researches in that direction [53]. However, the evaluation of intrusion detection systems is a difficult and open research topic [54], an in-depth analysis of which would be outside the scope of this work.

If the dropped packets are part of an attack, the attack is probably lost. If this coincidence seems difficult to happen, let us consider the problem from an attacker perspective: overloading an IDS and making it loose packets is an easy way to avoid detection. Generation of a huge volume of false alerts is another useful technique to overload an IDS and make the logs completely unreadable as well.

3.3.6 Ambiguity in event reconstruction

Reconstructing a scenario from a disperse set of events (e.g. reconstructing network traffic sessions, or correlating log events across a large network) is one of the most difficult tasks for an IDS. Propagation time, the best effort nature of TCP/IP networks, time skews on timestamps, all

contribute to create difficulties on time-based reconstruction and correlation algorithms. Moreover, different scenarios usually tend to be interleaved, mixing together alertsm and creating confusion.

Writing signatures for scenarios (or aggregating them through various techniques) is also not easy, due to the problems of unification (the possibility of an arbitrary number of different alternatives to be present at a step) and of partial ordering. Artificial Intelligence researchers would define the situation to be in the class of uncertain reasoning problems, which are known to be easier to solve in an off-line fashion rather than in an on-line environment.

These difficulties often lead to serious attack windows, since it is enough for an attacker to disperse the steps of his aggression over a long time in order to avoid detection through correlation.

Network based traffic reconstruction also presents some uniquely difficult problems to solve. Theoretically, an arbitrarily powerful system should be able to reconstruct all the sessions, by observing all the network traffic through sniffing probes. However, in real systems this cannot happen. In fact, it is very difficult to understand, from a single point of view, exactly how packets will be reconstructed on the endpoints. This was beautifully demonstrated in [4], leading to evasion techniques that still work nowadays on many network based IDSs. Particularities in the TCP/IP network stack implementations, as well as the topology of the network, can be used by a skilled attacker in order to insert fake packets that will never reach the target host (insertion attacks), thus creating sequences that the IDS will reconstruct in a different way than the target. For instance, a packet could have a TTL set in such a way that it will be seen by the IDS, but discarded before reaching the target host.

On the other hand, using evasion techniques (such as fragmentation) the attacker can try to create sequences that once reconstructed will create an attack, but on the network will look legitimate. Fragmentation has proved to be particularly effective: most IP stacks handle exceptions in a different way, making it very difficult to figure out how exactly the reconstructed packets will look. Fragroute [55] uses this concept to hide attacks in fragmented packets, also using the techniques from [4].

Combining evasion techniques creates very effective attacks, even if an IDS is resistant to the single components, as shown in [56].

A perverse relationship makes it so that the more an IDS engine is resistant to evasion attacks, the more it is likely to fall prey to insertion. Usually, misuse based systems, being based on non-flexible signatures, are more prone than anomaly detection systems to fail against these techniques.

3.3.7 Target based correlation

Knowledge on the network topology, on the services offered by different systems and on the operating system and software versions can be used in order to prioritize or filter the alerts, for instance discarding "out of context" alerts such as attacks against wrong operating platforms (i.e. a Linux exploit against a Windows machine). This type of knowledge can evidently be exploited only in misuse based systems, as they are the only one generating alerts with enough knowledge attached to be managed in this way.

This technique is debated because there is a trade-off between having to deal with less security alerts, and the possibility of recognizing attack actions even if the attacker is making mistakes. In addition, the need to maintain an accurate map of the protected network - including valid points of vulnerability - creates a further updating problem in misuse based systems.

3.3.8 Reactivity and Intrusion Prevention

Reactivity, or the ability to stop attacks as well as flag them, is probably the Holy Grail of Intrusion Detection. Dubbed "Intrusion Prevention Systems", reactive IDSs have been marketed as a security panacea, but they are really more of an unexplored territory with many questions left open.

A first problem is a performance issue: on-line systems just need to have a throughput high enough to avoid dropping packets; if a system is placed "in-line", acting as a gateway very much in the same way a firewall would, its response time becomes important, as it is the delay it is introducing onto the network. In a very similar manner, a host based IPS could easily overload a very crowded system, but this is easier to deal with.

A second problem is the possibility of denial-of-service. If a reactive network IDS blocks services based on detected attacks, a spoofed attack packet could be enough to block legitimate connections. False positives make this problem even more troublesome.

A third problem is architectural: in order to block attacks, a network based IPS should be placed on an enforcement point, usually in cascade or on board of a firewall. This makes it ineffective against internal attacks, which were one of the reasons that led to the very development of IDS systems. Some network based systems try to deal with this problem using RST packets to kill connections even if not placed directly in the middle of the network path, but this is offen inefficient. For this

reason, host based IPS are a much more effective choice, in our opinion.

Aggressive reactions, or counter attacks, have been proposed also, but the possibility is shunned by security experts: if we just think about the false positive problems, we can easily see why, without even beginning to take into account legal liabilities tied to a "vigilante" behavior.

3.4 Learning Algorithms: supervised and unsupervised

In the following, we will make use of terminology drawn from machine learning literature. Machine learning is an area of artificial intelligence concerned with the development of techniques which allow computers to "learn", or, more specifically, concerned with the creation of algorithms whose performance can grow over time. Machine learning is heavily related with statistics, since both fields study the analysis of data, but unlike statistics, machine learning is concerned with the creation of algorithms of tractable computational complexity. While we refer the reader to [57, 58] for an in depth analysis of the topic, we introduce in this paragraph some of the key terminology of this area.

The first distinction is between *supervised* and *unsupervised* learning algorithms. Supervised learning algorithms generate their model (i.e. they learn) from a labeled dataset, i.e. a dataset where inputs are labeled with the desired outputs. An example is is the classification problem: the algorithm is required to learn a function which maps a set of input vectors into one of several classes by looking at several pre-classified examples.

Unsupervised algorithms instead try to model a set of inputs according to inner criteria (usually statistical density-based criteria of some sort). The unsupervised problem which is symmetrical to the classification problem is called "clustering" (see Section 4.3.1) and means creating "natural" groupings of similar elements, without example classifications to work on.

From a theoretical point of view, supervised and unsupervised algorithms differ in the causal structure of the model. In supervised learning, the model defines the effect of a set of observations (the inputs), on the set of the labels (outputs). The models can include hidden, mediating variables between the inputs and outputs. In unsupervised learning, on the contrary the observations are assumed to be the outputs, and to be caused by latent variables.

With unsupervised learning algorithms it is possible to learn larger and more complex models than with supervised learning, because in

supervised learning the difficulty of the learning task increases exponentially in the number of steps between the set of inputs and outputs. In unsupervised learning, the learning time increases (approximately) linearly in the number of levels in the model hierarchy.

In our application domain, the use of *supervised learning* has two main drawbacks. The first is that, requiring examples of both normal behavior and attacks, these systems are only partially anomaly based, and often work more like a generalized misuse detector. But the main drawback is that a huge labeled dataset is needed for training. This can be either artificially generated (and therefore not really representative), or a manually labeled dataset of real world events (which is evidently difficult to obtain, in particular in a network based environment).

3.5 Anomaly Detection Systems: State of the Art

3.5.1 State of the art in misuse detection

Misuse detection techniques have evolved little over the years. The simplest form of misuse detection, expression matching, searches an event stream (usually log entries in host based schemes, or network traffic), for occurrences of specific attack patterns.

Snort [59], a lightweight network intrusion detection system which is arguably the most famous open source product in this area, uses a rather advanced form of these early techniques in order to flag attacks. Advancements in this area include various techniques to improve the performance of rule matching engines such as the ones presented in [60] (where clustering of signatures is used to accelerate the matching process). Similar features are offered by other systems such as NFR [61] or BRO [51]

These rules however do not represent contexts. This results in poor expressivity, at times. In works such as [62] pattern-matching signatures are complemented by higher-level knowledge on network and connection status. Other approaches are more general. For instance, state-transition analysis has been proven an effective technique for representing attack scenarios [63]. In this type of matching engines, observed events are represented as transitions in finite state machine instances representing signatures of scenarios. When a machine reaches the final (acceptance) state, an attack has been detected.

This approach allows for the modeling of complex intrusion scenarios, and is capable of detecting slow or distributed attacks, but somehow complicates signature generation. Other state machine representations (e.g. colored Petri nets) offer similar advantages.

There are also quite interesting approaches which try to create "generalized" signature matches. For instance, the GASSATA system (Genetic Algorithm as an Alternative Tool for Security Audit Trail Analysis) [64] uses a genetic algorithm to search for combinations of known attacks. One of the evident drawbacks of systems such as these is that, like anomaly detectors, they cannot offer an explanation for any positive match they encounter.

3.5.2 Host based anomaly detection

Anomaly detection has been present in the intrusion detection concept since the very inception, in the seminal works by Anderson [2] and then Denning [65]. At that time, obviously, *host based* techniques were the focus.

Various types of approaches have been widely researched in literature, and they can be divided as follows, without pretending to be taxonomically sound.

Statistical models

In [65] a number of statistical characterization techniques for events, variables and counters were first outlined. IDES used parameters such as the CPU load and the usage of certain commands in order to flag anomalous behaviors. NIDES in its statistical component [66] further developed this scheme. Examples of these early statistical models are:

- Threshold measures, or "operational model" [65], in which standard or heuristically-determined limits are used to flag anomalous rates of event occurrences over an interval (e.g. on the number of failed login attempts);
- Computation of mean and standard deviation of descriptive variables, in order to compute a confidence interval for "abnormality";
- Computation of co-variance and correlation among the different components of multivariate measurements on a computer system.

Another interesting approach is the use of an incidence matrix command/user, which is searched for structural zeroes representing rare commands [67]. More complex theoretical works (e.g. [68, 69]) have also followed this purely statistical approach, sometimes with very interesting results. Most of these works, however, do not take into account the sequence of events, but just atomic events, or their quantity over a sliding time window.

Immune Systems

In the immune system approach, computer systems are modeled after biological immune systems. In reality, artificial immune system have been proposed as a computational approach for solving a wide range of problems [70]. These approaches have been widely presented in intrusion detection literature (with earliest ideas ranging back as far as 1997 [71]), but they never became mainstream, and are so heterogeneous they would deserve a taxonomy and literature review effort of their own. We limit ourselves to refer the reader to [72, 73, 74].

File Alteration Monitoring

Sometimes known as "Tripwire", from the name of its most widely deployed and best known representative, this intrusion detection technique uses cryptographic checksums (hashes) of sensitive system data in order to detect changes to critical system files - including unauthorized software installations, backdoors implanted by intruders, configuration file changes, and so on. Such information can be of invaluable help in detecting attacks, in recovering a compromised system, as well as in forensic post-mortem examination.

Of course, the core problem of this methodology is that if the checksum database is conserved locally, it can be altered. Moreover, the host based program which routinely checks for tampering can be damaged or subverted, thus making this type of systems inherently unreliable.

Whitelisting

Whitelisting is a very simple, yet effective, technique for reducing an event stream (e.g. a system log, a connection trace, etc.) to a humanly-manageable size. It involves passing the stream through a cascade of whitelisting filters corresponding to known benign patterns of events. What remains after known events have been filtered out are either novel or suspicious events. If classified as normal, they enter the whitelist for future filtering, otherwise they undergo detailed analysis. This is therefore a very basic form of supervised learning. Described with the name "artificial ignorance" in a seminal work by M. Ranum [75], this technique is more a post-processing technique for intrusion detection alerts than a standalone technique for detection. In addition, it has several limits, first of all the difficulty in recognizing the fact that whereas a single event of some type (let's say, a failed user login) is meaningless, a long sequence of the same event may instead be highly suspicious.

Burglar alarms and honeypots

An alternative approach is to focus on identifying events that should never occur. This type of techniques, named "burglar alarms" by Ranum, are concerned with creating monitors that look for instances of policy violation, effectively placing "traps" which attackers are prone to trip. For instance, we could monitor any outbound connection from an HTTP server, if it is not expected for this machine to make any such connection. This type of detection is independent of attack description and therefore qualifies as anomaly detection. However, it requires care and extensive knowledge of the network administrator to properly lay down such traps.

This approach brings into consideration the use of honeypots [76] as intrusion detectors. An honeypot is a resource whose value lies in being compromised by an attacker. Since an honeypot does not have any legitimate use, access to such a resource is usually a very good indicator that something strange is happening. In fact, this approach is so promising that some authors have proposed a way to extract information from honeypots and use it to build misuse detection system signatures [77].

It is important to note that these approaches can be as effective against outsiders as against insider attacks and privilege abuse [78].

Supervised Learning

Supervised learning algorithms have also been applied for host based intrusion detection purposes. For instance, Neural Network algorithms have been used to analyze interactive user sessions (such as NNID, Neural Network Intrusion Detection [79], but see also [80, 81, 82]). Neural networks avoid an arbitrary selection of thresholds, but are nevertheless sensitive to proper selection and preconditioning of input values. A common critique that can be drawn against many published works in this particular area is the arbitrariness of selection of observed variables.

The learning ability of these systems allows to compensate for behavior drift through constant retraining. This is however a difficult choice, since then an attacker could slowly use the semantic drift to retrain the network to accept his behavior.

An alternative, supervised approach is based on data mining, as seen in [83, 84]. These approaches have the advantage of giving insights on how the features can be selected, how they interact with each other, and on the appropriate models to fit them. On the other hand, the output is less usable on a real-world system than in most other cases. Other researchers proposed using Instance Based Learning (IBL) techniques

[85], which have both supervised and unsupervised applications.

Unsupervised Learning

Various unsupervised learning techniques have been used for host based intrusion detection. Some of the more advanced applications of statistics can already be defined as "learning algorithms". For instance, some uses of Markovian process models can be ranked as such. Among them, clustering techniques to group similar activities or user patterns and detect anomalous behavior [72];

Even genetic algorithms have been proposed for this task [86].

Host based anomaly detection using system calls

In this paragraph (which is not strictly taxonomical, since it overlaps the previous ones), we introduce the main contributions on the analysis of the sequence of system calls invoked by programs, which will be the focus of most of the research outlined in Chapter 5.

The very first approaches dealt with the analysis of the sequence of syscalls of system processes. The first mention of the idea is in [87], where "normal sequences" of system calls (similar to n-grams) are considered (without paying any attention to the parameters of each invocation). A similar idea was presented earlier in [88]: however, the authors of the latter paper suppose that it is possible to describe manually the normal sequence of calls of each and every program. This is evidently beyond human capacity in practice. However, an interesting element of this paper is that it takes into account the values of the arguments of syscalls.

Variants of [87] have been proposed in [89, 90, 91, 92]. This type of techniques have also been proposed as reactive, IPS-like components [93].

An inductive rule generator called RIPPER [94, 95], invented for text classification, has been used for analyzing sequences of syscalls and extracting rules [96]. This type of approach can also been used for automatically defining protection policies, i.e. for intrusion prevention purposes [97, 98].

Finite State automata have been used to express the language of the system calls of a program, using deterministic or nondeterministic automs [99, 100], or other representations, such as a call graph [101]. Hidden Markov Models have also been used to model sequences of system calls [102], with better detection results but with computational problems [103].

In [104] a detailed review of different approaches is presented, along with a comparative evaluation on live datasets that are unfortunately not available anymore for testing. An Elman Network, a recurring neural network with memory properties, has also been used [105].

None of these methods analyzes either the arguments or the return values of the system calls. This is due to the inherent complexity of the task, but the arguments contain a wide range of information that can be useful for intrusion detection. For instance, mimicry attacks [106] can fool the detection of syscall sequence anomalies, but it is much harder to devise ways to cheat both the analysis of sequence and arguments.

Two recent research works began to focus on this problem. In [107] a number of models are introduced to deal with the most common arguments. We discuss in depth and extend this work in Chapter 5. In [108] an alternative framework is proposed, using the LERAD algorithm (Learning Rules for Anomaly Detection) which mines rules expressing "normal" combinations of arguments. Strangely, neither work uses the concept of sequence analysis. A concept named "Resilience" has also recently been introduced [109], involving the mapping of arguments of system calls as multidimensional data points. However, this approach is still in the early stages of development.

3.5.3 Network based anomaly detection

Anomaly detection algorithms have been applied also to network intrusion detection, mostly using statistical techniques. Again, we roughly divide the approaches in literature, without a pretense to be complete or taxonomically sound.

Protocol Anomaly Detection

Many attacks rely on the use of unusual or malformed protocol fields, which are incorrectly handled by target systems. Protocol anomaly detection techniques (also known as "protocol verification") check protocol fields and behavior against standards or specifications. This approach, used in commercial systems and also presented in literature [110], can detect some commonly used attacks, as well as lots of faults in standard compliance. They are therefore prone to generate false positives. On the other hand, many attacks do not violate the specifications of the protocols they exploit, and are therefore undetected by this approach.

It is also worth to note that evaluations on protocol anomaly detection performed over the DARPA dataset benefit from the artificial anomalies we describe in Section 4.6 and are therefore unreliable.

Supervised Learning

Supervised learning algorithms have also been used. In [111, 96, 84] the authors fully explore this approach using data mining techniques along with domain knowledge, with interesting results. ADAM [112, 113] is a rule-based, supervised system that mines association rules for detecting anomalies in TCP connection traces.

More sophisticated examples of supervised, statistical analysis and classification of network traffic anomalies have been also proposed in [114].

As noted before, the problem here is the need of a huge dataset of labeled traffic, something which is really difficult to obtain.

Simple statistical methods

Simple statistical methods have been used also in the analysis of network anomalies. This was already present, for instance, in NIDES [115]. Usually, these systems work on network-wide feature variables dealing with global traffic volume [116]. This is also true in most of the few network based anomaly detection systems commercially available today.

Statistical methods have also been applied to packet information, and in particular to the information in the packet headers, discarding packet content. For instance, PHAD, Packet Header Anomaly Detection [117, 118], is a simple statistical modeling method which has been applied to data extracted from packet headers. More complex, information-theoretic methods such as the Parzen Window method have also been proposed [119], but they suffer from a need to be able to characterize statistically the observed variables, and from low throughput.

Unsupervised learning and outlier detection

Unsupervised learning techniques are more difficult to apply to network data, for dimensionality reasons that we will discuss in detail in Chapter 4. Also in this case, the information of packet headers, or summary information on connections, have been used, discarding packet content.

Clustering Algorithms can be used for detecting anomalies in a sequence of packets by applying them to a rolling window of features: some authors proposed the use of a SOM to detect attacks in the DARPA dataset, by applying it to connection data, with 6 characteristics for each connection [120]; others even used a SOM to analyze network traffic, discarding the payload and putting the header information in a rolling window. The prototype, called NSOM, can detect denial of service attacks [121]. Other authors propose instead to explicitly use time as a

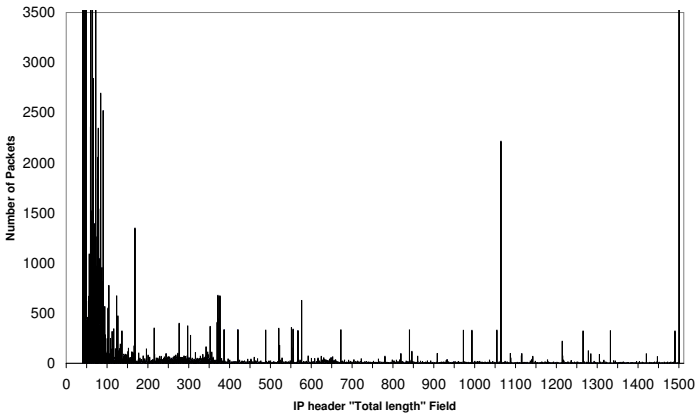


Figure 3.1: Distribution of the values of field Total Length in a portion of the IDEVAL dataset

feature, and to show the packets to a SOM one at a time [122]: this approach is theoretically flawed, since adding a linearly increasing feature to the data cannot possibly contribute positively in an algorithm which focuses on dimensionality reduction through bidimensional mapping.

A further attempt to use Self Organizing Maps is INBOUNDS (Integrated Network Based Ohio University Network Detective Service) [123, 124]. Packets are organized by connection, a subset of categorical features is extracted, and normalized on a Gaussian hypothesis. Then further packets are flagged as anomalous if they are too "far" from the best matching unit (see Section 4.3.2). Unluckily, this work has several theoretic weaknesses. Normalizing categorical data makes the features meaningless at best, and in our test even non-categorical data such as packet length have a non-Gaussian distribution (see Figure 3.1). Average length is 138.348721, standard deviation 342.267625, but evidently the distribution is not Gaussian.

MINDS (Minnesota INtrusion Detection System) [125] uses a density-based anomaly detection algorithm (called a Local Outlier Factor, or LOF) to identify outliers in ten-minutes long time windows over network connection traces. The algorithm however is batch and cannot run in real-time.

Discounting learning algorithms have been used in real time onto

packet header information in the SmartSifter prototype [126], and we will analyze this approach in depth in Section 4.4.3.

A paper which takes into account the payload of the packets, which appeared a year later than our earliest results, shows some interesting statistical properties of packet payload characterization [127]. Also in [128] a monitor stack of Self Organizing Maps is proposed, and in some of the layers payload bytes are considered. However, this experiment is not really meaningful in our context, having been performed over some tens of packets at most, and with a set of features which is appropriately chosen to trigger the alerts on the attacks used for evaluation. In [129] a rule-based algorithm is presented, which takes into account payloads but ignores totally the meaning of the header fields.

We will analyze some of these approaches (the most interesting ones in our opinion) in further detail in Section 4.4.1.

3.6 Evaluation of Intrusion Detection Systems

Evaluation of an intrusion detection system is a difficult and open research topic [54]. We have briefly seen, in Section 3.3, that many different issues plague intrusion detection systems. It is very difficult to plan tests for many of these issues, and even more difficult to combine these tests in a meaningful, overall evaluation.

The simplest evaluation of intrusion detection systems deals with the following quantities:

TP True Positives, alerts raised for real intrusion attempts;

FP False Positives, alerts raised on non-intrusive behaviors;

TN True Negatives, no alerts raised and no intrusion attempts present;

FN False Negatives, no alerts raised when real intrusion attempts present.

False positives are the bane of intrusion detection systems, because after a while an error-prone system is just ignored and not used anymore. Anomaly detection systems are particularly prone to false positives, while signature based systems usually do not have a lot of false positives. They rather have non-contextual alerts, which means true positives on attacks that are nevertheless useless, since they are targeting a non vulnerable platform. The so called target-based architectures [130] try to reduce this problem.

False negatives are obviously also a problem. In particular, for misuse based systems, most new attacks will generate false negatives, unless they are very similar to an existing attack.

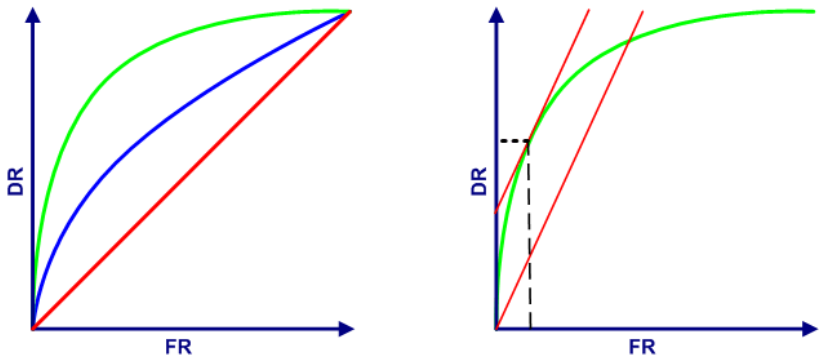


Figure 3.2: Examples of ROC curves

We can then define two important metrics: the Detection Rate, measuring how many attacks are detected overall:

$$DR = \frac{TP}{TP + FN}$$

and the False Positive Rate, measuring how many alerts are false:

$$FR = \frac{FP}{TN + FP}$$

It is easy to see that the DR is equivalent to the "recall" rate in information retrieval systems, while the FP rate is somehow the inverse of the concept of "precision".

Intuitively, these two variables are bound by a trade off: the more sensitive a system is, the more false positives it generates, but the higher the detection rate is. If sensitivity is a variable, s, then $FR = FR(s)$ and $DR = DR(s)$. Therefore we can represent the two quantities as a parametric curve, which is named ROC, Receiver Operating Characteristic, in radar and signal analysis literature.

In Figure 3.2 we have traced some imaginary ROC curves. Axes are obviously scaled from 0 to 1. An IDS which does not generate any alert has $DR = 0$ and $FR = 0$ since $TP = FP = 0$, while an IDS which flags anything has $DR = 1$ and $FR = 1$ (since $FN = TN = 0$). Among these extremes, any behavior can happen.

In general ROC curves are monotonous non decreasing and above the bisectrix. Intuitively, the larger the area below the curve, the better the detection to false alert ratio is. But this definition is scarcely operative (needing a point-by-point analysis to trace and interpolate the curve). Additionally, this global dominance criterion is not always valid: the costs we associate to a false positive or a false negative is generally

different, and subjective (depending on the network size, number of analysts, and so on). Let us call α the cost of a false positive, and β the cost of a false negative, and p the ratio of positive events on the total ($\frac{FP+TP}{FP+TP+FN+TN}$). We can write the cost function:

$$C = FR\alpha(1-p) + (1-DR)\beta p$$

The gradient of this cost function is a line with coefficient:

$$\frac{\alpha(1-p)}{\beta p}$$

If we trace it in the ROC diagram (Figure 3.2 on the right), we can determine the minimum cost point on the ROC curve, and thus a satisfying sensitivity value. In this locality, an algorithm that globally performs worse could still be a better choice.

4 Network Intrusion Detection Applications

4.1 Network Intrusion Detection Problem Statement

The problem of network intrusion detection can be reformulated, in the unsupervised learning framework, as the following: we wish to detect anomalies in the flow of packets, or in the flow of connections, on a TCP/IP network.

There is a number of points to consider:

- Any IP packet has a variable dimension, which on an Ethernet network ranges between 20 and 1500 bytes.
- The first 20 bytes (but the number is not fixed) constitute the IP header and the meaning of each byte and bit of the header is fully described by the Internet standard; thus we can extract a number of "features" from the IP header.
- Another sequence of up to 20 bytes is the header of the transport protocol (such as TCP, UDP, ICMP or others). The same consideration as above applies, with the exception that in many transport protocols correlation between different packets is required to fully understand the headers and their meaning.
- The data included in the payload is both heterogeneous and of varying length: we could decode upper layer protocols such as HTTP, FTP and so on, but this would require full session reconstruction, and anyway it would be difficult to represent them as simple features.
- Even if looking at protocols that do not need correlation and session reconstruction, in order to understand what is happening we need to correlate and track relations among different packets over a time window.

In particular, the varying size of the payload data, and its heterogeneous nature which defies a compact representation as a single feature is the single hardest problem to solve. As we have seen, most existing researches on the use of unsupervised learning algorithms for network intrusion detection avoid this problem altogether by discarding the payload and retaining only the information in the packet header [117, 119, 121, 131, 132].

Ignoring the payload of packets, however, inevitably leads to information loss: most attacks, in fact, are detectable only by analyzing the payload of a packet, not the headers alone. Despite their reduced coverage, these algorithms show interesting, albeit obviously limited, intrusion detection properties. In section 4.4 we will analyze these earlier attempts in depth, with their points of strength and their shortcomings.

Some earlier works tried to deal with this problem, e.g. [129] uses a rule-based algorithm to evaluate the payloads but, on the contrary, ignores totally the meaning of the header fields; ALAD [133] detects "keywords" in the protocols in a rather limited manner; PAYL [127] uses statistical techniques on the payloads, ignoring the headers.

4.2 A two-tier architecture for Intrusion Detection

We propose a novel architecture for building a network based anomaly detection IDS, using only unsupervised learning algorithms, and capable of handling also the content of the payload of network packets (it has been described originally in [134]).

It was our strong belief, based on the consideration that most attacks show up only in the payload of the packets and not in the headers, that the information loss generated by discarding the payload was unacceptable. So we focused on how to retain some of the information contained in the payload, while keeping the problem tractable.

In order to solve this problem, we developed the concept of a two-tier architecture (shown in Figure 4.1), which allows us to retain at least part of the information related to the payload content. Our work hypothesis was that on most networks, the traffic would belong to a small number of services and protocols, regularly used, and so that most of it would belong to a relatively small number of classes.

In the first tier of the system, an unsupervised clustering algorithm operates a basic form of pattern recognition on the payload of the packets, observing one packet payload at a time and "compressing" it into a byte of information (a "payload class" value). This classification can then be added to the information decoded from the packet header (or

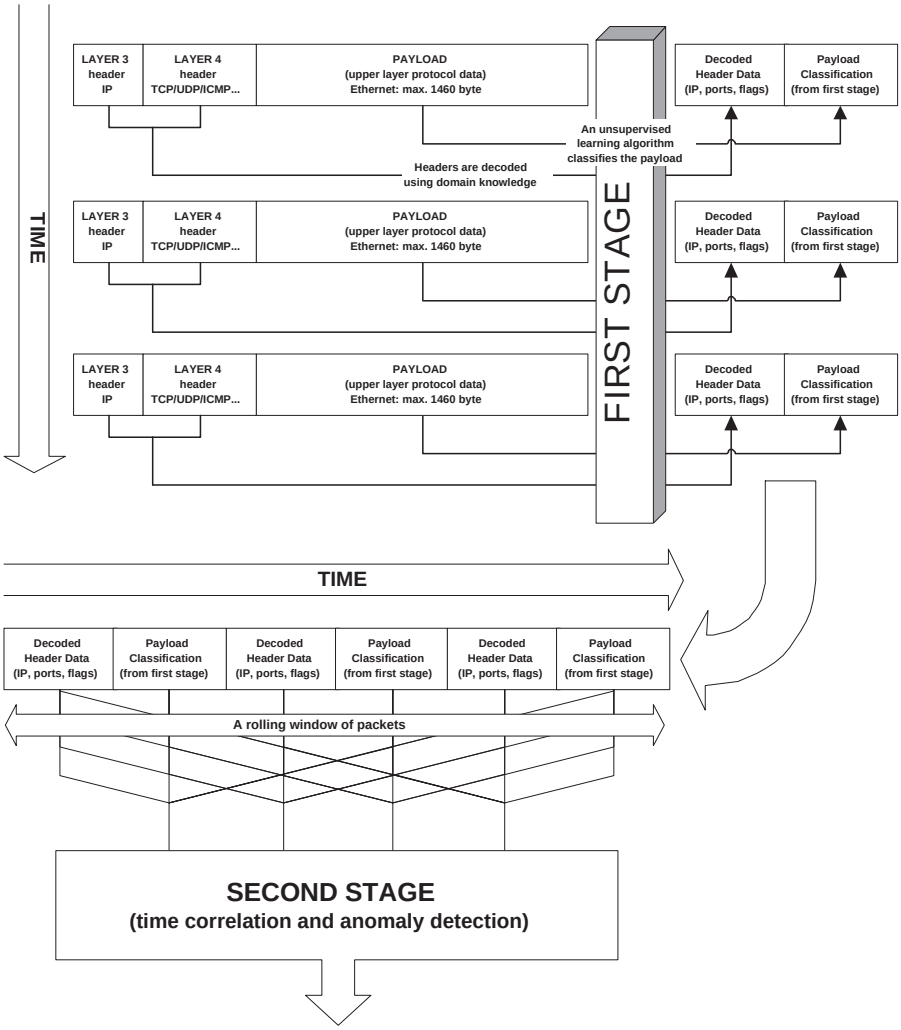


Figure 4.1: Scheme of the overall architecture of the network based IDS

to a subset of this information), and passed on to the second tier.

The second tier algorithm instead takes into consideration the anomalies, both in each single packet and in a sequence of packets. It is worth noting that most of the solutions proposed by previous researchers in order to analyze the sequence of data extracted by the packet headers could be used as a second tier algorithm, complemented by our first tier of unsupervised pattern recognition and clustering.

4.3 Payload Clustering Techniques

4.3.1 Requirements and algorithms selection

In the first tier we need to find an algorithm that receives as an input the payload of a TCP packet[1]. On an Ethernet segment this means up to 1460 byte values which can be interpreted as *pattern vectors* of variable size.

This algorithm must classify these vectors in a *sensible* way. TCP and UDP are used to carry the data of an high number of upper layer protocols. Relying on domain knowledge to pre-divide traffic by type and decode it is not realistically feasible, and would leave us with the same problem on how to handle the contents of each communication.

Additionally, we want to create an independent source of information, based on the *content* of the payload. Using, for instance, the destination port as a trusted information source, we would be assuming that each port is actually used for its proper communication protocol. Thus a connection which for example gives the output of a shell command over a HTTP communication channel would not be detected an anomalous connection.

The algorithm must thus be able to handle these heterogeneous payloads belonging to different protocols, and classify them in a "sensible" way. By sensible we mean that the transformation should exhibit three important properties:

1. It should preserve as much information as possible about the "similarity" between packets; in the following we will better define the concept of "similarity".

[1] In the following we will use TCP as an example, but our reasonings can be easily generalized to UDP packets. ICMP packets, on the other hand, are less of a problem, since the simple decoding of the protocol header is usually sufficient. Some covert communication protocols use the payload of an ICMP packet, but this can safely be ignored for simplicity at this level of analysis.

2. It should separate, as much as possible, packets from different protocols in different groups; previous researches have shown that, for instance, neural algorithms can recognize protocols automatically [135].

3. Most importantly, since our final goal is to detect intrusions, the classification should also separate, as much as possible, anomalous or malformed payloads from normal payloads.

This is a typical problem of clustering, even if it can also be seen as an instance of a pattern recognition problem, where we are trying to characterize the recurring patterns in packet payloads in order to detect anomalies [136]. A classic definition of clustering is:

Definition 4.1 *Clustering is the grouping of similar objects from a given set of inputs [137].*

Another is:

Definition 4.2 *A clustering algorithm is an algorithm by which objects are grouped in classes, so that intra-class similarity is maximized and inter-class similarity is minimized [138].*

Clustering problems look deceivingly simple. In order to approach a a clustering problem we must decide a measure of similarity between elements, and also an efficient algorithm to find an acceptable solution, since finding the "optimal" solution (which maximizes both the intra-class similarity and minimizing inter-class similarity) is an NP-hard problem.

There is an endless variety of algorithms designed to solve this problem. Choosing the correct algorithm for a particular problem is often difficult, but we can identify some of the properties we need for our particular problem. Firstly, many algorithms need a criterion to define a correct or acceptable number of classes, while some others are capable of automatically discovering a suitable number directly from the data, and others are quite tolerant to an arbitrarily high choice. Secondly, some algorithms are better than others when dealing with the presence of outliers in training data. An outlier is classically defined as follows:

Definition 4.3 *An outlier is an observation that deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism. [139]*

We studied many different clustering algorithms (a comprehensive review of which can be found in [140]), and we implemented three representative and widely used approach: the K-means algorithm, which is

a centroid-based approach; the Principal Direction Divisive Partitioning (PDDP), a hierarchical divisive approach [141]; and Kohonen's Self Organizing Maps algorithm [142], which is a competitive, hard neural approach.

Our results [134] show that the SOM algorithm is indeed able to sensibly cluster payload data, discovering interesting information in an unsupervised manner. Additionally, the SOM algorithm is robust with regard to the choice of the number of clusters, and it is also resistant to the presence of outliers in the training data, which is a desirable property. In addition, we have shown that the SOM had the best performance trade-off between speed and classification quality.

It is important to discuss how we can evaluate classification quality at this stage. There are four main criteria to evaluate an unsupervised classification:

- Inspection-based: by manually inspecting the classification and checking if it "makes sense" to us.
- Expert-based: by letting an expert manually classify the same data and see if the results are comparable.
- Metrics-based: using an inner quality criterion such as the ratio of the average cluster radius to the inter-cluster distance.
- Task-based: by evaluating the algorithm against the result of the task it is trying to accomplish; in our case this means evaluating the performance of the first tier using the performance of the complete architecture as a criterion.

While the task-based criterion is appealing, we needed some preliminary criteria to evaluate the algorithms without training a fully functional architecture. Expert classification is not an option for large datasets; so we resorted to manual inspection and "proof of concept" tasks for a first evaluation of the quality of the classification.

In a first experiment, we considered how these algorithms classify two sets of about 2000 packets: the first representing normal traffic, the second being the dump of a vulnerability scan with the "Nessus" tool (`www.nessus.org`). Nessus generates a huge volume of anomalous traffic, composed of attacks and scans. In the following, the histograms represent the number of packets (on y-axis) present in each cluster (on x-axis). Please remind that for graphical reasons the number of packets on y-axis may be differently scaled in the various pictures.

In particular, in Figure 4.2 we present the results of a 10×10 Self Organizing Map (which therefore creates a division of the data in 100

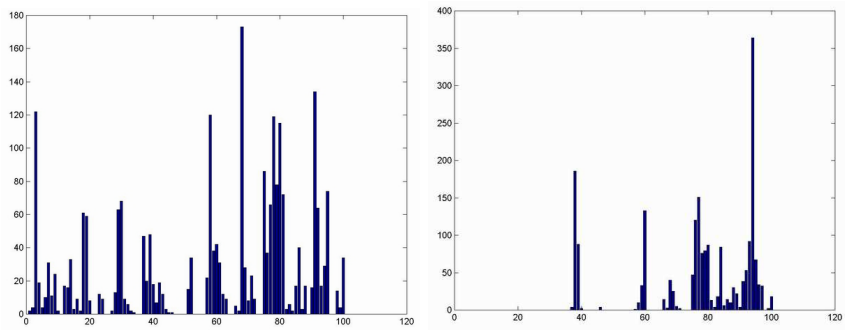


Figure 4.2: Comparison between the classification of normal traffic (above) and Nessus traffic (below) by a 10x10 SOM network

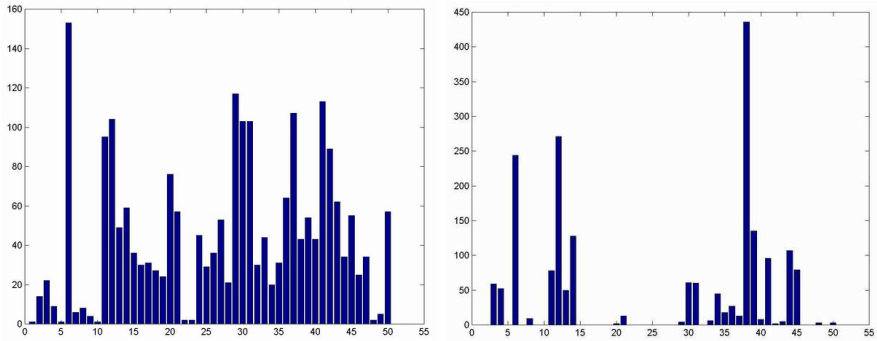


Figure 4.3: Comparison between the classification of normal traffic (above) and Nessus traffic (below) over 50 classes by a principal direction algorithm

clusters). The network was trained for 10.000 epochs with a representative subset of normal traffic. As you can see the difference in the distribution of packets is noticeable. Manual inspection proves that most of the resulting clusters made "sense", which means that the packets falling in the same classes were either the same type of files, or the same portions of protocols (i.e. all the e-mail traffic fell into a narrow group of classes; all the FTP commands fell into another group of classes ...).

We noted an extreme inefficiency in Matlab 6 built-in SOM algorithms. A network training time is about linear in the product of the map dimensions (i.e. for a $n \cdot m$ network, the time is about $O(n \cdot m)$). The training time is linear in the number of epoch, $O(n)$. But, strangely, in the Neural Network Toolbox implementation of Kohonen's algorithms the training time is not linear in the number of items in the training set, nor in the number of dimensions of the vectors: it grows linearly

in both dimension and cardinality until it exhausts system resources, but afterwards the I/O costs make it explode exponentially. The SOM Toolbox (`http://www.cis.hut.fi/projects/somtoolbox/`) showed a much better behavior, but still with some inefficiencies. Therefore, we resorted to creating our own C implementation of the algorithm.

In Figure 4.3 we present instead the results of a division in 50 classes operated by the principal direction divisive partitioning algorithm, in the same experimental conditions used for the SOM. We can see that also in this case the distribution of packets varies wildly between normal and Nessus traffic. The manual inspection also confirms the impression of a sensible classification.

It is worth noting that using the PDDP algorithm poses an additional problem. At each step of the algorithm we must choose the cluster which is going to be split. We would like, obviously, to choose the most "scattered" leaf of the hierarchical divisive tree. Various ways to define the scattering of a leaf have been studied in [143], but for our implementation we chose the simplest (a measure of variance). Other variants could certainly be experimented, and maybe lead to better results.

The computational cost of the PDDP algorithm during training is critical, because it happens that the first step of the algorithm is the most costly (since the training set is split at each step). Normally, for computing the Principal Direction, Matlab uses a SVD (Singular Value Decomposition) algorithm with a time complexity $O(p \cdot q^2 + p^2 \cdot q + q^3)$, where p and q denote the dimensions of the matrix (which are the cardinality and the dimensionality of the training set). In our case, this is way too heavy, so we used an efficient implementation of the Lanczos algorithm, with bidiagonalization and partial reorthogonalization, which offers a complexity of $O(p{\cdot}q{\cdot}r2)$, where r is the rank of the matrix and so $r = min\{p, q\}$ [144]. However, even this algorithm slows down (mainly for memory constraints) as the cardinality of the training set grows. This would be a problem in real-world applications: a spin-off of our research dealt with the creation of an updating algorithm for computing an approximate PDDP progressively [145].

In Figure 4.4, finally, we see that the K-means algorithm does not behave as well as the other two algorithms. Aside from the distribution of traffic which is not as distinct, manual inspection reveals that K-means clusters are less significant. In addition, the random initialization of the algorithm makes the quality of the final result unpredictable, since it converges rapidly to a local (not global) minimum in the distribution of the centroids.

K-means is the fastest of all the algorithm we tested, showing no performance problems even in the training phase; however, the corrections

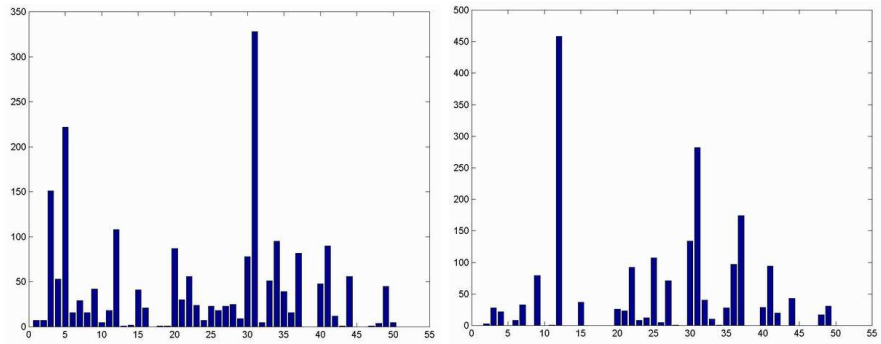


Figure 4.4: Comparison between the classification of normal traffic (above) and Nessus traffic (below) over 50 classes by a K-means algorithm

necessary to eliminate or reduce the random initialization weakness (for instance using the so-called "global K-means" algorithm [146]) make the algorithm intractable. There is, however, a divisive variant of the K-means algorithm, which is compared to the PDDP algorithm in [147] and which could solve the locality problem, still with a cost in performance.

Thus, overall, the SOM algorithm works best, closely followed by the PDDP algorithm which is hampered by its performance problems. K-means is fast, but unreliable: since the SOM algorithm is fundamentally as fast as K-means at runtime, even if slower in training, it is the best overall choice for our first tier algorithm.

4.3.2 An introduction to Self Organizing Maps

Formally [142] Self Organizing Maps are used in order to map, in an unsupervised, ordered but non linear manner, high-dimensional data over a so-called "map" space, usually bidimensional, composed of discrete units called *neurons*. Thus, a SOM compresses the information contained in a high-dimensional input stream (*input space* or Δ) in a bi-dimensional discrete output (*node space* or Γ), converting the nonlinear relationships between data in geometrical relations between the points in Γ. SOMs are used in various pattern recognition tasks such as voice recognition or image analysis.

Neurons have a fixed position in the bidimensional neuron space Γ, usually organized in a rectangular or hexagonal mesh (see Figure 4.5). These positions do not change during training. Each neuron also has a position in the k dimensional input space Δ. Usually, the position of a

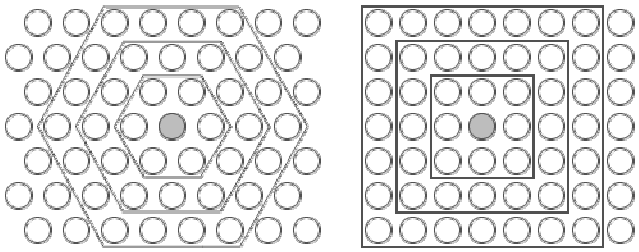


Figure 4.5: Two variants of neuron meshes in Γ

neuron in Δ is indicated as its weights, while its position in Γ is called its coordinates.

Once the dimensions and the type of the neuron mesh are entered, the neuron positions in Γ are automatically fixed, while their weights in Δ must be initialized. This initialization can be random in each of the k dimensions, or linear, trying to uniformly cover the k dimensional domain. This is obviously difficult to do if k is much higher than the number of neurons.

The training of a SOM is both *competitive* and *cooperative*. Each input vector is compared with the weights of all neurons in Δ, and the best matching unit (BMU) is chosen (competitive element). The weights of the BMU are then adjusted to better match the input. The neighboring neurons in Γ are also adapted to the input (cooperative element).

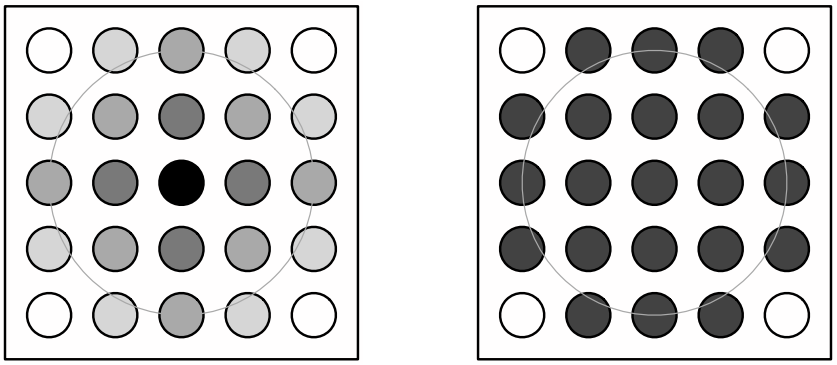
The training process happens in epochs: in each epoch, all the vectors in the training set are shown once to the network. There are two variants of the SOM training algorithm: sequential or batch.

In the sequential variant, neuron weights are adjusted after each input vector, using training function:

$$\vec{m_i}(t+1) = \vec{m_i}(t) + \alpha(t)h_{ci}(t) \cdot (\vec{x}(t) - \vec{m_i}(t)) \; \forall i \in N.$$

where t is the current training iteration, N is the set of all neurons on the map, $\alpha(t) \in [0, 1]$ is the current learning rate (a nonincreasing function of t), $\vec{x}(t)$ is the input of the t-th iteration of the learning process, $\vec{m}_i(t)$ is the vector of the weights of the i-th neuron at iteration t. Neurons are chosen randomly from the dataset, or shown in order.

$h_{ci}(t)$ is the proximity function between the BMU c (therefore $c = argmin_k(\delta(\vec{x}(t), m_k))$ $k \in N$) and neuron i: it is a function of the distance between c and i in Γ, and of the proximity radius at iteration t, $\rho(t)$. $\rho(t)$ defines the maximum distance for two neurons in Γ in order to be considered neighbors; it is also a nonincreasing function of t which starts from a value $\rho(0) \geq 1$ and decreases to 1 during training.



(a) Gaussian Proximity Function (b) Bubble Proximity Function

Figure 4.6: Visual representation of different proximity functions: the darker the color, the higher the adaptation factor

Generally, two alternative proximity functions can be used: a bubble function or a Gaussian function. A bubble function has the form:

$$h_{ci}(t) = \begin{cases} 1 & \forall i \in N : \gamma_{c,i} \leq \rho_c(t) \\ 0 & \text{otherwise} \end{cases} .$$

In this case, all the neurons inside the neighborhood are modified by a same quantity, while the ones outside the "bubble" stay fixed (see Figure 4.6(b)). A Gaussian function can be formulated as follows:

$$h_{ci}(t) = e^{-p} \; \forall i \in N$$

where

$$p = \frac{\gamma_{c,i}}{2\rho_c^2(t)}.$$

As can be seen, in this case the weights of all the neurons on the map are modified in an exponentially decreasing quantity depending on distance from the winning neuron (see Figure 4.6(a)).

Batch training uses instead the whole training data set at once at each epoch. The input dataset is partitioned according to the Voronoi regions created by each neuron, based on its weights in Δ. The new weights are then calculated as:

$$\vec{m}_i(t+1) = \frac{\sum_{j=1}^{n} h_{ci}(t)\vec{x}_j}{\sum_{j=1}^{n} h_{c,i}(t)},$$

where n is the total number of inputs in the training dataset, $\vec{x}_j$ is the j-th vector in the dataset,

Alternatively we can calculate the sum of vectors in each Voronoi region:

$$\vec{s}_i(t) = \sum_{j=1}^{n_{V_i}} \vec{x}_j \ \forall i \in N,$$

where we denote with n_{V_i} the number of vectors in the Voronoi region i. Then, the weights of each neuron can be adapted using the formula:

$$\vec{m}_i(t+1) = \frac{\sum_{j=1}^{l} h_{i,j}(t)\vec{s}_j(t)}{\sum_{j=1}^{l} n_{V_j} h_{i,j}(t)},$$

where $l = |N|$ is the number of neurons on the map.

The training is composed of two consequential phases: ordering and tuning. The difference among these two phases is in the learning rate and neighborhood extension.

The ordering phase lasts for a given number of epochs, usually a few thousands. The neighborhood distance starts as the maximum distance between two neurons on the whole map, and it decreases to the initial tuning neighborhood distance (which is a parameter). The learning rate starts at the initial ordering phase learning rate, and decreases until it reaches the initial tuning phase learning rate (these are parameters, too). As the neighborhood distance and learning rate decrease over this phase, the neurons of the network typically order themselves in the input space with the same topology in which they are ordered in the map space.

The tuning phase lasts for all the rest of the training, and is usually much longer and slower than the ordering phase. The neighborhood distance is fixed throughout the tuning phase to a very small value (e.g., typically 1). The learning rate continues to decrease from the initial tuning phase learning rate, but very slowly. The small neighborhood and slowly decreasing learning rate fine tune the network, while keeping stable the ordering learned in the previous phase.

The result is that the neurons' weights initially take large steps toward the area(s) of input space where input vectors are located. Then, as the neighborhood size decreases to 1, the map tends to order itself topologically over the input vectors. Once the neighborhood size reaches 1 and the ordering phase ends, the network should already be well ordered. During tuning, the learning rate is slowly decreased over a longer period to give the neurons time to spread out evenly across the input vectors. A variant of the algorithm uses a stop criterion for the tuning phase, halting the algorithm when training converges (for instance, when weights are almost unchanged from one epoch to the following).

The final result is that the neurons of a self-organizing map will order themselves with approximately equal distances between them if input vectors are evenly distributed throughout a section of the input space; if instead input vectors occur with varying frequency throughout the input space, the SOM tends to allocate more neurons in the more crowded areas. Thus the SOM learns how to categorize inputs by learning both their topology and their distribution.

In the recognition/runtime phase, each input vector is simply compared against the weights of each neuron in Δ, making the SOM work in a very similar way to a traditional K-means algorithm.

4.3.3 Using a SOM on high-dimensional data

As it is known, the computational complexity of unsupervised learning algorithms scales up steeply with the number of considered features, and the detection capabilities decrease correspondingly (this is usually called the "curse of dimensionality"). This effect hits heavily against the first tier of our system, which receives up to 1460 bytes of data. A few algorithms can be optimized to treat data with many thousands of dimensions, but only in the case that they are sparse (for instance, a word/document incidence matrix in a document classification and retrieval problem [148]), but we are dealing with dense data. There are alternative algorithms for clustering which are much faster in the learning phase than SOM, for example, the well known K-means algorithm is one of the fastest. But during recognition even K-means is not more efficient than a SOM, so we cannot solve this problem by simply choosing a different algorithm.

A traditional approach to the problem would use dimension reduction techniques such as dimension scaling algorithms [149] or Principal Component Analysis [150]. But our early experiments demonstrated that such techniques are quite ineffective in this particular situation, since by their nature they tend to "compress" outliers onto normal data, which is exactly the opposite of what we want to achieve.

Since no alternative solution was viable, we developed various approximate techniques to speed up the SOM algorithm [151]. The reference machine for our tests is an Athlon-XP 3200 based computer with 1 GB of DDR RAM, running GNU/Linux with a 2.6 kernel. All the tests, unless otherwise stated, refer to a SOM with square topology, and a size in the space of neurons of 10×10. The test are conducted on TCP packets, as they constitute over 85% of Internet traffic.

As we can see from the first line of values in table 4.1, the throughput of a straightforward C implementation of the Kohonen algorithm on our

hardware and software configuration is on average of 3400 packets per second, which is enough to handle a 10 Mb/s Ethernet network, but insufficient for a 100 MB/s network.

Thus, we developed some heuristics for speeding up the computation, introducing minimal errors in the classification. The idea behind our heuristic is simple. Let N be the number of classes, and d the number of dimensions of the data. At runtime, the Self Organizing Map algorithm consists simply of N evaluations of the distance function: in our test implementation, an euclidean distance function over d dimensions. Since the number of computations is $N \cdot d$, in order to speed up the computation we can try to reduce d by applying any dimensionality reduction technique: this, as we said before, cannot be done meaningfully via dimensionality reduction techniques. However, since just a few packets contain a high number of bytes of payload, we can try to use just the first $d' < d$ dimensions. Further experimental evaluation would then of course be required in order to understand if the "reduced" payloads carry the same information value as the complete packets.

If we do not want to reduce d, we must try to reduce the number of evaluations N. A smart way to do this is to pre-compute a grouping of the $N = 100$ centroids of the classes in $K < N$ super-clusters, and then select the winning neuron in a two-step procedure. First, we determine which of the super-clusters the observation belongs to; and then we evaluate the distance function just over the $N' < N$ neurons belonging to the winning super-cluster. The algorithm is heuristic, since it can happen that the best matching neuron is not in the best matching super-cluster, but as we will see the error rate is very low. Obviously the best performance gain with this heuristic happens if each of the K super-clusters is formed by $\sim N/K$ neurons, since the average number of computations becomes $d \cdot (K + N/K)$ which has a minimum for $K = \sqrt{N}$. If the clusters are not balanced then in the worst case the computational cost is higher, and this leads to a lower overall throughput. For smaller values of K the algorithm would be on average slower, and the error rate statistically would be slightly lower.

To form the super-clusters, a first naïve idea would be to exploit the map structure, which tends to keep "close" to each other the neurons which are close in the map space. However, this does not work very well experimentally, probably because of the high dimensionality of the feature space, causing a 35% error rate with $N = 3$, and even 60% with $N = 10$. Thus we resort to a K-means approach.

However, we must overcome two different issues in doing this. A first issue has to do with the nature of K-means, which is inherently initialization dependent, and prone to create very unbalanced clusters.

Experimentally, with $N = 100$, using $K \geq 4$ does not create a balanced structure of clusters, unless we correct the randomness of the algorithm. Some authors proposed, in order to eliminate these weaknesses, the "global K-means" algorithm [146], which repeats K-means with all the possible initializations. We use a different and faster approach, by using the algorithm a fixed number m of times, and choosing the distribution in classes which minimizes the average expected number of operations, roughly approximating the probability that an observation falls into the i-th super-cluster as proportional to the fraction N_i/N (where N_i is the number of neurons in the i-th super-cluster). In Table 4.1 we refer to our variant of the K-means algorithm as "K-means+", and the column labeled "Crossv." reports the parameter m (number of runs of the K-means algorithm).

A second, more difficult issue, is how to deal with the training phase. During the training phase the neurons change their position, so theoretically we should repeat the K-means algorithm once for each training step. We can avoid to do so, and fix an arbitrary update frequency, a number of step after which we will recalculate the position of the centroid. As an additional attempt to reduce the cost of the K-means step, we decided to initialize the position of the K centroids to the same position they held before, even if this could lead the convergence to a local optimum, creating a non-optimal clustering. Our tests showed that in each case the cumulative approximations introduced by the algorithm make the training very unstable, leading to results which are not compatible with the ones obtained by normal training, and in which the properties of outlier resilience and robustness of the SOM algorithm are impaired. We are working to find a way to overcome these issues without sacrificing the throughput gain, but for now, our heuristic cannot be applied and the only reliable way to speedup the training phase is to lower the number of dimensions.

In Table 4.1 we report the runtime throughput of the algorithm, evaluated in packets per second, depending on different combination of the parameters.

In order to evaluate the results, we refer to a well known study of the statistical properties of Internet traffic [152]. Analyzing the traffic flowing through an Internet Exchange data-center, they show that approximately 85% of the traffic is constituted by TCP packets, and that a large proportion of TCP packets are 40 bytes long acknowledgments which carry no payload (30% to 40% of the total TCP traffic). Zero-size UDP packets, on the contrary, are almost non-existant. Since the first tier analyzes only packets with a non-null payload, almost 30% of the total traffic on the wire will not even enter it. The average size of a

Bytes	Heuristics	K	Crossv.	Packets/sec.	Error %
1460	None	-	-	3464.81	-
1460	K-means	10	No	8724.65	0.8
1460	K-means+	5	10	5485.95	0.4
1460	K-means+	10	10	10649.20	0.8
800	None	-	-	4764.11	-
800	K-means+	5	10	9528.26	0.5
800	K-means+	10	10	15407.36	1.0
400	None	-	-	8400.45	-
400	K-means+	5	10	28965.84	0.6
400	K-means+	10	10	30172.65	1.2
200	None	-	-	10494.87	-
200	K-means+	5	10	51724.70	0.8
200	K-means+	10	10	65831.45	2.3

Table 4.1: Throughput and errors during runtime phase, calculated over a window of 1.000.000 packets. The values are averages over multiple runs of the algorithm on different portions of the dataset

TCP packet is 471 bytes, of a UDP packet 157, and the overall average is approximately 420 bytes. It is also known from theoretical modeling and practical experience that an Ethernet network offers approximately 2/3 of its nominal capacity as its peak capacity. This means that a saturated 10 Mbps Ethernet LAN carries about 2.000 packets per second. Other statistics suggest that this value could be higher, up to 2.500 pps.

From Table 4.1, we can see that the original SOM algorithm, considering the full payload of 1460 maximum bytes per packet, with no heuristics, operates at a speed that is acceptable for use on a 10 Mb/s Ethernet network, but insufficient for a 100 MB/s network. However, using the K-means algorithm with 10 classes and no cross-validation, we obtain a much higher throughput (more than three times higher than the original one) but also a 0.7% error rate. Introducing K-means+ and crossvalidation, we obtain a better tradeoff between throughput and error rate, improving the former without compromising the latter. A speed of 10.500 packets/second is enough to handle a normal 100 Mbps link (considering also the presence of empty packets). If necessary, performance could also be improved by reducing the number of bytes of the payload.

It can be also shown that the use of our modified algorithm does not diminish the detection capabilities of the system. For better clarity, we

will demonstrate this alongside with the overall evaluation of the system.

4.3.4 Meaningful metrics in high-dimensional spaces

When using similarity-based algorithms, obviously the choice of a similarity criterion is of uttermost importance. Given that, theoretically, there is no indication of a good criterion for our particular field of interest (because ours is, as of our knowledge, the first attempt to characterize packet payloads by the means of clustering algorithms), we have observed similar problems in different fields in literature.

The two most used distance criteria in SOM literature are the inner product and the euclidean metric. Since the inner product is closely related to the so-called cosine distance, it is particularly useful in those cases where attributes have values whose characteristic is to be either zero or nonzero. We have a range of discrete values with different meanings instead, so we resorted to the euclidean distance. While this choice has no theoretical support in a problem like ours, our experiments have shown that it works well. More work could be done to study other, maybe better suited, distance functions, for instance lexical distances, matching percentage, or similars. The problem is that introducing such non-metric distances would require to modify heavily the SOM algorithm, and this would be ground for a deeply interesting theoretical work which is quite outside the scope of this thesis.

In recent researches, however, the effect of the curse of dimensionality on the concept of "distance metrics" has been studied in detail. In high dimensional spaces such as the one we are considering, the data become very sparse. In [153, 154] it is shown that in high dimensional spaces the concept of proximity and distance may not be meaningful, even qualitatively.

Let $Dmax_d$ be the maximum distance of a query point to the points in a d-dimensional dataset, and $Dmin_d$ the minimum distance, and let X_d be the random variable describing the data points. It has been shown, under broad conditions, that if

$$lim_{d\to\infty}\, var\left(\frac{\parallel X_d \parallel}{E[\parallel X_d \parallel]}\right)$$

then

$$\frac{Dmax_d - Dmin_d}{Dmin_d}$$

This means, plainly, that in a high dimensional space the difference between the distance of a query point to the farthest and to the nearest point in the dataset tends to be of a smaller order of magnitude than the

minimum distance: in other words, the nearest neighbor identification becomes unstable and does not give much information.

However, most of the hypotheses of such theoretical works do not hold for our variables. We have experimentally observed that in our setup this effect does not happen: most points are extremely well characterized into dense and compact clusters. In order to better understand if this condition applied to our dataset, we recursively filtered out the most compact clusters and the "farthest" centroids, and analyzed the results, and in each case the difference between $Dmin$ and $Dmax$ was still significant. We thus concluded that the effect observed in the cited articles does not apply to our particular situation, probably because we are working in a compact region where the maximum possible distance between two different points is $\sqrt{255^2 \times 1460}$.

In [154] it was also reported that in high dimensional spaces the L_1 metric, or "Manhattan distance", behaves considerably better than the usual euclidean metric we applied. In [155] distance metrics with a fractional index $f \epsilon (0, 1)$ are also proposed.

We explored the application of these distance metrics and their effects on the classification of packets. However, in our particular application the use of these alternate distance seems to lump all the data in a few cluster, diminishing the overall recognition capabilities of the algorithm instead of enhancing it.

We are currently studying the applicability of wavelet-based distance metrics such as the ones proposed in [156].

4.3.5 Experimental results: Pattern Recognition Capabilities of the First Tier

For repeatability, we used for our experiments the datasets created by the Lincoln Laboratory at M.I.T., also known as "DARPA IDS Evaluation dataset" or IDEVAL dataset. In Section 4.6 we will analyze thoroughly the reasons of this choice.

Since our objective is to add the classification of payloads produced by the first tier as one of the features analyzed by the second tier outlier detector, a precondition is that attack payloads are "classified differently" from normal payloads. As we already noted, this means that the first tier must be able to separate and recognize packets from different protocols, and also that it should separate, as much as possible, anomalous or malformed payloads from normal ones.

In Figure 4.7 we present a demonstration of the recognition capabilities of a 10×10 Self Organizing Map (using our modified algorithm for higher throughput) that creates a division of the data in 100 clusters.

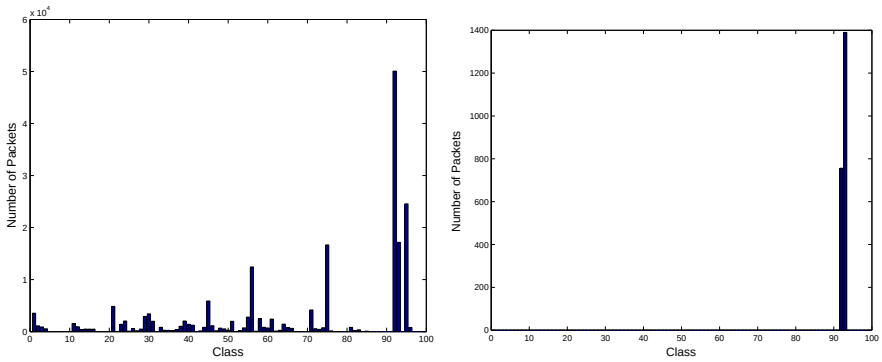


Figure 4.7: Comparison between the classification of a window of traffic and the traffic destined to port 21/TCP by a 10x10 SOM with our modified algorithm.

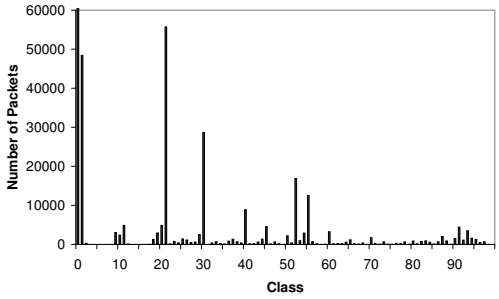
The network was trained for 10.000 epochs on TCP packet payloads. The histograms represent the number of packets (on y-axis) present in each cluster (on x-axis). Here and in the following, for graphical reasons, the number of packets on y-axis may be differently scaled in the various graphs. In Figure 4.7 we suppressed from the output the representation of classes 90 and 93, which are the most crowded and less characterized clusters in the classification, for better display.

On the left side, we can see the classification of a whole window of traffic in the day Thursday, 2nd week of the 1998 DARPA dataset. On the right side, we can see how the network classifies the subset of the packets with the destination port set to 21/TCP (FTP service command channel). It can be observed that all the packets fall in a narrow group of classes, demonstrating a strong, unsupervised characterization of the FTP protocol, which is the first key characteristic we need.

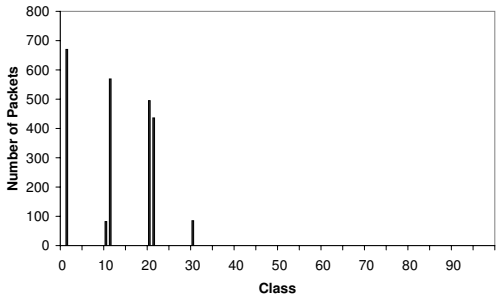
To show that this happens constantly, in Figure 4.8(a) we show the classification of the TCP/IP packets of another whole day (Monday, 2nd week) of the 1999 DARPA dataset, with the above described SOM. Figure 4.8(b) shows how the packets with destination set to port 21/TCP are very well characterized. The same happens for port 80/TCP (HTTP), as shown in Figure 4.8(c). In addition, the two protocols are very different. For the same graphical reasons as above, we have not shown the class of "empty packets".

In Figure 4.9 we can see the plot of the same tests, but with a SOM using our heuristics for added speedup. The histograms are perfectly in line with the result above, with minimal differences in classifications.

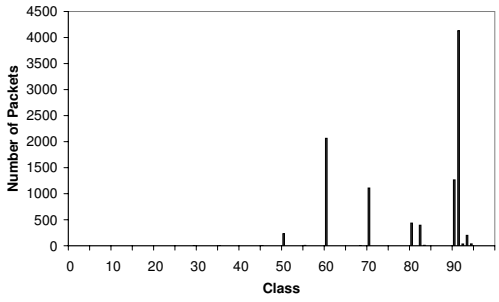
In order to evaluate the recognition capabilities of the new algorithm,



(a) Total

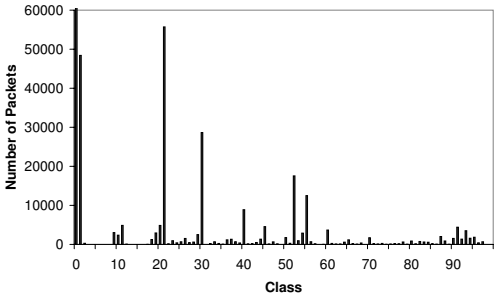


(b) Port 21

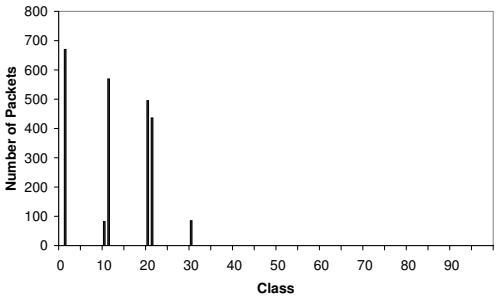


(c) Port 80

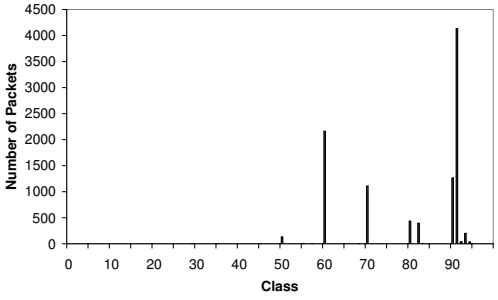
Figure 4.8: Classification of payloads obtained by a non-heuristic SOM, on the whole traffic and on two specific ports



(a) Total



(b) Port 21



(c) Port 80

Figure 4.9: Classification of payloads obtained by a heuristic SOM, on the whole traffic and on two specific ports

we must also see if it can usefully characterize traffic payloads for different protocols, and detect anomalous attack payloads from normal payloads.

We recreated attacks using packet dumps available on the Internet, as well as scripts and attack tools created by ourselves or downloaded from the ExploitTree project [157] or through the MetaSploit Framework [158]. The results are encouraging: for instance, the payload of the format string WU-FTPd bug exploit (CVE-2000-0573) is classified in class 69, which is not one of the usual classes for FTP traffic.

Even more interesting are the results we obtain when analyzing the globbing denial-of-service attack (CERT Advisory CA-2001-07). The attack is polymorph, since the aggressor tries to overload the FTP server by sending a long string of wild-card operators that must be expanded. Any combination can be used, e.g. `LIST */../*/../*/...`, or `LIST */.*/*/.*/*/....` It is difficult to write a good signature for this attack for misuse based IDSs. In order to achieve a generalized match with a signature based system such as Snort we need to write a signature matching `/*`, thus generating a lot of false positives. The SOM classifies all the known variations of the attack in a single class, which does not contain any normal FTP packet.

Another format string attack against Wu-FTPd (CVE-2000-0574) is classified into class 81, which does not contain any normal FTP traffic. This attack uses a long "padding" composed by NOP instructions (`0x90`), for reasons similar to the ones we discussed before in the context of buffer overflow attacks. Most IDSs detect a long sequence of NOP as a possible shellcode, but sneaky attackers use a jump to the following instruction (`0xeb 0x00`) instead of a NOP to fool them. But even if we substitute the NOP codes with `0xeb 0x00` and run the attack again, the system still classifies it into the anomalous class 81.

In Figure 4.7 a 10×10 SOM (with hexagonal topology) has been trained on the TCP packet payloads of the usual dataset. We extracted then the subset of packets with destination port 80/TCP, in a day where various attacks on the same port were present (for graphical reasons, the scale of y-axis is in percentage, not absolute). As it can be seen, the attacks consistently fall outside of the scope of the normal characterization of HTTP protocol. Other cases are similars: let us pick two examples. Firstly: a race condition and buffer overflow bug in the "ps" command, which is exploited over a perfectly legitimate telnet connection. 99.76 % of packets destined to TCP port 23 fall in classes 91 and 95, and all of them fall between class 90 and 95. The packets containing the attack fall instead in classes 45, 54, 55, 65, 71, 73 and 82, which are not normally associated with DPORT 23. This happens consistently over each

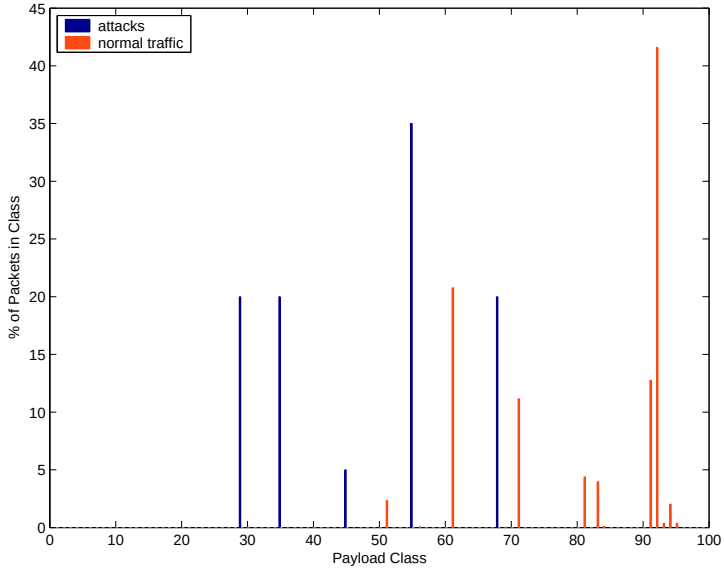


Figure 4.10: A comparison between the classification of attack payloads and normal traffic payloads on port 80/TCP

instance of the attack.

A similar, albeit less defined, situation happens in the case of a buffer overflow in the "sendmail" MTA daemon. The packets destined to port 25 are less characterized, but over 90 % of them fall into 7 classes. The attack packets fall instead into three different classes that contain less than 3% of the normal packets destined to port 25. This helps us to understand that an important requirement for the second tier detection algorithm will be to keep track of anomaly scores in the recent past, and that the second tier cannot be a crisp, rule-based system, but must be a more statistical and fuzzy one.

In a famous article, some years ago, J. Frank, [159], while commenting the future trends of artificial intelligence, pointed at clustering algorithms as a possible future approach to intrusion detection. His early intuition was indeed correct.

4.4 Multivariate Time Series Outlier Detection

4.4.1 Requirements and available algorithms

The second tier algorithm must detect anomalies and outliers over a multivariate time series with at most 30 features. The algorithm should

deal with two problems:

1. Intra-packet correlation: to analyze the content of each packet looking for indicators of anomaly (for instance, a packet belonging to a class which is not usually associated with a particular port);

2. Inter-packet correlation over time: to recognize anomalous distributions of packets (for instance, a sudden burst of packets with normally rare characteristics).

Since it must analyze the correlation among different consecutive vectors of observations, the algorithm must be either endowed with some form of memory, or should observe a rolling window of data.

There is an open trade-off at this point, since enlarging the time window (or, correspondingly, increasing the weight of the memory) for a better correlation could blind the system to atomic attacks, which represent a significant share of network attacks. In fact, most misuse-based network IDS signatures can be applied even to a single packet at a time (except for what we observed in Section 3.3.6, which makes it more convenient to perform stream reassembly). But a statistical system, by its own nature, could be better at detecting significant variations over a long time than single attack packets.

A wide range of algorithms that can be used to detect anomalies in time series exist (a survey of outlier detection techniques can also be found in [160]), but they are mostly limited to continuous variables (we have discrete and categorical values) and to strictly ordered series. Packets are neither totally numeric nor strictly ordered. Missing these characteristic, we cannot use powerful mathematical instruments such as spectral analysis.

Additionally, since in a real world situation it would be difficult to collect a large base of attack-free traffic in order to train the algorithm, we need it to be resistant to the presence of outliers in the training dataset.

Excluding supervised algorithms, we are left with a handful of candidates. A first approach (which we tested in [134]) is to map a time series onto a rolling window of observation, and then use regular clustering techniques for finding outliers. For instance, SOMs have been used in this fashion, [120, 121] on either connection data, or on packet header data (discarding the payload). However, it has also been proposed [161] that this approach is deeply flawed for statistical reasons. Other authors propose instead to explicitly use time as a feature in a clustering algorithm [122]. This is an outright mistake, since time on a network is quite relative (even more than it is already by its own nature), and since

a clustering algorithm such as a SOM cannot handle in any meaningful way a linearly increasing dimension like that.

Instance Based Learning (IBL) is a class of algorithms which represent concepts by the means of a dictionary of "already seen" instances. There are both supervised and unsupervised variants, and an unsupervised one has been proposed for host based intrusion detection purposes [162]. However, it seems that this algorithm works well for problems where the number of instances in the dictionary is quite limited. More studies would be needed to apply this approach to network data.

PHAD, Packet Header Anomaly Detection [117], is a simple statistical modeling method which has been applied to data extracted by packet headers. By using a really simple method (which grants great performances), PHAD detects about half of the attacks in the DARPA 1999 dataset. The algorithm could be easily extended with the classification output by the first tier of our architecture.

NETAD [132] is an evolution of PHAD and LERAD (Learning Rules for Anomaly Detection [131]). NETAD prefilters traffic using various rules (based on protocol type and sequence numbers), and then models nine non-disjunctive subsets of traffic. The first 48 bytes of each packet are taken into account and modeled. Denoting with $A(b, i)$ the anomaly score of the value i for the byte b, NETAD uses the following formula:

$$A(b,i) = \frac{t_b n_b (1 - \frac{r_b}{256})}{r_b} + \frac{t_{b,i}}{f_{i,b} + \frac{r_b}{256}}$$

where n_b is the number of packets since a previously unseen value last appeared in b during training; t_b is the number of packets since the last anomaly was flagged in b during runtime; r_b is the number of different values allowed for b, which is equal to the number of different values observed during training; $t_{b,i}$ is the number of packets since b has last assumed value i; $f_{i,b}$ is the frequency of $b = i$ during training.

The statistical models gives thus an high anomaly rating to values that are either very rare, or that have not happened for a long time. Both PHAD and NETAD have the conspicuous disadvantage that they do not identify intra-packet anomalies, but just inter-packet sequence anomalies. In addition, they require to be trained on attack-free datasets.

MUSCLES (MUlti-SequenCe LEast Squares) is an algorithm based on multivariate linear regression [163] for outlier detection in correlated time series. We describe our experience with MUSCLES in section 4.4.2.

Information theoretic methods such as the Parzen Window method have been proposed in [119]. They have the advantage that being formulated as a statistical hypothesis test uses as a parameter an "acceptable false detection rate" which can be used for tuning, and that they do not

need training. However we experimented their runtime to be unacceptable. A much better alternative, which shares many of the features we have describe, is to use a discounting learning algorithm such as the one used in the SmartSifter prototype [126], which combines the elegance of a statistical approach with a smooth running time. After a lot of testing, we decided that a modified version of SmartSifter was the best approach. We will describe this algorithm in depth in Section 4.4.3

In [164] a framework is proposed for using Hidden Markov Models for modeling multivariate time series. The approach is indeed interesting and novel, but does not seem suitable for high-speed modeling and recognition.

4.4.2 MUSCLES

MUSCLES (MUlti-SequenCe LEast Squares) is an algorithm based on multivariate linear regression [163] for outlier detection in correlated time series.

Let us consider k time series $s_1, \ldots, s_k$. Suppose now that we want to make the best estimate of $s_1[t]$ (let us call it $\hat{s_1}[t]$), given $s_1[t-1], s_1[t-2], \ldots, s_1[t-w]$ and $s_2[t], s_2[t-1], \ldots, s_2[t-w]; s_3[t], s_3[t-1], \ldots, s_3[t-w]; \ldots; s_k[t], s_k[t-1], \ldots, s_k[t-w]$.

We can estimate $\hat{s_1}[t]$ as a linear combination of the values of the signals in a time window of width w. Formally:

$$\begin{aligned} \hat{s_1}[t] &= a_{1,1}s_1[t-1] + \cdots + a_{1,w}s_1[t-w] + a_{2,0}s_2[t] + \\ &\quad a_{2,0}s_2[t] + a_{2,1}s_2[t-1] + \cdots + a_{2,w}s_2[t-w] + \\ &\quad \cdots \\ &\quad a_{k,0}s_k[t] + a_{k,1}s_k[t-1] + \cdots + a_{k,w}s_k[t-w], \\ \forall t = w+1, \ldots, N. \end{aligned} \tag{4.1}$$

Equation 4.1 is a linear equation with $v = k(w+1) - 1$ independent variables. Through linear regression we can compute the set of values for the regression coefficients $a_{i,j}$ in order to minimize the squared errors $\sum_{i=1}^{N}(s_1[t] - \hat{s_1}[t])^2$.

Naively, we could compute the best vector of coefficients $\vec{a}$ as:

$$\vec{a} = (X^T \times X)^{-1} \times (X^T \times \vec{y}) \tag{4.2}$$

where X is the $N \times v$ matrix, where each line j holds the independent variables of Equation 4.1 for $t = j$. However, this equation is extremely inefficient, in terms of spatial complexity ($O(N \times v)$, with an a priori unbounded N), as well as in terms of computational complexity ($O(v^2 \times (v+N))$ for each new incoming observation).

Using the matrix inversion lemma [165] we can obtain a much simpler form of the equation. Let X_n denote matrix X when $N = n$ and let $G_n = (X_n^T \times X_n)^{-1}$. We can obtain G_n from G_{n-1} using the following relation (see [163] for details on the derivation of this equation):

$$G_n = G_{n-1} - (1 + \vec{x}[n] \times G_{n-1} \times \vec{x}[n]^T)^{-1} \times (G_{n-1} \times \vec{x}[n]^T) \times (\vec{x}[n] \times G_{n-1}),\ n > 1 \quad (4.3)$$

where $\vec{x}[n]$ is the line vector of the independent variables value at $t = n$ (i.e. the new input vector). Equation 4.3 spares us to perform a matrix inversion since $1 + \vec{x}[n] \times G_{n-1} \times \vec{x}[n]^T$ is a scalar value; thus the computation is $O(v^2)$. Additionally, we just need to keep in memory G_n, which requires $O(v)$ space (with $v \ll N$). We can also add to Equation 4.3 a forgetting factor $\lambda \in (0, 1]$ to account for slow changes in the correlation laws of the source. If we redefine the problem as minimizing $\sum_{i=1}^{N} \lambda^{N-1}(y[i] - \hat{y}[i])^2$, we obtain the following equations (with $n > 1$):

$$G_n = \frac{1}{\lambda} G_{n-1} - \frac{1}{\lambda}(\lambda + \vec{x}[n] \times G_{n-1} \times \vec{x}[n]^T)^{-1} \times (G_{n-1} \times \vec{x}[n]^T) \times (\vec{x}[n] \times G_{n-1})$$

and

$$\vec{a}_n = \vec{a}_{n-1} - G_n \times \vec{x}[n]^T \times (\vec{x}[n] \times \vec{a}_{n-1} - y[n]).$$

Since we have defined an outlier as a value which is radically different than expected, if we suppose estimation error to be a Gaussian distributed random variable with standard deviation σ, we can label as anomalous any value of $\hat{s_1}$ which differs from s_1 by more than 2σ or 3σ.

Despite its efficiency, this algorithm has the obvious disadvantage that it can be applied only to variables for which a metric concept makes sense, i.e. where average and standard deviation can be computed: this is evidently not our case. The experimental results we obtained with this algorithm are therefore predictably bad.

A different approach, but with similar limitations, is the algorithm SPIRIT (Streaming Pattern dIscoveRy in multIple Time-series) [166]. Given n numerical data streams, whose values are observed in a discretized way, SPIRIT can incrementally find correlations and hidden variables, which summarize the key trends in the entire stream collection. It can do this quickly, with no buffering of stream values and without comparing pairs of streams. Moreover, it is a single pass algorithm, and it dynamically detects changes. The discovered trends can also be used to immediately spot potential anomalies, to do efficient forecasting and, more generally, to dramatically simplify further data processing. Our experimental evaluation and case studies show that SPIRIT can incrementally capture correlations and discover trends, efficiently and effectively, but only on metric data.

4.4.3 SmartSifter

SmartSifter [126, 167, 168] is an unsupervised algorithm for outlier detection in multivariate time series based on discounting learning. It is designed for online usage, and it uses a "forgetting factor" in order to adapt the model to non-stationary data sources. The output of SmartSifter is a value expressing the statistical distance of the new observation, which means "how much" the new observation would modify the model currently learned. SmartSifter has also the great advantage to be able to use both categorical and metric variables. In the following sections we will briefly describe the algorithm, and how we modified it to adapt it to our needs.

SDLE Algorithm: handling categorical variables

The *SDLE* algorithm is used to learn probability densities associated with categorical variables.

Suppose we have n categorical variables. Let $\mathbb{A}^{(i)} = \left\{a_1^{(i)}, \ldots, a_{u_i}^{(i)}\right\}$ $(i = 1, \ldots, n)$ be the domain of the i-th variable. We partition the domain in disjoint sets such that $\left\{\mathbb{A}_1^{(i)}, \ldots, \mathbb{A}_{v_i}^{(i)}\right\}$ $(i = 1, \ldots, n)$ where $\mathbb{A}_j^{(i)} \cap \mathbb{A}_k^{(i)} = \emptyset$ $(j \neq k)$ and $\mathbb{A}^{(i)} = \cup_{j=1}^{v_i} \mathbb{A}_j^{(i)}$. Inside the n-dimensional space we can identify the cell $\mathbb{A}_{j_1}^{(1)} \times \cdots \times \mathbb{A}_{j_n}^{(n)}$ as the j-th cell. The domain is thus partitioned in $k = v_1 \cdot \cdots \cdot v_n$ cells.

The probability density function (or histogram) assumes a constant value for each cell, expressed by $\theta = (q_i, \ldots, q_k)$ where $\sum_{j=1}^{k} q_j = 1$, $q_j \geq 0$ and q_j denotes the probability value for the j-th cell. The learning algorithm makes use of the "forgetting factor" $r_h \in [0, 1]$: the lower this value, the more the algorithm is influenced by past events.

In algorithm 1 we used the following notation:

$$\delta_t(j_1, \ldots, j_n) = \begin{cases} 1 & \text{if } x \in \mathbb{A}_{j_1}^{(1)} \times \cdots \times \mathbb{A}_{j_n}^{(n)} \\ 0 & \text{otherwise} \end{cases}$$

Also, $T_t(j_1, \ldots, j_n)$ represents the number of times that an input falls into cell j. To see the meaning of the forgetting factor let us consider the equation on line 7:

$$q^{(t)}(j_1, \ldots, j_n) := \frac{T_t(j_1, \ldots, j_n) + \beta}{(1 - (1 - r_h)^t)/r_h + k\beta} \tag{4.4}$$

Considering the denominator, we can see in Figure 4.11 the plot of $f(t) = \frac{1-(1-r_h)^t}{r_h}$ with $r_h = 0.2$ and $r_h = 0.9$. We can see that, the

Algorithm 1 The SDLE algorithm

Require: a partitioning of domain $\left\{\mathbb{A}_1^{(i)}, \ldots, \mathbb{A}_{v_i}^{(i)}\right\}$ $(i = 1, \ldots, n)$, r_h and $\beta \in (0, 1)$

1: $T(j_i, \ldots, j_n) \Leftarrow 0 (1 \leq j_i \leq v_i,\ i = 1, \ldots, n)$ {Initialization}
2: $t \Leftarrow 1$ {Parameter updating}
3: **while** $(t \leq T)$ **do**
4: $\quad \vec{x_t} = \texttt{read}(x_1, \ldots, x_n)$
5: $\quad$ **for all** j **do** {for each cell do:}
6: $\quad\quad T_t(j_1, \ldots, j_n) \Leftarrow (1 - r_h) T_{t-1}(j_1, \ldots, j_n) + \delta_t(j_1, \ldots, j_n)$
7: $\quad\quad q^{(t)}(j_1, \ldots, j_n) \Leftarrow \frac{T_t(j_1, \ldots, j_n) + \beta}{(1-(1-r_h)^t)/r_h + k\beta}$
8: $\quad$ **end for**
9: $\quad$ **for all** $\vec{x} \in \mathbb{A}_{j_1}^{(1)} \times \cdots \times \mathbb{A}_{j_n}^{(n)}$ **do**
10: $\quad\quad p^{(t)}(\vec{x}) := \frac{q^{(t)}(j_1, \ldots, j_n)}{|\mathbb{A}_{j_1}^{(1)}| \cdot \ldots \cdot |\mathbb{A}_{j_n}^{(n)}|}$
11: $\quad$ **end for**
12: $\quad t \Leftarrow t + 1$
13: **end while**

larger r_h is, the less t must be in order to influence the learning. In the following, unless we specify otherwise, the tests have been executed using the parameters suggested by the authors, $\beta = 0.5$ and $r_h = 0.0003$. Different values have sporadically better results, but usually result in more brittle performances.

SDEM Algorithm: handling continuous variables

In [126] two different versions of the SDEM algorithm are proposed: a parametric one, and a kernel-based one. We will focus on the parametric version, which consistently performs better in the authors' own tests. It uses a classical Gaussian mixture model:

$$p(\vec{y}|\theta) = \sum_{i=1}^{k} c_i p(\vec{y}|\vec{\mu_i}, \Lambda_i),$$

where $k \in \mathbb{N}$, $c_i \geq 0$, $\sum_{i=1}^{k} c_i = 1$ and each $p(\vec{y}|\vec{\mu_i}, \Lambda_i)$ is a multivariate Gaussian distribution with d variables, where d is the number of continuous variables in the model. The vector of the averages is $\vec{\mu_i}$ and the covariance matrix is Λ_i.

Let s be an iteration index; we call a *sufficient statistic*

$$S_i^{(s)} = (c_i^{(s)}, \vec{\mu_i}^{(s)}, \Lambda_i^{(s)}) =$$

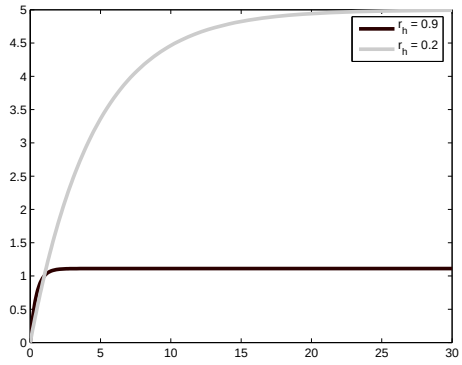


Figure 4.11: Plot of function $f(t) = \frac{1-(1-r_h)^t}{r_h}$

$$\frac{1}{t} \cdot (\sum_{u=1}^{t} \gamma_i^{(s)}(u), \sum_{u=1}^{t} \gamma_i^{(s)}(u) \cdot \vec{y}_u, \sum_{u=1}^{t} \gamma_i^{(s)}(u) \cdot \vec{y}_u \vec{y}_u^T) \quad (i = 1, \ldots, k),$$

where

$$\gamma_i^{(s)}(u) = \frac{c_i^{(s-1)} p(\vec{y_u} | \mu_i^{(s-1)}, \Lambda_i^{(s-1)})}{\sum_{j=1} k c_j^{(s-1)} p(\vec{y_u} | \mu_j^{(s-1)}, \Lambda_j^{(s-1)})}.$$

We also define $S_i^{(s)}(v)$ $(i = 1, \ldots, k)$ for the input $\vec{y_u}$:

$$S_i^{(s)}(v) = \frac{1}{t} \cdot (\gamma_i^{(s)}(v), \gamma_i^{(s)}(v) \cdot \vec{y}_v, \gamma_i^{(s)}(v) \cdot \vec{y}_v \vec{y}_v^T).$$

The parameters of the algorithm are the stabilizing parameter $\alpha \in [1, 2]$ and the forgetting factor r. The *SDEM* algorithm is shown in Listing 2.

SDEM has time complexity $O(d^3 k)$, but it can be reduced to $O(d^2 k)$ by adopting algorithms for computing the determinant and the inverse of the covariance matrices that use the estimate of the previous iteration.

It is easy to see that:

$$c_i^{(t)} = \sum_{j=1}^{t} (1 - r)^{t-j} r \gamma_i^{(j)}$$

which means that the smaller r, the more the old inputs weigh over the estimate. The values of $\mu_i^{(t)}$ and $\Lambda_i^{(t)}$ given as output by the updating formulas are such that the weighted sum of the logarithmic likelihoods is maximized, i.e.:

$$\sum_{j=1}^{t} (1 - r)^{t-j} r \gamma_i^{(j)} \ln p(\vec{y_j} | \mu_i^{(j)}, \Lambda_i^{(j)}).$$

Algorithm 2 The SDEM algorithm

Require: r, α, k.

$c_i^{(0)} \Leftarrow \frac{1}{k}$

$\vec{\mu_i}^{(0)}$ initialized uniformly dispersed over the input space

$\vec{\mu_i}^{(0)}, c_i^{(0)}, \bar{\vec{\mu}}_i^{(0)}, \Lambda_i^{(0)}, \bar{\Lambda_i}^{(0)}$ $(i = 1, \ldots, k)$ {Initialization}

$t \Leftarrow 1$ {Parameter updating}

while $(t \leq T)$ **do**

  `read` $\vec{y_t}$

  for all $i = 1, \ldots, k$ **do**

    $\gamma_i^{(t)} := (1 - \alpha r) \frac{c_i^{(t-1)} p(\vec{y_t} | \mu_i^{(t-1)}, \Lambda_i^{(t-1)})}{\sum_{j=1}^{k} c_j^{(t-1)} p(\vec{y_t} | \mu_j^{(t-1)}, \Lambda_j^{(t-1)})} + \frac{\alpha r}{k}$

    $c_i^{(t)} := (1 - r) c_i^{(t-1)} + r \gamma_i^{(t)}$

    $\bar{\vec{\mu}}_i^{(t)} := (1 - r) \bar{\vec{\mu}}_i^{(t-1)} + r \gamma_i^{(t)} \cdot \vec{y_t}$

    $\vec{\mu_i}^{(t)} := \frac{1}{c_i^{(t)}} \bar{\vec{\mu}}_i^{(t)}$

    $\bar{\Lambda_i}^{(t)} := (1 - r) \bar{\Lambda_i}^{(t-1)} + r \gamma_i^{(t)} \cdot \vec{y_t} \vec{y_t}^T$

    $\Lambda_i^{(t)} := \frac{1}{c_i^{(t)}} \bar{\Lambda_i}^{(t)} - \vec{\mu_i}^{(t)} \vec{\mu_i}^{(t)^T}$

  end for

  $t \Leftarrow t + 1$

end while

Outlier factor: Hellinger distance

After applying the previous algorithms to $(\vec{x_t}, \vec{y_t})$ we must compute how much the adjoint probability $p^{(t)}(\vec{x}, \vec{y})$ differs from $p^{(t-1)}(\vec{x}, \vec{y})$ with the adaptation to the new sample. The *Hellinger distance* is defined as follows:

$$S_H(\vec{x}_t, \vec{y}_t) = \frac{1}{r_h^2} \sum_{\vec{x}} \int (\sqrt{p^{(t)}(\vec{x}, \vec{y})} - \sqrt{p^{(t-1)}(\vec{x}, \vec{y})})^2 d\vec{y}.$$

The intuitive meaning of the formula is to compute how much the $p^{(t)}$ distribution differs from $p^{(t-1)}$ after the learning step on the input $(\vec{x}_t, \vec{y}_t)$.

This distance cannot be easily computed in this form, so we must resort to heuristics and approximations (as pointed out in [167]) to compute this distance online.

In order to automatically tune the threshold beyond which a data vector is to be considered an outlier, we modified SmartSifter by introducing a training phase during which the distribution of the anomaly scores is approximated, and an estimated quantile of the distribution is also computed. In this way we can directly set the IDS sensitivity as the percentage of packets we want to consider as outliers.

As we can see from Figure 4.12, the distribution of scores cannot be approximated well by a normal distribution, so computing the sample mean and variance would not be of help.

Therefore, we discretized the distribution with a quantization interval i, assuming that in each interval there is a uniform distribution, and computing the quantile of this discretized approximation. Of course, increasing i makes training faster and the approximation rougher.

The quantile Q of order $q \in [0, 1]$ can be determined through the following formula:

$$Q(q) = s \cdot \Big(j + \frac{q - \sum_{i=0}^{j} P(S_H \in I_i)}{P(S_H \in I_{j+1})} \Big),$$

where I_j is the interval in the discretization such that $P(S_H \in I_j) < q$ and $P(S_H \in I_{j+1}) \geq q$. Thus, setting the threshold of the anomaly score to $Q(q)$ the ratio of packets flagged as anomalous is more or less q.

As opposed to this totally unsupervised outlier determination, in [168] the authors of SmartSifter proposed a mixed supervised/unsupervised approach.

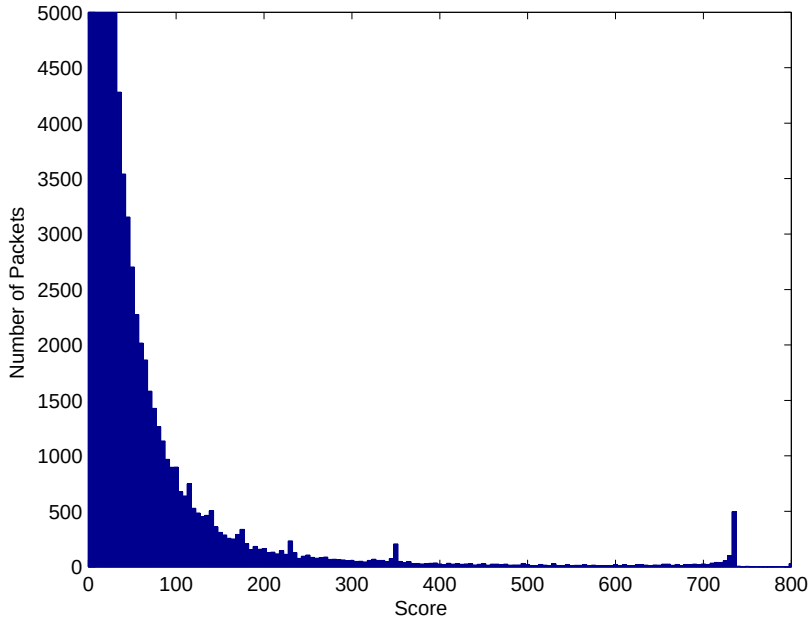


Figure 4.12: Distribution of scores

4.4.4 Feature selection

Feature selection is an important step for any learning application. We wish to stress this point, since the algorithms proposed in the literature have been applied to more or less arbitrary selections of features of the packets: our tests suggest that a deeper analysis should be done to determine which features are really important and which can be safely discarded. The importance of correctly choosing features for machine learning problem has been widely discussed in literature (see [169]), and our problem is not an exception: in our tests, keeping all the features effectively blinded the system, while accurately selecting a subset of them brought forth good results.

Feature Subset Selection for multivariate time series is a well studied process. In an unsupervised setting such as ours, unsupervised techniques for feature selection should be employed [170]. Unsupervised FSS techniques usually compute the similarity between features and removes redundancies in order to reduce the number of features. Usually this is accomplished through partitioning or clustering of the original feature set into partitions, each of which will be then represented by a single representative feature to form the reduced subset. A very representative

example of FSS technique can be found in [171]. However, no reliable method exists which takes into account categorical variables.

Therefore, we resorted to a much simpler approach, by testing different combinations of the variables. As a first consideration, we chose to use only categorical variables, since the SDLE algorithm is much more efficient than the SDEM algorithm (which contains matrix inversions) and since the non-categorical values (e.g. the *Window Size* and the *TTL* features) are not really significant and exhibit flaws due to the artificial generation (see Section 4.6 below for the details)

A set containing source port, destination port, TCP flags, source and destination address and the payload classification worked best in our setup. As noted above, the domain of the variables must be divided in cells. In our case we operated as follows:

- For source and destination ports, we divided the domain according to well known and widely used services, and created an "other ports" fallback class for unknown or upper ports.
- IP addresses were classified as either "external", or using three arbitrary "classes" of machines on the internal network.

A future extension of this work could automatically reconfigure the input features based on the particular network setup, provided that a reliable unsupervised algorithm for automatically selecting the feature subset is developed.

4.5 Evaluation of the proposed architecture

4.5.1 Our results

In order to evaluate our architecture in a repeatable manner, we ran the prototype over various days of traffic drawn from the 4th week of the 1999 DARPA dataset. We also added various attacks against the Apache web server and against the Samba service generated through the Metasploit framework (`www.metasploit.org`). The average results are reported in Table 4.2. The first column contains the sensitivity threshold set for the algorithm, and as we described it is a good statistical predictor of the percentage of data that will be flagged as outliers by the algorithm. Therefore, it is also a good predictor of the false positive rate, if the attack rate is not too high. The prototype is able to reach a 66.7% detection rate with as few as 0.03% false positives.

Examples of attacks which are detected easily by our algorithm are the ones indicated in the "truth file" of the dataset as `land, secret,`

Threshold	Detection Rate	False Positive Rate
0.03%	66.7%	0.031 %
0.05%	72.2%	0.055 %
0.08%	77.8%	0.086%
0.09%	88.9%	0.095%

Table 4.2: Detection rates and false positive rates for our prototype

`sechole, loadmodule` and `ps` (for a complete description of the attacks, see [172, 173]). Attacks that are less easy to detect, but still catched by our system are `mailbomb, httptunnel, crashiis` and `processtable`.

Attacks such as `sqlattack` and `phf` are more difficult to detect, probably because they are very similar to normal traffic.

In order to evaluate how well the proposed system performs, in the next sections we will compare it against two comparable state-of-the-art systems. As we will see, our prototype shows a better detection rate, with a number of false positives which is between one and two order of magnitudes lower than such systems. It should be noted that we refer, as much as possible, to the original experimental data as reported in literature, because in many cases we were not fully able to reproduce the results with the same degree of accuracy claimed by the authors.

4.5.2 Comparison with SmartSifter

The authors of SmartSifter in [167] tested their algorithm against the KDD Cup 1999 [174] dataset, which is extracted from the DARPA 1999 dataset [173] by converting the tcpdump records in connection records by a traffic reconstruction tool. For each connection, 41 attributes are recorded (34 of which continuous, and 7 categorical) and a label (which states whether or not the connection contains an attack).

In the original test three continuous variables (duration, bytes transmitted from source, bytes transmitted from destination), and a categorical one (the service) were used. The categorical variable is divided into five "cells": `HTTP`, `SMTP`, `ftp`, `ftp_data`, and `others`. In our opinion, this representation is way too reductive, and partly reflects the intrinsic biases of the DARPA dataset.

It is self evident that our approach is considerably different: we process packets, and not connections. Since there are a lot less connections than packets, should the detection rate and false positive rate values be comparable, a connection-based approach would be much better than a packet-based one, since there are many more packets than connections.

The authors of SmartSifter claim a 18% detection rate, with a 0.9%

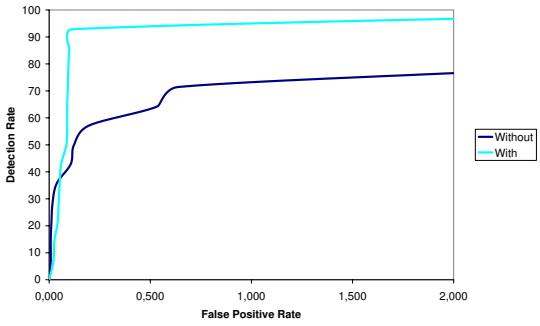


Figure 4.13: ROC curves comparing the behavior of SmartSifter with (lighter) and without (darker) our architecture

false positive rate (6421 connections). Our algorithm can instead reach a 92% detection rate with a 0.17% false positive rate (2035 packets), thus demonstrating a highly superior performance.

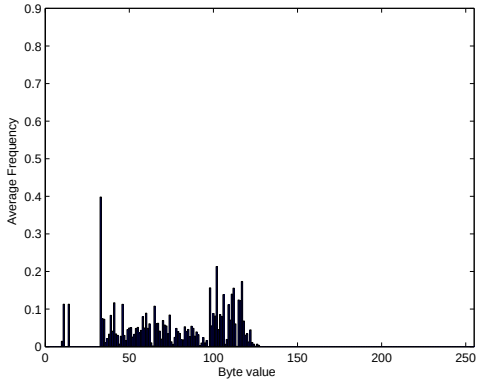
In Figure 4.13 we further show how our 2-tier architecture benefits the detection rate by comparing the ROC curves of the SmartSifter system with and without the payload classification tier by including and excluding the feature. The results are clearly superior when the first tier of unsupervised clustering is enabled, proving the usefulness of our approach.

4.5.3 Comparison with PAYL

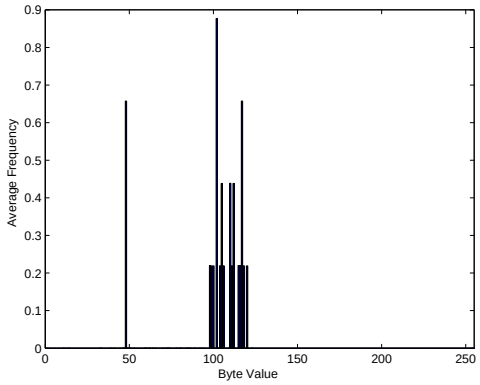
PAYL [127] is a prototype of intrusion detection system which uses part of the payload of packets: in fact, it is the only instance in literature, besides our own work, where such a concept is applied. PAYL builds a set of models of payload $M_{i,j}$ depending on payload size i and destination port j. The authors show how the frequency distribution of the payload bytes average varies significantly depending on i, j. We confirmed this result, and we show evidence of this in Figure 4.14.

Each model $M_{i,j}$ contains the average frequency, and the standard deviation, of each of the 256 possible byte values. In the detection phase, the model M of each packet is computed, and compared against the model $M_{i,j}$ created during training using a roughly simplified form of the Mahalanobis distance (a distance measure for statistical distributions).

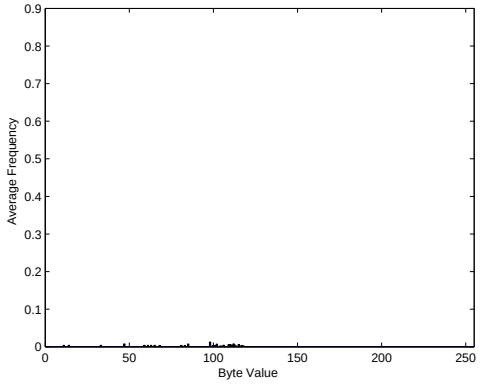
In order to avoid an explosion in the number of models, during training



(a) $M(25, 1460)$



(b) $M(80, 1460)$



(c) $M(25, 35)$

Figure 4.14: Average of byte values for three different models $M_{i,j}$

models that are similar are aggregated (this is very similar to what we propose in Section 5.1.6 for Markovian models in host based detection). The merging step aggregates couples of models $M_{i,j}$ and $M_{k,j}$ if $k \simeq i$ and the Manhattan distance between the two models is below a threshold. The authors also propose a Zipf-like representation of strings called "Z-string" which is beyond the scope of this section to explore. The same model was initially proposed in [175], and in [176] is proposed as a collaborative way of automatically creating worm signatures.

PAYL does not take into account the header informations. It also ignores the correlation over time in the packet flow. An additional limitation of PAYL is that it needs attack free traffic for training, even if the authors suggest, without demonstration, that if attacks are in minority with respect to normal packets.

Analyzing the results in in [127], we can see that PAYL shows its best results on communications over ports 80 and 21, which are also the ports that are best characterized in our own experiments on the first tier of our architecture. Symmetrically, port 25 (which is more difficult also for our prototype) is not handled very well by PAYL. The best overall results for PAYL show a detection rate of 58.7%, with a false positive rate that is between 0.1% and 1%. Our architecture can reach the same detection rate with a false positive rate below 0.03%, thus an order of magnitude better than PAYL, or on the other hand reach a 88.9% detection rate with no more than a 1% rate of false positives.

4.5.4 Resistance to fragmentation and basic evasion techniques

We also tested, in a very limited way, the resistance of the proposed architecture against fragmentation. In order to do this, we used *Fragroute* [55] to artificially introduce fragmentation in the DARPA dataset (in order to create fragmented, but normal packets and connections), fragmenting the 5% of the TCP traffic.

Then, we used the same tool to introduce fragmentation and evasion techniques on the attacks. Our results are shown in Table 4.3. Of course, in such extreme situation, the signal to noise ratio of the algorithm is worse, but still comparable or better than the results that comparable systems have on the non-fragmented data of the IDEVAL dataset.

4.6 Questioning the validity of the DARPA dataset

At the beginning of our research we were somehow surprised to find only one source of test data with full `tcpdump` payload record available

Threshold	Detection Rate	False Positive Rate
0.05%	28.6%	0.07 %
0.08%	35.8%	0.09%
0.8%	57.1%	0.57%
1.79%	64.28%	1.32%

Table 4.3: Detection rates and false positive rates with high fragmentation and use of evasion techniques

and a complete description of the traffic and attacks contained: the dataset created by the Lincoln Laboratory at M.I.T., also known as "DARPA IDS Evaluation dataset". These data have been collected by the IST (Information Systems Technology) group between 1998 and 1999 in order to evaluate detection rates and false positives rates of Intrusion Detection System. There are two datasets: 1998 and 1999 [173]. The dataset contains the tcpdump traces generated by a sensor inside the network and a sensor placed on the boundary, BSM auditing data for Solaris systems, NT auditing data for Windows systems, directory tree snapshots of each system, the content of sensitive directories, and inode data where available.

For privacy reasons, it is very difficult to gather the full payload traces of real networks. In addition, IDS researchers need clearly labeled data where attacks are described in full details, something which is usually impossible to achieve with real-world dumps. Other datasets exist (e.g. the DEFCON CTF packet capture [177]), but they are not labeled and do not contain "background traffic". Thus, most existing researches on network based IDSs use the DARPA datasets for evaluation. This is a crucial factor: any bias or error in the DARPA dataset has influenced, and will influence in the future, the very basic research on this topic.

For privacy reasons, both the background traffic and the attack traffic are artificially generated. The 1998 dataset is commented and described by a master's thesis [178]. The 1999 dataset [172], which we extensively used, contains the packet dumps (headers and payloads of packets in *tcpdump* format) of 5 weeks, over 2 sniffers, in a simulated network. One of the sniffers is placed between the gateway and 5 "target" machines (thus emulating an "internal" sniffer), while the other is placed beyond the gateway, recording packets flowing between the simulated LAN and the simulated Internet. The dataset also includes the system logs of the target machines, and BSM audit data, in order to be able to test host based IDSs. Both attack-free data and clearly labeled attack traces are present.

It is important to note, however, that these data have been artificially generated specifically for IDS evaluation. In fact, in [179] there is a detailed analysis of the shortcomings of the 1999 traffic sample set. In particular, the author notes that no detail is available on the generation methods, that there is no evidence that the traffic is actually realistic, and that spurious packets, so common on the Internet today, are not taken into account. The same can be said for checksum errors, fragmented packets, and similars. The simulated network is flat, and therefore unrealistic.

In [180] it is additionally noticed that the synthetic packets share strange regularities that are not present in real world traffic:

- SYN packets use always a 4-byte set of options, while in the real world this value ranges from 0 to 28 bytes.
- The TCP window size varies among seven fixed values ranging from 512 and 32120.
- There are just 29 distinct IP source addresses, and half of these account for over 99.9% of the traffic; in real world data, for a similar network with similar characteristics, over 24.000 unique addresses were counted.
- TTL and TOS fields are unrealistically similar for most packets. For instance, in the dataset 9 values of TTL out of 256 are used, while in real world data 177 different values can be seen; similarly, just 4 different TOS fields were observed in the dataset, against over 40.
- There are no packets with checksum errors in the IDEVAL dataset, while in real data a small but not null percentage of packets exhibits checksum errors; similarly, the dataset lacks fragmented packets, flag anomalies, etc.
- HTTP requests are all of the form `GET url HTTP/1.0` with 6 different keywords and 5 different User-Agent. Real traffic shows different commands, over 70 different keywords and over 800 different user agents; in real traffic commands and keywords are sometimes malformed, while in the dataset this is not present. Similar consideration apply to SMTP and SSH traffic.

The authors even propose a simple IDS system based on a single byte of the IP header (the third byte of the IP address, in particular), which achieves a 45% Detection Rate with just a bunch of false positives.

These characteristics make it difficult to understand whether IDSs tested and developed on DARPA traffic are capable of detecting true anomalies, or they are just capable of detecting the irregularities in the synthetic DARPA traffic. For instance, attacks *back*, *dosnuke*, *neptune*, *neptbus*, *netcat*, *ntinfoscan* and *quaeso* can be easily spotted, even by human eye, because they use TTL values that never appear into the training set. SMTP attacks are recognizable by the fact that they do not begin with a regular `HELO` o `EHLO` command; most attacks come from IP addresses that are not present in the training files; and so on.

For partially obviating these problems, we excluded the *TTL* and *Window Size* fields from our test. But evidently, this isn't enough. Thus, we positively validated our results using also smaller dumps collected and generated on our own internal network, as well as attacks generated with the Metasploit framework. These tests obviously lack repeatability, but this problem is shared by all the current researches on intrusion detection.

5 Host Based Intrusion Detection Applications

5.1 A Framework for Behavioral Detection

5.1.1 Introduction to Behavior Detection problems

In [181] we proposed to consider anomaly based intrusion detection in the more general frame of *behavior detection* problems. This type of problems has been approached in many different fields: psychology, ethology, sociology. Most of the techniques applied in these areas are of no immediate use to us, since they are not prone to be translated into algorithms. However, some useful hints can be drawn forth, in particular by analyzing the quantitative methods of ethology and behavioral sciences [182].

In order to understand the problem and to transfer knowledge between these different fields, we must analyze parallel definitions of concepts we will be dealing with. The first term is "behavior", which ethology describes as the stable, coordinated and observable set of reactions an animal shows to some kinds of stimulations, either inner stimulations (or motivations) or outer stimulations (or stimuli). The distinction between "stimuli" and "motivations" is as old as ethology itself, being already present in Lorenz's work [183].

Our definition of "user behavior" is quite different. We could define it as the "coordinated, observable set of actions a user takes on a computer system in order to accomplish some task". Depending on the observation point we assume, we can give different definition of actions, but for the scope of this introductory reasoning we will define them as the commands, the data communications and the inputs that the user exchanges with the system. We wish to make clear that our effort is not focused on the behavior of the computer system (which is by definition entirely predictable) but on the behavior of the user, which has relevant intentional components.

We will also make use of the concept of "typical behavior", which quantitative ethology would describe as the "most likely" one. In our definition, this behavior is the "normal" user behavior, as opposed to an

"atypical" behavior which is not, however, always devious or dangerous.

5.1.2 Motivations for action and action selection

This consideration brings us to the point of analyzing the motivations of behavior. We are interested in detecting any anomalous behavior which is motivated by the desire to break the security policy of the system. Anomalous behavior with no devious motivation is not a problem by itself; on the other hand perfectly normal, inconspicuous network traffic, motivated by a devious goal, should in some way be detected by a perfect intrusion detection system.

Even if terminology varies from school to school in behavioral sciences, we can recognize three broad levels of increasing complexity in the analysis of behavior: reflex behavior (sensorial stimuli and innate reactions), instinctual behavior (genetically evolved, innate behavior of a species), and finally intentional behavior, with actions that an animal begins autonomously to reach its own goals.

Clearly, when dealing with computer misuse, we are mostly dealing with intentional behavior, and we need to define what *motivates* an action. The concept of motivation is crucial to ethology, and it has been a theme of a number of philosophical researches as well. Without getting deeply into the philosophical debate, we can define motivations as the dynamic factors of behaviors, which trigger actions from an organism and direct it towards a goal. We will try to recognize which motivations are behind a particular behavior of a user.

The problem of understanding how, or why, an animal comes to perform certain activities and not others, mixing all these sometimes conflicting inputs, is known as the action selection problem, and has been studied for a very long time [184]. In computer science, this problem has been widely studied for designing rational agents in the AI field. Computational models have been developed for action selection, often jointly with observations drawn from ethologic studies, for example to develop the so-called animats [185]. However, it is important to remember that there's no proof that an arbitration mechanism for action selection is present in real animals, and some work demonstrates that it's not necessary [186]. Other works show that behavior is not simply a product of the "state" of the agent, but is instead a joint product of the agent, the environment surrounding it and the observer, who is giving a particular meaning to the actions he perceives in the agent.

Our models to infer the motivations of a particular sequence of action are based on the supposition that there is actually a meaning to be discovered in that sequence. While this is an acceptable premise for

intrusion detection (user actions have almost always a rational explanation), it may be a radically wrong approach for ethology. We need to take into account this difference while trying to adapt ethological and behavioral models to IDSs.

5.1.3 Fixed action patterns, modal action patterns, and ethograms

Closely associated with these concepts are *patterns*, elements shared by many slightly different behaviors, which are used to classify them. The concept of "behavioral pattern" is widely used in ethology.

Ethologists typically define as *Fixed Action Patterns* (FAP) the atomic units of instinctual behavior. FAPs have some well defined characteristics: they are mechanic; they are self-similar (stereotyped) in the same individual and across a species, and they are extensively present; they usually accomplish some objective. More importantly, they are atomic: once they begin, they are usually completed by the animal, and if the animal is interrupted, they are aborted.

A FAP must also be independent from (not correlated with) other behaviors or situations, except at most one, called a "releasor", which activates the FAP through a filter-trigger mechanism, called Innate Release Mechanism (IRM). The IRM can be purely interior, with no external observable input (emitted behavior), or it can be external (elicited behavior). In the latter case, sometimes the strength of the stimulus results in a stronger or weaker performance of the FAP (response to supernormal stimulus). In other cases, there is no such relation.

In [187], the whole concept of FAPs and IRMs is examined in detail. The author criticizes the rigid set of criteria defining a FAP, in particular the fact that the IRM must be different for each FAP; the fact that the IRM has no further effect on the FAP once it has been activated; and the fact that components of the FAP must fall into a strict order. Many behaviors do not fall into such criteria. Barlow proposes then to introduce MAPs, or *Modal Action Patterns*, action patterns with both fixed and variable parts, which can occur in a different order and can be modulated during their execution. Barlow suggests that the environment can modulate even the most stereotyped behavior. His definition of MAP is a "spatio-temporal pattern of coordinated movement that clusters around some mode making it recognizable as a distinct behavior pattern". Unfortunately, the flexibility of a MAP is difficult to implement in a computer-based model of behavior.

A subset of FAPs, called "displays", are actually communication mechanisms. In an interesting chain of relations, a display can be the releasor

of an answer, creating a communication sequence. An interesting characteristic of displays is the principle of *antithesis*, stating that two displays with opposite meanings tend to be as different as they can be. This is not necessarily true in behavior detection problems: for example, malicious computer users will try to hide behind a series of innocent-like activities.

We must also introduce the concept of an *ethogram*, which is an attempt to enumerate and describe correctly and completely the possible behavioral patterns of a species. On the field, an ethologist would observe the behavior of animals and list the different observed behavioral patterns in a list, annotated with possible interpretations of their meaning. Afterwards, s/he would observe at fixed interval the animals and "tick" the appropriate squares in an ethogram, generating a sequence data on the behavior of the observed animals. A similar discretization will be used also in our framework.

5.1.4 A methodology for behavioral detection

We will try to exploit the similarities we have found, in order to propose a framework for studying behavior detection and classification problems.

First of all, we need to specify which kind of displays of behavior we can detect and build appropriate sensors for detecting them. It is not difficult to collect and analyze the logs of a workstation, but detecting the behaviors of users in a virtual classroom environment could be difficult. For our example architecture we choose to use the interactions with a terminal. Other likely displays that could be analyzed are the logs of the interactions between a user and a web application, the sequence of system calls generated by user processes [188], or the generation of audit data (using for instance the syslog facilities of UNIX and similar systems).

As a second step, we must choose an appropriate model for representing the behavior. We could approach the problem at different levels of abstraction, making hypotheses on the action selection problem (as seen in 5.1.2) and analyzing the actual process which generates the behavior. However, we will use a traditional approach in quantitative behavior study, trying to model just the sequence of the displays of behavior, in order to infer various properties about the subject. In order to choose an appropriate model, we must understand if we want a binary classification, or a more complex one with several disjunct classes, or even one with overlapping categories.

Upon this model we must build an inference meta-model, which can help us learn actual parameters from observed data in order to tune the

model. This is a classical instance of machine learning problem. Finally, we must set thresholds and logics that help us extract useful information from the observed behavior. Due to space constraints, we will now focus our discussion on how to build an appropriate model for representing the behavior. As a future work we will deal with the other steps required for building a complete behavior detection system.

5.1.5 Representing behavior: Markov Models

Markov models are widely used in quantitative behavioral sciences to classify and report observed behaviors. In particular, in ethology simple Markov Models are built on field observation results. A time domain process demonstrates a Markov property if the conditional probability density of the current event, given all present and past events, depends only on the K most recent events. K is known as the *order* of the underlying model. Usually, models of order $K = 1$ are considered, because they are simpler to analyze mathematically. Higher-order models can usually be approximated with first order models, but approaches for using high-order Markov models in an efficient manner have also been proposed, even in the intrusion detection field [189].

A first order Markov Model is a finite set of N states $S = \{s_1, s_2, \ldots s_n\}$, each of which is associated with a (generally multidimensional) probability distribution. Transitions among the states are governed by a set of probabilities called transition probabilities $a_{i,j} = P\{t = k+1,\ s_j \mid t = k,\ s_i\}$ (whereas in order K models the probability depends on the states in the K previous steps, generating a $K+1$-dimensional array of probabilities). We consider a time-homogeneous model, in which $A = a_{i,j}$ is time-independent. This type of model is also called "observal" Markov Model, since the state is directly observable.

In a Hidden Markov Model, in any particular state, an outcome or observation o_k can be generated according to a probability distribution associated to the state ($b_{j,k} = P\{o_k \mid s_j\}$), in an alphabet of M possible observations. These probabilities obviously form a matrix $B = b_{j,k}$ which we also suppose to be time independent. Only the outcome, and not the state, is visible to an external observer; therefore states are "hidden" from the outside. The definition also implies an assumption which is probably not true: the output is assumed to be statistically independent from the previous outputs. If the observations are continuous, then a continuous probability density function is used, approximated by a mixture of Gaussians. However, ethologists discretize animal behavior using FAPs and MAPs and ethograms, in order to simplify the model. In our case, user-computer interactions are mostly discrete sequences of

events. Obviously, observal Markov models are special cases of HMMs.

In order to use HMMs in behavior detection, we need to solve two common problems associated with HMMs [190]. The first is the *evaluation problem*, which means, given a sequence of observations and a model, what is the probability that the observed sequence was generated by the model. The second is the *learning problem*: building from data a model, or a set of models, that properly describe the observed behavior. A third problem, the so called *decoding problem*, is not of particular interest to us.

5.1.6 A Bayesian algorithm for building Markovian models of behavior

The *evaluation problem* is trivial to solve in the case of a normal model, more complex to solve in the case of an HMM: in this case, the naive approach yield a complexity of N^T, where T is the length of the sequence of observations. The so-called forward algorithm [191] can be used, which has a complexity of N^2T.

The *learning problem* is more complex, in particular if we do not know the structure of the model. First of all, we need to choose the order of the model we will use. Often a first-order approximation is used for simplicity, but more complex models can be considered. A good estimate for an HMM can be extracted from data using the criteria defined in [192]; for normal Markov models, a χ^2-test for first against second order dependency can be used [193], but also an information criterion such as BIC or MDL can be used.

In order to estimate the correct number of states for an HMM, in [194] an interesting approach is proposed, by eliminating the time dependency and constructing a classification by means of clustering of the observations, considering each state as a generation mechanism.

Once we have chosen the model structure, learning a sequence of T observations means to find the matrices $\{A, B\}$ that maximize the probability of the sequence: $maxP[o_1 o_2 \dots o_T | A, B]$. This is computationally unfeasible, however the Baum-Welch algorithm [195] can give a local maximum for that function. Another approach to the parameter estimation problem is proposed in [196]. If the model is not hidden, however, the calculations become simple.

In many earlier proposals for the use of Markovian models in intrusion detection [103] the authors either build a Markov model for each user and then try to find out masquerading users (users accessing illicitly the account of another user); or they build a Markov model for the generic user and flag as anomalous any user who behaves differently.

The first approach brings an explosion of models, lacking generalization or support for users who are not identified uniquely to the system, while the second approach ignores the existence of different classes of users on the system.

In order to account for the existence of different *classes* of user behaviors, we propose the following algorithm, based on a Bayesian approach. Denoting with M a generic model and with O a sequence of observations, $P(M|O) \propto P(O|M)P(M)$. This means that, if we have a set of I models $M_1, M_2 \dots M_I$, the most likely model for the sequence of observations O is given by: $max_i\, P(M_i|O) = max_i\, P(O|M_i)\, P(M_i)$

We need now to choose an appropriate prior $P(M_i)$ for the models. Let us suppose that this procedure is iterative, which means that we have built the existing I models out of K observation sequences $O_1 \dots O_K$, iteratively associating each sequence with the best-fitting model and retraining the model with the new observations. This also means that we need to define a criterion for choosing whether it is appropriate to associate the new observations O_k with an existing model, or to create a new model for representing them.

A common decomposition for studying the prior of the model would be $P(M_i) = P(\theta_i|M_s)P(M_s)$, denoting with $P(\theta_i)$ the probability of the particular parameter set of M_i given a basic structure M_s and with $P(M_s)$ the probability of the structure itself. However, this type of approach leads to very complex calculations.

Using a simpler approach, we could proceed as follows. Let us call O_i the union of the observation sequences that have generated model M_i. We can build a non-informative prior criterion such as:

$$P(M_i) = \left(\frac{|O_i| + |O_k|}{(\sum |O_i|) + |O_k|} \right)^{log(|O_k|)} \tag{5.1}$$

which penalizes more particular models, favoring more general ones. Inserting the exponent $log(|O_k|)$ is necessary in order to account for the fact that different length of observation strings will generate different orders of magnitude in posterior probability. This generates also a simple criterion for the creation of new models. In fact, denoting with M_{I+1} a new model built on the new observations O_k, we would choose: $max_i P(M_i|O_k) = max_i P(O|M_i)P(M_i)$ with $1 \leq i \leq I+1$, defining:

$$P(M_{I+1}) = \frac{|O_k|}{(\sum |O_i|) + |O_k|}$$

In this way, the prior biases the probability towards more general models instead of more fitting but less general ones, averaging out the fact

that less general models tend to have an higher posterior probability $P(M_i|O_k)$. Once we have selected which model the k-th sequence O_k will be associated with, we re-train the model including in training data the new sequence.

Afterwards, we may optionally include a *merging* step, which means we will try to find couples of models M_i, M_j such that, denoting with $M_{i,j}$ the "merged" model and with O_i and O_j the observations associated with M_i and M_J:

$$P(O_i \cup O_j|M_{i,j})P(M_{i,j}) > P(O_i|M_i)P(M_i)$$
$$P(O_i \cup O_j|M_{i,j})P(M_{i,j}) > P(O_j|M_j)P(M_j)$$

In this case, a suitable criterion for selecting models to merge and for merging them must be also researched. There are some examples in literature of criteria for measuring a distance between two Markov models, for instance in [197] the following (asymmetric) distance is proposed: $D(M_i, M_j) = 1/T[logP(O^{(i)}|M_i) - logP(O^{(i)}|M_j)]$, where $O^{(i)}$ is a sequence of observations generated by model M_i. Criteria for merging HMM models can be found in [198] [199], where they are proposed as a suitable way to induce the models by aggregation.

If we wish to incorporate the insights from section 5.1.3 on the presence of FAPs and MAPs in behavior, we will need to use higher order models, because we need to express the probability on the base of a history. A suggestion that we may borrow from Barlow's studies on modal components of behavior, however, is that we may also want to detect clusters of states in the Markov chain that exhibit the following properties: they have "similar" outgoing transition probabilities and "similar" symbol emission probabilities (if we are dealing with an HMM). These states can be collapsed together in a single state, with simple probability calculations that we omit. This method is also applied in quantitative behavioral science, see [200].

5.1.7 A proof-of-concept behavior detector

For proof-of-concept testing of our framework, we acquired test data from a limited number of users of two different terminal systems, with 10 users on a system and 9 on the other, and 4 months of data. We prepared the data by discarding command options and encoding each different command with a number. In the first system, for example, on 2717 interactions, 150 unique commands were used. However, as we can see in Figure 5.1, a significant fraction of the interactions consists of a limited subset of frequently used commands, so we can set a minimum threshold below which we will group all the commands together as "other".

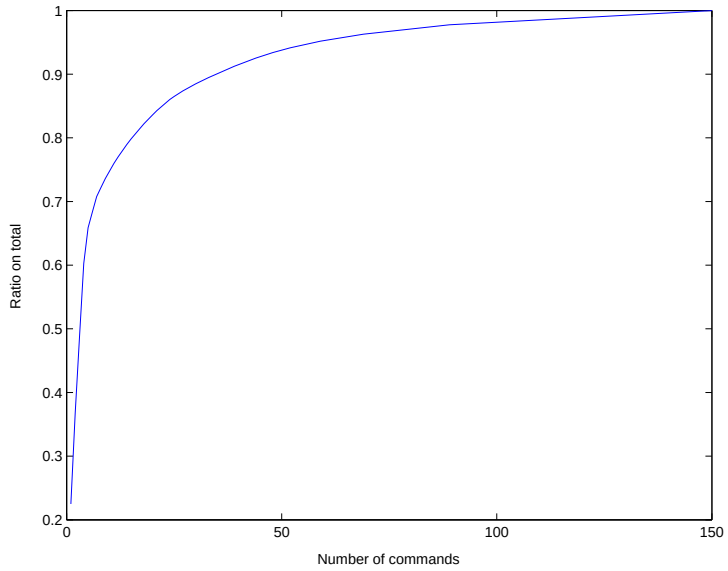


Figure 5.1: Cumulative distribution of commands

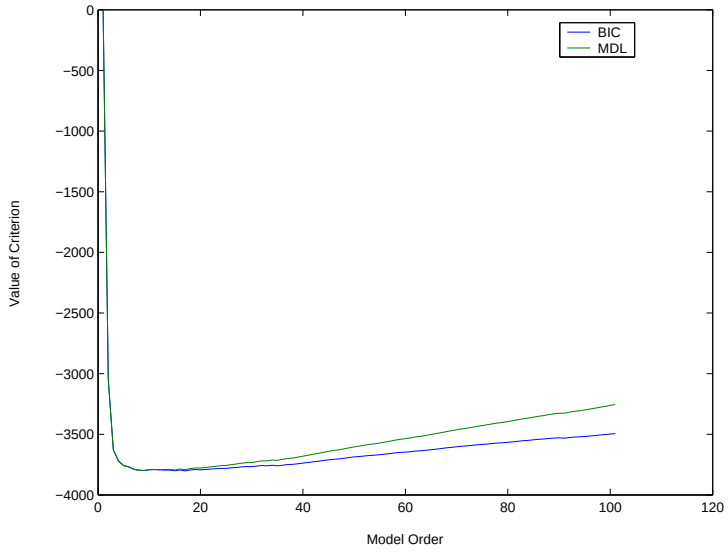


Figure 5.2: Information criteria: MDL and BIC

Commands	Our algorithm		Naive Markov
	Fitting	Detection	Detection
60	90.0	95.9	90.0
40	89.2	95.6	87.8
30	87.8	94.8	86.3
20	84.8	92.9	78.9
10	67.4	81.1	65.6
8	61.1	78.5	59.3
6	38.1	63.3	51.5
4	20.4	55.9	50.7

Table 5.1: Performance of our algorithm vs. naive application of Markov Models

In order to estimate the optimal order for the model, we used both the BIC and MDL criteria, and both agree on order $k = 4$ as being the optimal value. However, as a first approximation we will use a first-order model to fit the observations (approximation supported by the steep descent in criteria curves, which can be observed in Figure 5.2). Also, since the observations are finite in number, we use a normal Markov chain and not an HMM, to fit it.

We trained a set of Markov models following the basic algorithm outlined above. We experimented with various combinations of thresholds and parameters: we show the results in Table 5.1, compared with a naive application of Markov models (by pre-labeling the traces and building a transition matrix for each user, or class of user). For our models, we show also a measure of the overfitting of the model classes on the training sequences (the higher the fitting, the lower the generalization capacity of the algorithm). Creating Markov models with a high number of nodes increases both detection rate (because users are identified by relatively uncommon commands they perform) and overfitting. Using only 6 types of commands, we obtain a much better generalization and still a 63.3% detection rate. The rate may seem overall low, but it is still much higher than the detection rate of a naive application of Markov models. The computation time for building the model is quite higher than the naive one (about 6 times higher), but still in the order of seconds. At runtime, there is no difference in complexity between our model and a naive one.

5.2 System Call Argument Analysis: the LibAnomaly framework

5.2.1 LibAnomaly and SyscallAnomaly: an introduction

We decided to begin our work on system call anomaly detection by analyzing the LibAnomaly project.

LibAnomaly is a tool created by the Reliable Software Group of the University of California, Santa Barbara [107]. LibAnomaly implements a framework to build anomaly detection systems. The authors used LibAnomaly's API to implement a demo system called SyscallAnomaly, which can detect anomalies by analyzing system call arguments. SyscallAnomaly works by analyzing either Solaris or Linux syscall traces (respectively in BSM or Snare formats).

Both projects are developed in C++ and available under a GPL license. No exhaustive manual or documentation of the projects exist, besides what is present in scientific articles. Thus, in order to study SysCallAnomaly and propose improvements and alternative implementations, we used the following process:

1. We studied the theoretical foundations described in [107].
2. We recreated the test environment and results claimed by the authors.
3. We analyzed thoroughly the source code of the applications in order to discover implementation details.

On the base of this process, we proposed an alternative system, which implements some of the ideas of SyscallAnomaly along with Markovian based modeling, clustering and behavior identification, and which outperforms the original application, as we will show in the following.

5.2.2 LibAnomaly models: design and implementation

Generic structure of a model

LibAnomaly implements a set of anomaly models. Any of these models implements four methods:

1. `insert_item()`: adds a new element in the training set of the model;
2. `switch_mode()`: terminates the learning phase, synthesizes the model from the training set, and begins the threshold tuning phase;

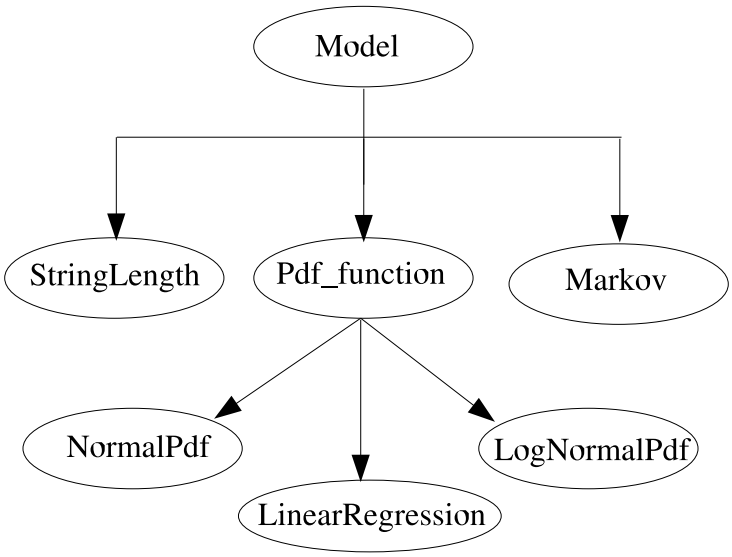


Figure 5.3: Class tree for `LibAnomaly` models

3. `check_item()`: gives the probability that a new item belongs to the model, i.e. the likelihood rating, for the detection phase;

4. `get_confidence()`: gives the confidence rating, i.e. how reliable is the model in describing normality.

LibAnomaly implements the abstract class `Pdf_function` (see Figure 5.3) to implement a generic probability distribution function:

- `Histogram`: a discrete probability distribution;
- `NormalPdf`: a normal probability distribution;
- `LognormalPdf`: a log-normal probability distribution;
- `LinearRegression`: a linear interpolation between input data.

String Length Model

The string length model computes, for the strings seen in the training phase, the campionary average and variance μ and σ^2. In the detection phase, let l be the length of the observed string. The model returns 1 if $l < \mu$, or $\frac{\sigma^2}{(l-\mu)^2}$ otherwise. This value measures the likelihood of the input string length with respect to the values observed in training.

Character Distribution Model

The model uses the class `Histogram`, which implements a discrete probability distribution. During training, each string is considered as a set of characters, which are inserted into an histogram, in decreasing order of occurrence. The actual value of the character is ignored. So the strings "a good example string" would have the following representation: 2 2 2 2 1 1 1 1 1 1 1 1 1 (because there are 4 letters that are repeated twice, and 10 that appear a single time). During the learning phase, a compact representation of the average and the variance of each position of the histogram is computed. For detection a χ^2 Pearson test returns the likelihood that the observed string histogram belongs to the learned model.

Structural Inference Model

The structural inference model learns the structure of strings. Firstly, strings are simplified using the following rules:

$$[A-Z] \rightarrow A$$
$$[a-z] \rightarrow a$$
$$[0-9] \rightarrow 0$$

In other words, uppercase characters, lowercase characters, and numbers, are lumped together. Other characters instead are kept. Here are two examples of this conversion:

`/usr/lib/libc.so` $\rightarrow /aaa/aaa/aaaa.aa$
`/etc/X11/XF86Config` $\rightarrow /aaa/A00/AA00Aaaaaa$

As a final preprocessing step, multiple occurrences of the same character are lumped together, as follows:

$/aaa/aaa/aaaa.aa \rightarrow /a/a/a.a$
$/aaa/A00/AA00Aaaaaa \rightarrow /a/A0/A0Aa$

Strings that after this compression are still longer than 40 characters are ignored by the model, perhaps for simplification.

A probabilistic grammar of the input strings is then built, using a Hidden Markov Model (HMM), and implementing the algorithms and the optimizations described in [199, 198, 201] which we already commented in Section 5.1.6. However, as we have shown, this type of Bayesian merging is heavily dependent on the choice of the prior of the Bayesian model, and this choice is not well documented in the literature of LibAnomaly.

Curiously, the probability values associated with the Markov model are ignored in the detection phase. The input string is preprocessed as outlined above and compared with the HMM. If the HMM can generate the string (i.e. the generation probability has a value greater than 0) a probability of 1 is returned, otherwise 0 is returned.

This awkward choice is probably explained by our observation over Equation 5.1 in Section 5.1.6, where we noticed that the length of observation string introduces a difference in probability which must be accounted for, in order not to penalize longer observation against shorter ones.

Token Search Model

The Token Search Model, during training, uses a statistical test to determine whether or not an input field contains a token, that is, a finite series of values. The test works in the following way: let I be the vector of inputs. Two vectors are created, A and B: at step n, $A[n] = A[n-1]+1$ (so A contains the first n natural numbers), while $B[n] = B[n-1]+1$ if $I[n-1] = I[n]$, or $B[n] = B[n-1]-1$ otherwise. A Kolgomorov-Smirnov non parametric test is then used to establish whether or not the vectors are correlated. If they are, the field probably contains a set of tokens, and the set of values observed during training is stored.

During detection, if the field has been flagged as a token, the input is compared against the stored values list. If it matches a former input, the model returns 1, else it returns 0.

5.2.3 SyscallAnomaly: design and implementation

SyscallAnomaly creates a profile of system calls for each different application. The input of SyscallAnomaly, for each instance of execution of an application, is a sequence of system calls $S = \{s_1, s_2, s_3, \ldots\}$, logged by operating system. Each system call s_i is composed by a list of arguments, a type and a return value.

SyscallAnomaly generates a profile for each system call type (e.g. `read`, `write`, `exec`, ...), for each application (e.g. `sendmail`, `telnetd`, ...). It does not take into account the sequence with which the system calls happen. The profile strives to capture the normal behavior of a program, by characterizing the normal arguments of each system call type inside that program.

These normal values are captured by the means of a set of models. Models are trained during a learning phase, and during the runtime/recognition phase they return the likelihood of a particular value

of an argument of a system call, based on previous observations of that system call in the context of the same application during training.

Each model, on each argument of the system call, operates independently. The probabilities are then aggregated to compute the total probability value of a system call. If this value is lower than a threshold, the call is flagged as anomalous. The threshold is learned by computing the maximum anomaly value over the whole training set, and incrementing this value of a user-defined percentage (a sensitivity threshold for the system).

We can see that SyscallAnomaly bases its structure on two great assumptions

- Firstly, that the attack actually appears and has some effect on system calls arguments, rather than on their sequence. Attacks that do not alter the content of system calls but just their sequence are undetectable by such a system.
- A second assumption is that anomalous system call arguments are different than training values more than the training values are different among themselves. Thus, the ability of detecting anomalies, even if the first assumption is satisfied, depends on the efficacy of at least a few of the various single models built upon arguments to capture normality and separate outliers (separately, since no correlation among models is analyzed).

SyscallAnomaly receives as an input a streams of events, corresponding to system call invocations system-wide. Each event carries the following informations:

- The invoking process and the program path,
- The invoked system call,
- A timestamp,
- The return value,
- The argument list.

For each event, if a profile for the same syscall in the context of the same program exists, the new arguments are added to that profile, otherwise a new profile is initialized. Not all the system calls are modeled, though. Out of more than 280 syscalls implemented in Linux, only 22 are considered. The authors probably chose only the calls that are invoked enough times to generate significant profiles, yet are sufficiently characterized to generate meaningful models.

The arguments are modeled according to their expected content. If the expected content is a file system path, the String Length, Character Distribution and Structural Inference models are used (collectively named "PathFactory"). If the expected content is a token, i.e. a flag, an opening mode, a UID or GID, and so on, the Token Search model is used instead ("FlagFactory"). A list of all the modeled system calls, along with the type of modeled values, is reported in Table 5.2.

During the detection phase, the probability value for each call is computed by computing the probability values for each of the model of each argument and then aggregating these models using the following equation:

$$\frac{\sum_{\forall\ model}^{i} confidence_i * log(probability_i)}{\#ofmodels} \tag{5.2}$$

5.2.4 Testing SyscallAnomaly on the IDEVAL dataset

The IDEVAL dataset, which we already described in 4.6, contains also host based auditing data in BSM format. However, it must be noted that many attacks (the ones against network services as well as the policy violation events) are not directly detectable through system call analysis. The most interesting attacks for testing SyscallAnomaly are the ones in which an attacker exploits a vulnerability in a local or remote service to allow an intruder to obtain or escalate privileges.

In particular, we use the BSM audit logs from a system named `pascal.eyrie.af.mil`, which runs a Solaris 2.5.1 operating system. The dataset contains 25 buffer overflow attacks against 4 different programs: `eject`, `fdformat`, `ffbconfig` e `ps`.

We used data from weeks 1 and 3 for training, and data from weeks 4 and 5 for testing purposes.

In addition to the four programs named above, we ran SyscallAnomaly also on three other programs, namely `ftpd`, `sendmail` and `telnetd`, which are not subject to attacks, in order to better evaluate the false positive rate of the system. In Table 5.3 we compare our results with the version of SyscallAnomaly available on the Internet [202] with the results reported in [107]

As can be seen, our results differ from the ones reported in [107], but the discrepancy can be explained by a number of factors:

- The version of SyscallAnomaly and LibAnomaly available online could be different than the one used for the published tests.
- A number of parameters can be tuned in SyscallAnomaly, and a different tuning could produce the discrepancy.

Syscall name	Model applied to each argument
`open`	`pathname` → PathFactory `flags` → FlagFactory `mode` → *none*
`execve`	`filename` → FlagFactory `argv` → PathFactory
`setuid` `setgid`	`uid` → FlagFactory
`setreuid` `setregid`	`ruid` → FlagFactory `euid` → FlagFactory
`setresuid` `setresgid`	`ruid` → FlagFactory `euid` → FlagFactory `suid` → FlagFactory
`rename`	`oldpath` → PathFactory `newpath` → PathFactory
`symlink` `link`	`oldpath` → PathFactory `newpath` → PathFactory
`mount`	`source` → PathFactory `target` → PathFactory `flags` → FlagFactory
`umount`	`target` → PathFactory `flags` → FlagFactory
`exit`	`status` → FlagFactory
`chown` `lchown`	`path` → FlagFactory `owner` → FlagFactory `group` → FlagFactory
`chmod`	`path` → PathFactory `mode` → FlagFactory
`creat`	`pathname` → PathFactory `mode` → FlagFactory
`mkdir`	`pathname` → PathFactory `mode` → FlagFactory
`mknode`	`pathname` → PathFactory `mode` → FlagFactory `dev` → FlagFactory
`unlink`	`pathname` → PathFactory
`rmdir`	`pathname` → PathFactory

Table 5.2: Recorded syscalls and applied models in SyscallAnomaly

Program	False Positives	
	Reported in [107]	Our experiment (number of syscalls)
`fdformat`	0	1 (4)
`eject`	0	1 (6)
`ps`	0	2 (10)
`ftpd`	14	2 (45)
`telnetd`	17	2 (198)
`sendmail`	8	4 (97)

Table 5.3: Experimental Evaluation of SyscallAnomaly on the IDEVAL dataset

- Part of the data in the IDEVAL dataset under consideration are corrupted or malformed.
- In [107] it is unclear if the number of false positives is based on the number of executions erroneously flagged as anomalous, or on the number of anomalous syscalls detected..

These discrepancies make a direct comparison difficult, but our numbers confirm that Syscall Anomaly performs well overall as a detector. However, the false positives and the anomalies are interesting to study, in order to better understand how and where SyscallAnomaly fails.

5.2.5 A detailed analysis of experiments and false positives

fdformat

`fdformat` is a simple program which is used to format removable media on UNIX-like systems. Normally, it has a very simple and predictable execution flow (and thus should be very well characterized): device `mm@0:zero` is opened, dynamic libraries are loaded, and finally the device `vol@0:volctl` is accessed. A typical execution instance has the following behavior:

```
execve: /usr/bin/fdformat, fdformat
open: /devices/pseudo/mm@0:zero, crw-rw-rw-
open: /usr/lib/libvolmgt.so.1, -rwxr-xr-x
open: /usr/lib/libintl.so.1, -rwxr-xr-x
open: /usr/lib/libc.so.1, -rwxr-xr-x
open: /usr/lib/libadm.so.1, -rwxr-xr-x
open: /usr/lib/libw.so.1, -rwxr-xr-x
```

System Call	`execve`
Argument 1	`/usr/bin/fdformat`
Argument 2	`fdformat\0x20\0x20\0x20\0x20[...]`
Model	**Probability (Confidence)**
String Length	10^{-6} (0)
Character Distribution	0.005 (0.995)
Structural Inference	10^{-6} (0.025)
Token Search	0.999999 (0)
Tot. Score (Thresh.)	*1.4043 (0.00137156)*

Table 5.4: True positive on `fdformat`: buffer overflow attack instance

```
open: /usr/lib/libdl.so.1, -rwxr-xr-x
open: /usr/lib/libelf.so.1, -rwxr-xr-x
open: /usr/platform/sun4u/lib/libc_psr.so.1, -rwxr-xr-x
open: /devices/pseudo/vol@0:volctl, crw-rw-rw-
open: /devices/pseudo/vol@0:volctl, crw-rw-rw-
open: /devices/pseudo/vol@0:volctl, crw-rw-rw-
open: /devices/pseudo/vol@0:volctl, crw-rw-rw-
exit: 0
```

The only attack in the dataset against `fdformat` is a buffer overflow with command execution (see Table 5.4). The exploit is visible in the `execve` system call, since the buffer overflow is exploited from the command line. Many of the models in `SyscallAnomaly` are able to detect this problem: the character distribution model, for instance, works admirably well. The anomaly value turns out to be 1.4043, much higher than the threshold (0.0013).

The interesting thing is that the string length and structural inference models, which should flag immediately this anomaly, work as expected, but are mostly ignored since their confidence value is too low.

Another alert happens in the opening of a localization file (Table 5.5), which triggers the string length model, creates an anomalous distribution of characters, and moreover the presence of numbers, underscores and capitals creates a structure that is flagged as anomalous by the structural inference model. The anomaly in the token search model is due to the fact that the open mode (`-r-xr-xr-x`) is not present in any of the training files.

Concluding, this is a very simple attack, which is detected with no effort; the detection of the opening of the localization file (which is a consequence of the attack) is also counted as a true positive, but is more

System Call	`open`
Argument 1	`/usr/lib/locale/iso_8859_1/[...]`
Argument 2	`-r-xr-xr-x`
Model	**Probability (Confidence)**
String Length	0.0096 (0.005)
Character Distribution	0.005 (0.995)
Structural Inference	10^{-6} (0.986)
Token Search	10^{-6} (1)
Tot. Score (Thresh.)	*8.186 (1.454)*

Table 5.5: True positive on `fdformat`: opening localization file

of a random side effect.

eject

`eject` is a similarly simple program, used to eject media: dynamic libraries are loaded, and the device `vol@0:volctl` is accessed; finally, the device `unnamed_floppy` is accessed.

```
execve: /usr/bin/eject, eject
open: /devices/pseudo/mm@0:zero, crw-rw-rw-
open: /usr/lib/libvolmgt.so.1, -rwxr-xr-x
open: /usr/lib/libadm.so.1, -rwxr-xr-x
open: /usr/lib/libintl.so.1, -rwxr-xr-x
open: /usr/lib/libc.so.1, -rwxr-xr-x
open: /usr/lib/libelf.so.1, -rwxr-xr-x
open: /usr/lib/libw.so.1, -rwxr-xr-x
open: /usr/lib/libdl.so.1, -rwxr-xr-x
open: /usr/platform/sun4u/lib/libc_psr.so.1, -rwxr-xr-x
open: /devices/pseudo/vol@0:volctl, crw-rw-rw-
open: /vol/dev/rdiskette0/unnamed_floppy, crw-rw-rw-
open: /devices/pseudo/vol@0:volctl, crw-rw-rw-
open: /vol/dev/rdiskette0/unnamed_floppy, crw-rw-rw-
open: /vol/dev/rdiskette0/unnamed_floppy, crw-rw-rw-
exit: 0
```

The attack is fairly similar to the attack on `fdformat`, and the same considerations apply. However, in this case a false positive is present when a removable unit unseen in training (`c0t6d0/volume_1`) is opened (see Table 5.7).

System Call	`execve`
Argument 1	`/usr/bin/eject`
Argument 2	`eject\0x20\0x20\0x20\0x20[...]`
Model	**Probability (Confidence)**
String Length	10^{-6} (0)
Character Distribution	0.005 (0.928)
Structural Inference	10^{-6} (0.025)
Token Search	0.999999 (0)
Tot. Score (Thresh.)	*1.316 (0.0012)*

Table 5.6: True positive on `eject`: buffer overflow on `execve`

System Call	`open`
Argument 1	`/vol/dev/rdiskette0/b9`
Argument 2	`crw-rw-rw-`
Model	**Probability (Confidence)**
String Length	0.667 (0.005)
Character Distribution	0.99 (0.995)
Structural Inference	10^{-6} (1)
Token Search	0.999 (1)
Tot. Score (Thresh.)	*8.186 (1.454)*

Table 5.7: False positive on `eject`: use of a new unit

The structural inference model is the culprit of the false alert, since the name structure is different than the previous one for the presence of an underscore. As we will see later on in Section 5.2.6, this extreme brittleness of the transformation and simplification model is a constant weakness of the structural inference model.

ps

`ps` is a jack-of-all-trades program to monitor process execution, and as such is much more articulated in its options and execution flow than any of the previously analyzed softwares. The sequence of system calls, however, does not vary dramatically depending on the user specified options. Besides library loading, the program opens `/tmp/ps_data` and the files containing process information in `/proc`.

```
execve: /usr/bin/ps, ps
open: /devices/pseudo/mm@0:zero, crw-rw-rw-
open: /usr/lib/libw.so.1, -rwxr-xr-x
open: /usr/lib/libintl.so.1, -rwxr-xr-x
open: /usr/lib/libc.so.1, -rwxr-xr-x
open: /usr/lib/libdl.so.1, -rwxr-xr-x
open: /usr/platform/sun4u/lib/libc_psr.so.1, -rwxr-xr-x
open: /tmp/ps_data, -rw-rw-r--
open: /proc/3345, -rw-------
open: /proc, dr-xr-xr-x
open: /proc/00000, -rw-------
open: /proc/00001, -rw-------
open: /proc/00002, -rw------- [...]
```

Also in this case, the attack is a buffer overflow on a command-line parameter. We do not repeat our observations above. In this case, as was the case for `fdformat`, a correlated event is also detected, the opening of file `/tmp/foo` instead of file `/tmp/ps_data` (see Table 5.8).

Token search model and structural inference model flags an anomaly, because the opening mode is unseen before, and because the presence of an underscore in `/tmp/ps_data` makes it structurally different than `/tmp/foo`. However, if we modify the exploit to use `/tmp/foo_data`, the structural inference model goes quiet.

A false positive happens when `ps` is executed with options `Iux`, because as we see in Table 5.9 the structural inference model strongly believes this to be an attack. Another false positive happens when a zone file is opened, because during training no files in `zoneinfo` were opened (details in Table 5.10).

System Call	`open`
Argument 1	`/tmp/foo`
Argument 2	`-rw-r--r--`
Model	**Probability (Confidence)**
String Length	0.17 (0.005)
Character Distribution	0.9 (0.995)
Structural Inference	10^{-6} (1)
Token Search	10^{-6} (1)
Tot. Score (Thresh.)	*6.935 (1.457)*

Table 5.8: True positive on `ps`: opening `/tmp/foo`

System Call	`execve`
Argument 1	`/usr/bin/ps`
Argument 2	`ps -Iux`
Model	**Probability (Confidence)**
String Length	0.0687 (0.005)
Character Distribution	0.95 (0.995)
Structural Inference	10^{-6} (0.41)
Token Search	0.999 (1)
Tot. Score (Thresh.)	*1.434 (0.017)*
Similar events	`ps -aKx, ps -aNx, ps -a[x)`

Table 5.9: False positive on `ps`: different command line arguments

System Call	`open`
Argument 1	`/usr/share/lib/zoneinfo/US/Eastern`
Argument 2	`-rw-r--r--`
Model	**Probability (Confidence)**
String Length	0.0063 (0.005)
Character Distribution	0.005 (0.995)
Structural Inference	10^{-6} (1)
Token Search	10^{-6} (1)
Tot. Score (Thresh.)	*8.232 (1.457)*

Table 5.10: False positive on `ps`: zone file opening

System Call	`open`
Argument 1	`/export/home/ftp/dev/tcp`
Argument 2	`crw-r--r--`
Model	**Probability (Confidence)**
String Length	0.063 (0.005)
Character Distribution	0.95 (0.995)
Structural Inference	0.999 (1)
Token Search	10^{-6} (1)
Tot. Score (Thresh.)	*3.467 (1.463)*

Table 5.11: False positive on `ftpd`: opening a file never opened before

Concluding, also in this case the detection of the opening of the `/tmp/foo_data` file is more of a random side effect than a detection, and in fact the model which correctly identifies it then creates false positives for many other instances.

ftpd

`in.ftpd` is a common FTP server, and as such is subject to a variety of commands. However, also because of the shortcomings of the IDEVAL dataset (see Section 5.4), the system call flow is fairly regular. After access to libraries and configuration files, the logon events are recorded into system log files. A `vfork` call is then executed to create a child process for actually serving the client requests

```
execve: /usr/sbin/in.ftpd, in.ftpd
open: /devices/pseudo/mm@0:zero, crw-rw-rw-
open: /usr/lib/libsocket.so.1, -rwxr-xr-x [...]
open: /etc/shells, -rw-r--r--
open: /etc/ftpusers, ----------
open: /etc/nsswitch.conf, -rw-r--r-- [...]
open: /var/adm/wtmp, -rw-rw-rw-
open: /var/adm/wtmpx, -rw-rw-rw- [...]
vfork:
open: /devices/pseudo/clone@0:tcp, crw-rw-rw-
open: /export/home/mistyd/NewProjects/Working
      /Linux/Sparc/., drwxrwxr-x [...]
exit: 0
```

System Call	`open`
Argument 1	`/etc/shadow`
Argument 2	`-rwxrwxrw-`
Model	**Probability (Confidence)**
String Length	0.155 (0.005)
Character Distribution	0.975 (0.995)
Structural Inference	0.999 (1)
Token Search	10^{-6} (1)
Tot. Score (Thresh.)	*3.46 (1.46)*

Table 5.12: False positive `ftpd`: opening `/etc/shadow` with a mode different than usual

False positive: opening /etc/shadow with an unforeseen mode In this case, the false positives mostly happen because of the opening of files never accessed during training (e.g. the device `/export/home/ftp/dev/tcp`, as shown in Table 5.11), or with unusual modes (as happens with `/etc/shadow` in Table 5.12). In this case, the token search model is one of the culprits.

telnetd

`in.telnetd` has a very simple execution flow: after shared libraries have been opened, two `fork` calls are executed, the user logon is logged, and devices `clone@0:logindmux` and `pts@0:0` are opened.

```
execve: /usr/sbin/in.telnetd, in.telnetd
open: /devices/pseudo/mm@0:zero, crw-rw-rw-
open: /usr/lib/libsocket.so.1, -rwxr-xr-x [...]
open: /devices/pseudo/clone@0:ptmx, -rw-r--r--
fork:
open: /devices/pseudo/pts@0:0, -rw-r--r--
open: /etc/netconfig, -rw-r--r--
open: /etc/.name_service_door, sr--r--r--
open: /devices/pseudo/clone@0:logindmux, -rw-r--r--
open: /devices/pseudo/clone@0:logindmux, crw-------
fork:
open: /devices/pseudo/pts@0:0, crw--w----
open: /var/adm/utmpx, ---------- [...]
exit: 0
```

A false positive promptly happens when syslog.pid file is opened, something which didn't happen during training (see Table 5.13). The

System Call	`open`
Argument 1	`/etc/syslog.pid`
Argument 2	`-rw-r--r--`
Model	**Probability (Confidence)**
String Length	0.05 (0.005)
Character Distribution	0.95 (0.995)
Structural Inference	10^{-6} (1)
Token Search	0.999 (1)
Tot. Score (Thresh.)	*3.46 (0.63)*

Table 5.13: False positive on `telnetd`: opening `syslog.pid`

only model flagging an anomaly is the HMM of structural inference, because no files opened during training had a filename extension of any type. Once more, the robustness of structural inference is disputable.

sendmail

`sendmail` is a very complex program, with complex execution flows that include opening libraries and configuration files, accessing the mail queue (`/var/spool/mqueue`), transmitting data through the network and/or saving mails on disk. Temporary files are used, and the `setuid` call is also used, with an argument set to the recipient of the message (for delivery to local users).

```
execve: /usr/lib/sendmail, /usr/lib/sendmail
        -oi lizoletd@alpha.apple.edu
open: /devices/pseudo/mm@0:zero, crw-rw-rw-
open: /usr/lib/libkstat.so.1, -rwxr-xr-x
open: /usr/lib/libresolv.so.1, -rwxr-xr-x
open: /usr/lib/libsocket.so.1, -rwxr-xr-x [...]
open: /etc/nsswitch.conf, -rw-r--r--
open: /etc/mail/sendmailvars, ----------
open: /etc/mail/sendmailvars, ----------
open: /etc/mail/sendmail.cf, -rw-r--r--
open: /etc/mail/sendmailvars, ----------
open: /etc/mnttab, -rw-r--r--
open: /etc/mail/aliases.pag, -rw-r--r-- [...]
open: /var/spool/mqueue/qfIAA00307, -rw-------
open: /var/spool/mqueue/xfIAA00307, -rw-r--r--
open: /etc/ttysrch, -rw-r--r--
```

System Call Argument 1	`setuid` `2133`
Model	**Probability (Confidence)**
Token Search	10^{-6} (1)
Tot. Score (Thresh.)	*13.81 (*10^{-6}*)*

Table 5.14: False positive on `sendmail`: user seen for the first time

System Call Argument 1	`unlink` `/var/mail/emonca000Sh`
Model	**Probability (Confidence)**
String Length	$4 * 10^{-5}$ (0)
Character Distribution	0.5 (0.995)
Structural Inference	10^{-6} (0.25)
Tot. Score (Thresh.)	*1.381 (0.03)*
Correlated events	opened `emonc`, unlinked `emonc.lock`

Table 5.15: False positive on `sendmail`: operations in `/var/mail`

```
open: /var/adm/utmpx, ----------
open: /var/adm/utmpx, -rw-rw-rw-
open: /var/adm/utmp, -rw-rw-rw-
open: /devices/pseudo/clone@0:udp, crw-rw-rw-
open: /etc/resolv.conf, -rw-r--r--
open: /devices/pseudo/clone@0:udp, crw-rw-rw-
open: /devices/pseudo/clone@0:udp, crw-rw-rw- [...]
open: /etc/mail/sendmail.st, -rw-rw-r--
unlink: /var/spool/mqueue/xfIAA00307
fork:
open: /devices/pseudo/mm@0:null, crw-rw-rw-
open: /devices/pseudo/mm@0:null, crw-rw-rw-
open: /var/spool/mqueue/qfIAA00307, -rw-------
setuid: 2067
exit: 0
```

A false positive happens for instance when `sendmail` uses `UID` 2133 (Table 5.14) to deliver a message. In training that particular `UID` was not used, so the model flags it as anomalous. Since this can happen in the normal behavior of the system, it is evidently a generic problem with the modeling of `UID`s as it is done in LibAnomaly.

Operations in `/var/mail` (see Table 5.15) are flagged as anomalous because the filenames are of the type `/var/mail/emonca000Sh` and thus the alternance of lower and upper case characters and numbers easily triggers the structural inference model.

5.2.6 A theoretical critique to SyscallAnomaly

In the previous section we outlined different cases of failure of SyscallAnomaly. But what are the underlying reasons for these failures? In this section we analyze all the proposed models and their weaknesses, as we found them during our analysis

Structural inference model flaws

This model, as described in 5.2.2 is probably the weakest overall. Firstly, it is too sensitive against non alphanumeric characters. Since they are not altered or compressed, the model reacts strongly against slight modifications that involve these characters. This becomes visible when libraries with variable names are opened, as it is evident in the false positives generated on the `ps` program (see Section 5.2.5).

On the other hand, the compressions and simplifications introduced are excessive, and cancel out any interesting feature: for instance, the strings `/tmp/tempfilename` and `/etc/shadow` are indistinguishable by the model.

A very surprising thing, as we already noticed, is the choice of ignoring the probability values in the HMM, turning it into a binary value (0 if the string cannot be generated, 1 otherwise). This assumes an excessive weight in the total probability value, easily causing a false alarm.

In order to test our hypothesis, we excluded this model from the SyscallAnomaly program. As can be seen in Table 5.16 the detection rate is unchanged, while the false positive rate is strongly diminished (because many of the errors outlined above disappear).

Particularly striking is the case of `ls`, which is an attack-free program, where excluding the structural inference model instantly makes all the false positives disappear. Therefore, the Structural Inference model is not contributing to detection, but instead it is causing a growth in false positive rate.

Character Distribution Model

This model is much more reliable than the former one, and contributes very well to detection. However, the model does not care about which particular character has which distribution, which can lead to attack

Program	False Positives (Syscalls)	
	With the HMM	Without the HMM
`fdformat`	1 (4)	1(4)
`eject`	1 (6)	1 (3)
`ps`	2 (10)	1 (6)
`ftpd`	2 (45)	2 (45)
`telnetd`	2 (198)	0 (0)
`sendmail`	4 (97)	4 (97)

Table 5.16: Behavior of SyscallAnomaly with and without the Structural Inference Model

paths for mimicry attacks. For instance, executing `ps -[x)` has a very high probability, because it is indistinguishable from the usual form of the command `ps -axu`.

Token Search Model

This model has various flaws. First of all, it is not probabilistic, that is, it does not save the relative probability of the different values. Therefore a token with 1000 occurrences is considered just as likely as one with a single occurrence in the whole training. This makes the training phase non resistant to outliers or attacks in the training dataset.

Additionally, since the model is applied only on fields where it has already been determined that they contain a token, the Pearson test is not useful: in fact, in all our experiments, it never had a negative result.

String Length model

The string length model works very well (too well, as we note in Section 5.4). However, the code in the version we downloaded implements different logics than what is described in [107].

5.3 Beyond SyscallAnomaly: our proposal

5.3.1 Motivations for our proposal

Our objective is to create a host based intrusion detection system which deals with the sequence and the content of system calls, improving the ideas presented in `LibAnomaly` and solving the problems we outlined. Basically, the objectives that motivated our research are:

1. Improving the reliability of the proposed models for anomaly detection on arguments, creating a correlation among the various models on different arguments of the same syscall.

2. Introducing a model of the interrelation among the sequence of system calls over time.

In order to obtain the first improvement, we introduce the concept of clustering the system calls, in order to create, inside the set of the invocations of a single system call, subsets of arguments with an higher similarity and better characterization. This idea arises from the consideration that some system calls do not exhibit a single normal behavior, but a plurality of behaviors (ways of use) in different portions of a program. For instance, as we will see in the next sections, an `open` syscall can have a very different set of arguments when used to load a shared library or a user-supplied file.

This clustering step therefore creates relationships among the values of various arguments, creating correlations (e.g. among some filenames and particular opening modes).

The second improvement can be obtained by imposing a sequence-based correlation model through a Markov Chain, thus making the system also able to detect deviations in the normal program flow. This enables the system to detect deviations in the control flow of the program, as well as abnormalities in each single call, making evident not just the single point of the attack, but the whole anomalous context that arises as a consequence.

5.3.2 Clustering of system calls

Problem statement and algorithm description

Our objective, as we stated above, is to detect, for each system call, clusters of invocation with similar arguments, and to create models on this clusters, and not on the general system call, in order to better capture normality and deviations.

We applied a hierarchical clustering algorithm in order to find these clusters [138]. Hierarchical clustering is a bottom-up technique that progressively joins "similar" elements, until it reaches a predetermined number of clusters or inter-cluster distance rises above a certain threshold. As we already noted in Section 4.3.4, any clustering technique is substantially dependent on the definition of "distance".

Defining a distance among the set of arguments of a system call is not an easy task. Some possible metrics are:

Number of occurrences: we could associate to each argument a number proportional to its frequency of occurrence. In this way, common arguments will be clustered together.

Path depth: considering just the pathname, we could use the depth of the path as a distance, fixing that `/etc/passwd` (depth 1) is more similar to `/etc/group` (still 1) than to `/usr/src/linux/Makefile` (depth 3).

Path length: same observation as above, but using the length in characters instead than the depth.

Character distribution: the distance could grow if the distribution of characters in the names is different.

File extensions: whenever extensions are used, if the extension is different the two arguments should be more distant.

Difference in flags and modes: two calls should have an higher distance if the `flag` and `mode` fields have different values.

In the following, we will better define how we compute distance in our case.

A hierarchical algorithm is conceptually very simple. It begins by assigning each of the N input elements to a singleton cluster, and computing an $N \times N$ distance matrix D. Then the algorithm progressively joins the elements i and j such that $D[i, j] = min(D)$. D is updated by substituting i and j rows and columns with the row and column of the distances between the newly joined cluster and the remaining ones. The time complexity is roughly $O(N^2)$.

Distance between clusters can be defined in three ways:

single-linkage if the distance between two clusters is the *minimum* distance between an element of the first cluster and an element of the second cluster;

complete-linkage if the distance between two clusters is the *maximum* distance between an element of the first cluster and an element of the second cluster;

average-linkage if the distance between two clusters is the *average* distance between an element of the first cluster and an element of the second cluster.

Program Name	`open` syscall %
fdformat	92.42%
eject	93.23%
ps	93.62%
telnetd	91.10%
ftpd	95.66%
sendmail	86.49%
samba	92.72%

Table 5.17: Percentage of `open` syscalls in the IDEVAL dataset

Syscall	Occurrences
open1	100
open2	88
open3	21

Table 5.18: Relative frequencies of three `open` syscalls

Experiments on the open syscall

Our first experiments focused on the `open` system call, which is the most common one in the IDEVAL dataset, as can be seen in Table 5.17. Indeed, `open` is probably the most used system call, since it opens a file or device in the file system and creates an handle (descriptor) for further use. Open has three parameters: the file path, a set of flags indicating the type of operation, e.g. read-only, read-write, append, create if non existing, etc. (the complete list of flags is specified in `/usr/include/bits/fcntl.h`), and optionally an opening mode, which specifies the permissions to set in case the file is created.

In order to apply a hierarchical clustering algorithm and meaningfully discover clusters of similar uses of the `open` syscall, we need to define a distance function among elements as well as a stop criterion.

	open1	open2	open3
open1	0	0.75	5
open2	0.75	0	4.24
open3	5	4.24	0

Table 5.19: Distances obtained by the example in Table 5.18

For our first experiments, we used tentatively a distance composed by the sum of the following metrics:

- A fixed contribute for each argument which has a different value.
- A distance value assigned on the basis of relative occurrence of system calls. The distance among occurrence is normalized and multiplied by a constant, which is a parameter. For an example, observe Table 5.18: the maximum distance is 79 (among `open1` and `open3`), which will be used as a normalization factor. If we choose 5 as our scale parameter, the distances will be as shown in Table 5.19.
- A contribute based on path length (normalized against the maximum difference in path length).
- A fixed contribute based on the difference in path depth.
- A contribute based on the path, comparing the names of corresponding directories. For instance, comparing `/usr/local/bin` against `/usr/local/lib`, we compare `usr` $\Leftrightarrow$ `usr`, `local` $\Leftrightarrow$ `local`, and `bin` $\Leftrightarrow$ `lib`, obtaining a distance equal to 1 (the number of non corresponding elements), which can be multiplied for an arbitrary weight constant.
- A fixed contribute if the file extensions are different.
- A contribute proportional to character distribution, in this first example on the average value of the character ASCII code.

For visualizing the results on a simple example, here is the execution log of the `open` syscalls in `fdformat`:

```
/usr/lib/libvolmgt.so.1, -rwxr-xr-x
/usr/lib/libintl.so.1, -rwxr-xr-x
/usr/lib/libc.so.1, -rwxr-xr-x
/usr/lib/libadm.so.1, -rwxr-xr-x
/usr/lib/libw.so.1, -rwxr-xr-x
/usr/lib/libdl.so.1, -rwxr-xr-x
/usr/lib/libelf.so.1, -rwxr-xr-x
/usr/platform/sun4u/lib/libc_psr.so.1, -rwxr-xr-x
/devices/pseudo/mm@0:zero, crw-rw-rw-
/devices/pseudo/vol@0:volctl, crw-rw-rw-
/usr/lib/locale/iso_8859_1/LC_CTYPE/ctype, -r-xr-xr-x
```

Model	**Scaling parameter**
Difference of Arguments	+5
Number of occurrences	max +2
Different file extensions	+2
Path length	max +2
Path depth	+1 per level
Character distribution	0.5 × average

Table 5.20: Configuration of parameters used for the experiment

File	Distance from `libc.so.1`
`libintl.so.1`	7.10
`ctype`	23.18
`libc_psr.so.1`	11.99
`libw.so.1`	5.55
`libvolmgt.so.1`	8.22
`vol@0:volctl`	23.88
`mm@0:zero`	21.29
`libadm.so.1`	5.97
`libdl.so.1`	5.65
`libelf.so.1`	6.09

Table 5.21: Distances from `libc.so.1` in program `fdformat`

The scaling parameters we used for the metrics described above are reported in Table 5.20 (they are obviously the result of experimental tuning). In Table 5.21 we report, as an example, a portion of the matrix of distances.

It is easy to see that `libc.so.1` is opened in a very similar way to other libraries (among 5.65 and 11.99) and very differently than devices `vol@0:volctl` and `mm@0:zero` and localization file `ctype`; this is exactly the type of results we expected to see. The distance of library `libc_psr.so.1` (11.99) is much higher than the other libraries, because it is located in a path (`/usr/platform/sun4u/lib`) different than the others (`/usr/lib`). This factor however does not bring `libc_psr.so.1` so far away to be outside the proper cluster.

The clustering process runs with the results in Table 5.22. In the first 6 steps, the libraries are clustered together. Then the two devices join, and `libc_psr.so.1` finally lumps together with the other libraries. If

Step	Elements merged	distance
1	libw.so.1 libdl.so.1	5.097
2	libadm.so.1 libelf.so.1	5.125
3	libw.so.1, libdl.so.1 libadm.so.1, libelf.so.1	5.321
4	libc.so.1 libw.so.1, libdl.so.1, libadm.so.1, libelf.so.1	5.555
5	libintl.so.1 libc.so.1, libw.so.1, libdl.so.1, libadm.so.1, ...	6.007
6	libvolmgt.so.1 libintl.so.1, libc.so.1, libw.so.1, libdl.so.1, ...	6.122
7	vol@0:volctl mm@0:zero	7.588
8	libc_psr.so.1 libvolmgt.so.1, libintl.so.1, libc.so.1, libw.so.1, ...	8.768
9	vol@0:volctl, mm@0:zero libc_psr.so.1, libvolmgt.so.1, libintl.so.1, ...	18.070
10	ctype vol@0:volctl, mm@0:zero, libc_psr.so.1, ...	22.881

Table 5.22: Cluster generation process for `fdformat`

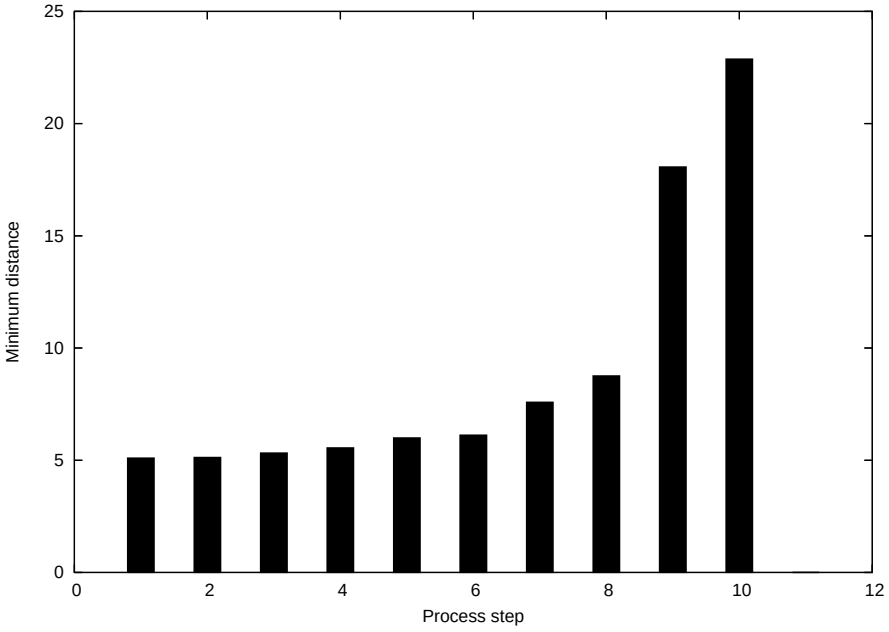


Figure 5.4: Minimum distance between clusters in function of the current step

we let the process go on further, everything ends up in one big cluster. We must thus determine a stop criterion.

As it is obvious, the distance among the elements to be merged is monotonous non decreasing. It grows slowly up to a point where a step upwards happens: this is evident in Figure 5.3.2. In this point, non-homogeneous elements are getting clustered together. This is the stop criterion we will use in Section 5.3.2.

In order to show the generality of the results, in Table 5.23 we show the clusters generated for `ps`. As we can see, also in this case the partitions make sense. Accesses in `/proc`, libraries and the device `mm@0:zero` are clustered apart. Similar results can be obtained for `eject`, `telnetd`, `ftpd`, `sendmail` and `samba`.

Computing meaningful distances

As we stated above, creating a good definition of *distance* is fundamental for a successful clustering process.

After our first experiments, we generalized the following distance measure for computing the distance among two corresponding arguments:

Cluster 1	/devices/pseudo/mm@0:zero
Cluster 2	/tmp/foo
Cluster 3	/proc
Cluster 4	/tmp/ps_data
Cluster 5	/etc/.name_service_door
Cluster 6	/usr/share/lib/zoneinfo/US/Eastern
Cluster 7	/proc/728, /proc/916, /proc/608, [...]
Cluster 8	/usr/lib/libintl.so.1, /usr/lib/libc.so.1, [...]

Table 5.23: Clusters generated for program `ps`

$$d = \begin{cases} K + \alpha\delta & \text{if the elements are different} \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

where K is a fixed quantity which creates a "step" between different elements, while the second term is the real difference among the arguments (Δ), normalized by a parameter α. How the actual difference is computed will be described below in more detail for various types of arguments.

Computation of distance among two different system calls will simply be the sum of distances among corresponding arguments

$$D_{tot} = \sum_{\forall arguments}^{i} d_i$$

Stop criterion

Since a hierarchical merging algorithm would not stop until all the inputs have been joined in a single cluster, we need to set a stop criterion for merging.

We must choose a trade-off, as always, between two conflicting requirements:

- Creating well-defined clusters that can be characterized well, which means creating more clusters with fewer elements;
- Limiting the number of clusters to avoid overfitting, which means creating fewer clusters with more elements.

Creating representative candidate models

Hierarchical clustering creates a problem for the detection phase, since there isn't a concept analogous to the concept of "centroid" in partitioning algorithms that can be used for clustering new inputs.

We thus need to generate, from each cluster, a "representative model" that can be used to cluster further inputs. This is a well known problem which needs a creative solution. For each identified type of argument, we developed a model that can be used to this end.

These models must be able to associate a *probability* to inputs, i.e. generate a *probability density function* that can be used to state the probability with which the input belongs to the model. In most cases, as we will see, this will be in the form of a discrete probability, but more complex models such as HMMs will also be used. Moreover, a concept of distance must be defined among the model and the input.

The model must be able to "incorporate" new candidates during training, and to slowly adapt in order to represent the whole cluster.

It is important to note that it's not strictly necessary for the candidate model, and its distance functions to be the same used for clustering purposes.

5.3.3 Clustering models and distances for each type of argument

As we stated above, at least 4 different types of arguments are passed to system calls:

1. Path names and file names,
2. Discrete numeric values,
3. Arguments passed to programs for execution,
4. Users and group identifiers (`UID`s and `GID`s).

For each type of argument, we created a representative candidate model and appropriate distance functions, which we describe in detail in the following sections.

Path names and file names

Path names and file names are very frequently used in system calls. They are complex structures, rich of useful information, and therefore difficult to properly model.

Some of the features we would like to capture are:

- The path, since files residing in the same branch of the file system are more similar than ones in different branches.
- Extensions, because they can indicate similitude of type.
- Common prefixes in the filename (e.g. the prefix `lib`) can be indicative.
- Inside a path, the first and the last directory carry the most significance.
- Path length is indicative of similitude.
- If the filename has a similar structure to other filenames, this is indicative.
- File system conventions (e.g. the leading dot to indicate "hidden" files in UNIX file systems) can be considered.

For the clustering phase, we chose to re-use a very simple model already present in SyscallAnomaly, the directory tree depth. This is easy to compute, and experimentally leads to fairly good results even if very simple. Thus, in Equation 5.3 we set Δ to be the difference in depth. E.g.: let $K_path = 5$ and $\alpha_path = 1$; comparing `/usr/lib/libc.so` and `/etc/passwd` we obtain $D = 5 + 1 * 1 = 6$, while comparing `/usr/lib/libc.so` and `/usr/lib/libelf.so.1` we obtain $D = 0$.

After clustering has been done, on the final clusters we can build more complex and rich models. We decided to represent the path name of the files of a cluster with a probabilistic tree which contains all the directories involved with a probability weight for each.

For instance, if a cluster contains the files: `/usr/lib/libc.so.1`, `/usr/lib/libelf.so.1`, `/usr/local/lib/libintl.so.1`, the generated tree will be as in Figure 5.5.

Filenames are usually too variable, in the context of a single cluster, to allow a meaningful model to be created. However, we chose to set up a system-wide threshold below which the filenames are so regular that they can be considered a model, and thus any other filename can be considered an anomaly.

The probability returned by the model is therefore $P_T = P_a * P_f$, where P_a is the probability that the path has been generated by the probabilistic tree and P_f is set to 1 if the filename model is not significant or if it is significant and the filename belongs to the learned set, and to 0 if the model is significant and the filename is outside the set.

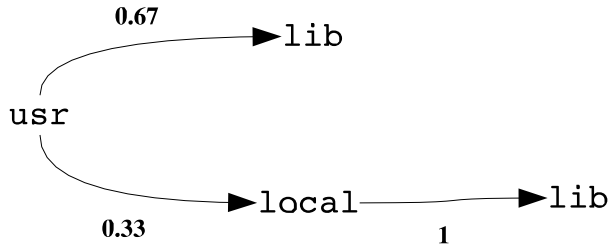


Figure 5.5: Probabilistic tree example

Discrete numeric values

Numeric values such as flags, opening modes, etc. usually are chosen from a limited set. Therefore we can memorize all of them along with a discrete probability.

Since in this case two values can only be "equal" or "different", we set up a binary distance model for clustering, where the distance among x and y is:

$$d = \begin{cases} K_d isc & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases} \tag{5.4}$$

and $K_d isc$, as usual, is a user-defined parameter.

In this case, fusion of models and incorporation of new elements are straightforward, as well as the generation of probability for a new input to belong to the model.

Execution argument

We noticed that execution arguments (i.e. the arguments passed to the `execve` syscall) are in need of special handling. Therefore we introduced an ad-hoc model to cluster and represent them, based on length. We noticed that this was an extremely good indicator of similitude of use.

Once again, we set up a binary distance model, where the distance among x and y is:

$$d = \begin{cases} K_a rg & \text{if } |x| \neq |y| \\ 0 & \text{if } x = y \end{cases} \tag{5.5}$$

denoting with $|x|$ the length of x and with $K_a rg$, as usual, a user-defined parameter. In this way, arguments with the same length are clustered together.

For each cluster, we compute the minimum and maximum value of the length of arguments. Fusion of models and incorporation of new

elements are straightforward. The probability for a new input to belong to the model is 1 if its length belongs to the interval, and 0 otherwise.

Users and groups

Many arguments express `UID`s or `GID`s, so we developed an ad-hoc model for these values. Our reasoning is that all these discrete values have three different meanings: `UID` 0 is reserved to the super-user, low values usually are for system special users, while real users have `UID`s and `GID`s above a threshold (usually 1000). So, we divided the input space in these three groups, and computed the distance for clustering using the following formula:

$$d = \begin{cases} K_u id & \text{if belonging to different groups} \\ 0 & \text{if belonging to the same group} \end{cases} \tag{5.6}$$

and $K_u id$, as usual, is a user-defined parameter.

Since `UID`s are limited in number, they are preserved for testing, without associating a discrete probability to them. Fusion of models and incorporation of new elements are straightforward. The probability for a new input to belong to the model is 1 if the UID belongs to the learned set, and 0 otherwise.

Association of models to system call arguments

In Table 5.24 we list the association of the models described above with the arguments of each of the system calls we take into account.

Validation of the models

In order to validate the models generated from cluster, we can cross-validate them by the following process:

1. Creating clusters on the training dataset;
2. Generating models from clusters;
3. Using models to classify the clusters, and checking that inputs are correctly assigned to the same clusters they contributed to create.

Table 5.25 shows, for each program in the IDEVAL dataset (considering the representative `open` system call), the percentage of inputs correctly classified, and a confidence value, computed as the probability for each element to belong to the correct cluster. The result are satisfactory, with a lower value for the `ftpd` program, which has a wider variability in filenames.

Syscall	Model used for the arguments
`open`	`pathname` → Path Name `flags` → Discrete Numeric `mode` → Discrete Numeric
`execve`	`filename` → Path Name `argv` → Execution Argument
`setuid` `setgid`	`uid` → User/Group
`setreuid` `setregid`	`ruid` → User/Group `euid` → User/Group
`setresuid` `setresgid`	`ruid` → User/Group `euid` → User/Group `suid` → User/Group
`rename`	`oldpath` → Path Name `newpath` → Path Name
`symlink` `link`	`oldpath` → Path Name `newpath` → Path Name
`mount`	`source` → Path Name `target` → Path Name `flags` → Discrete Numeric
`umount`	`target` → Path Name `flags` → Path Name
`exit`	`status` → Discrete Numeric
`chown` `lchown`	`path` → Path Name `owner` → User/Group `group` → User/Group
`chmod`	`path` → Path Name `mode` → Discrete Numeric
`creat`	`pathname` → Discrete Numeric `mode` → User/Group
`mkdir`	`pathname` → Path Name `mode` → Discrete Numeric
`mknode`	`pathname` → Path Name `mode` → Discrete Numeric `dev` → Discrete Numeric
`unlink`	`pathname` → Path Name
`rmdir`	`pathname` → Path Name

Table 5.24: Association of models to System Call Arguments in our prototype

Program	Nr. of elements	% correct assignments	Confidence
`fdformat`	10	100%	1
`eject`	12	100%	1
`ps`	525	100%	1
`telnetd`	38	100%	0.954
`ftpd`	69	97.1%	0.675
`sendmail`	3211	100%	0.996

Table 5.25: Cluster validation process

Program	Nr. of Elements	Naïve	Optimized
`ps`	880	104 MB	9 MB
`sendmail`	3450	700 MB	190 MB

Table 5.26: RAM memory reduction through our optimizations

5.3.4 Optimizations introduced on the clustering algorithm

The hierarchical algorithm as described in Section 5.3.2 is too heavy both for computation as well as for memory requirements. Besides introducing various tricks to speed up our code and reduce memory occupation (as suggested in [203]), we introduced an heuristic to reduce the average number of steps required by the algorithm.

Basically, at each step, instead of joining just the elements at minimum distance d_min, also all the elements that are at a distance $d < \beta d_min$ from both the elements at minimum distance are joined, where β is a parameter of the algorithm. In this way, groups of elements that are very close together are joined in a single step, making the algorithm (on average) much faster, even if worst-case complexity is unaffected.

Tables 5.26 and 5.27 indicate the optimization results.

Program	Nr. of Elements	Naïve	Optimized
`fdformat`	11	0.14" (0.12")	0.014" (0.002")
`eject`	13	0.24" (0.13")	0.019" (0.003")
`ps`	880	19'52" (37")	7" (5")
`sendmail`	3450	unable to complete	7'19" (6'30")

Table 5.27: Execution time reduction through our optimizations and use of the heuristic

5.3.5 Adding correlation: introduction of a Markov model

Model description and comparison with related works

In order to take into account the execution context of each system call, we decided to use a first order, observable Markov model to represent the program flow. The model states represent the system calls, or better, each cluster of each system call, as detected during the clustering process. For instance, if we detected three clusters in the `open` syscall, and two in the `execve` syscall, then the model will be constituted by five states: $\texttt{open}_1$, $\texttt{open}_2$, $\texttt{open}_3$, $\texttt{execve}_1$, $\texttt{execve}_2$. Each transition will reflect the probability of passing from one of these groups to another through the program.

This simple, short range correlation model isn't new by itself, and was proposed, although not fully explored, e.g in [204] and later analyzed in [104]. Alternatively, other authors proposed to use static analysis, as opposed to dynamic learning, to profile a program normal behavior. For instance, this approach was presented in [101] using syscall call graphs, in [99] using deterministic finite-state automata, and in [100] using non-deterministic FSA. Giffin et al. [205] developed a different version of this approach, based on the analysis of the binaries, and integrating the execution environment as a model constraint.

In [104] Hidden Markov models are compared with various other representations [89, 93, 96] and shown to perform considerably better, even if with an added computational overhead. In [103] they are observed to perform considerably better than static analysis models.

Making the models observable dramatically decreases this overhead, as observed in [206]. The same article introduces considerations which have deep affinities to the considerations we made in Section 5.1.

Training phase

During training, we will consider each execution of the program in the training set (making use of the PID value) as a sequence of observation, and train the model on these sequences.

The clustering process will already have generated the clusters for each system call, and the corresponding models. These models are used to classify each syscall into the correct cluster, by computing the probability value for each model and choosing the cluster whose models give out the maximum composite probability $\max(\prod^{i}_{\forall\ model} P_i)$.

The probabilities of the Markov model are then straightforward to compute. The final result can be similar to what is shown in Figure 5.6.

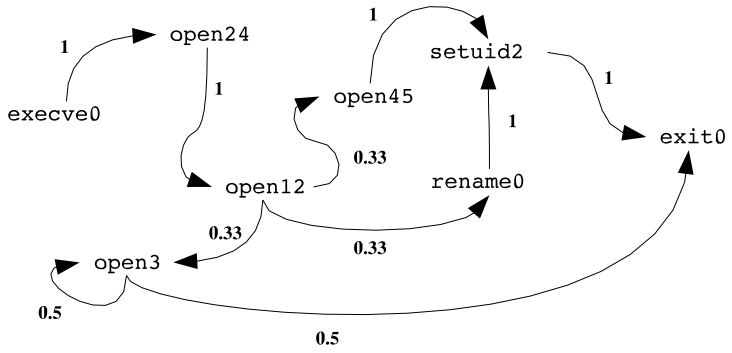


Figure 5.6: Example of Markov model

In future extensions of this work, the Markov model could then be simplified through a merging procedure, as outlined in Section 5.1.6. From our experiments, in the case of the simple traces of the IDEVAL dataset, this step is unnecessary.

Learning the anomaly thresholds

For detection, we have three distinct anomaly probabilities to take into account:

- The probability of the execution sequence inside the Markov model up to now, P_s;
- The probability of the syscall to belong to the best-matching cluster (the one it has been assigned to), P_c;
- The latest transition probability in the Markov model, P_m.

We decided to combine the latter two probabilities into a single "punctual" probability of the single syscall, $P_p = P_c P_m$, keeping a separate value for the "sequence" probability P_s.

In order to detect appropriate threshold values, we use the training data, compute the lowest probability over all the dataset for that single program (both for the sequence probability, and for the punctual probability), and set this (eventually modified by a tolerance value) as the alert threshold.

Detection phase

During detection, each system call is considered in the context of the process. The cluster models are once again used to classify each syscall

Program name	Number of executions
`fdformat`	5
`eject`	7
`ps`	105
`ftpd`	65
`telnetd`	1082
`sendmail`	827

Table 5.28: Number of instances of execution in the IDEVAL dataset

into the correct cluster, by computing the probability value for each model and choosing the cluster whose models give out the maximum composite probability $P_c = \max(\prod^i_{\forall\ model} P_i)$. This is also the first component of the punctual probability.

P_s and P_m are computed from the Markov model, and require our system to keep track of the current state for each running process. If either P_s or $P_p = P_c P_m$ are lower than the anomaly threshold, the process is flagged as anomalous.

5.4 Questioning again the validity of the DARPA dataset

We already reported in Section 4.6 some critical evaluations of the DARPA IDEVAL dataset, focused on the network dumps. The works cited there, however, fail to take into account the host based auditing data contained in the dataset. This part of the dataset, however, is all but immune from problems.

5.4.1 Limited variability and predictability

The first problem is that in the training datasets there are too few execution instances for each software, in order to representatively model its behavior, as can be seen in Table 5.28. Of just 6 programs present, for two (`fdformat` and `eject`), only a handful of executions is available, making training unrealistically simple.

The number of system calls used is also extremely limited, making execution flows very similars. Additionally, most of these executions are similars, not covering the full range of possible execution paths of the programs (thus causing overfitting of any anomaly model).

For instance, in Figure 5.7 we have plotted the distribution of the

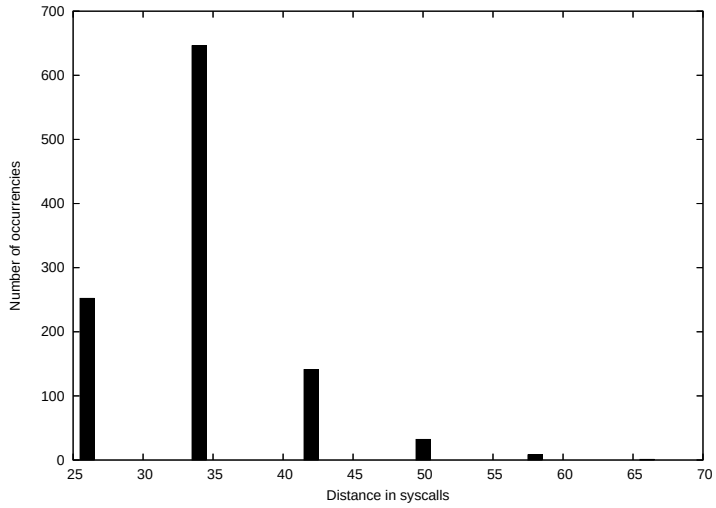


Figure 5.7: `telnetd`: distribution of distance among two `execve` system calls

distance (in system calls) among two execution of the `execve` system call in `telnetd` training data. As can be seen, in most cases a distance of 34 system calls is observed. This clearly shows how the executions of the program are sequentially generated with some script, and suffer of a lack of generality.

The arguments show the same lack of variability. In all the training dataset, all the arguments of the system calls related to `telnetd` belong to the following set:

```
fork, .so.1, utmp, wtmp, initpipe, exec, netconfig,
service_door, :zero, logindmux, pts
```

Just to give another example, the FTP operations (30 sessions on the whole) use a very limited subset of file (on average 2 per session), and are performed always by the same users on the same files, for a limitation of the synthetic generator of these operations. In addition, during training, no uploads or idle sessions were performed.

5.4.2 Outdated software and attacks

The last dataset in the IDEVAL series was created in 1999. Obviously, since then, everything changed: the usage of network protocols, the protocols themselves, the operating systems and applications used. For instance, all the machines involved are Solaris version 2.5.1 hosts, which are evidently ancient nowadays.

The attacks are similarly outdated. The only attack technique used are buffer overflows, and all the instances are detectable in the `execve` system call arguments. As we discussed before in 2.4, nowadays attackers and attack type are much more complex than this, operating at various layers of the network and application stack, with a wide range of techniques and scenarios that were just not imaginable in 1999.

5.4.3 String Length as the sole indicator

We were able to create a detector which finds all the attacks without any false positive. A simple script which flags as anomalous any argument longer than 500 characters can do this. In other words: the only meaningful indicator of attacks in the IDEVAL dataset is the length of strings.

5.5 Result analysis

For the reasons outlined above in Section 5.4, as well for the uncertainty outlined in Section 5.2.4, we do not think that purely numerical results on detection rate or false positive rate over the IDEVAL dataset are significant. We think that it is much more interesting to compare the results obtained by our software with the results of SyscallAnomaly in the terms of a set of case studies, comparing them singularly.

What turns out is that our software has two main advantages over LibAnomaly:

- a better contextualization of anomaly, which lets the system detect whether a single syscall has been altered, or if a sequence of calls became anomalous consequently to a suspicious attack;
- a strong characterization of subgroups with closer and more reliable sub-models.

As an example of the first advantage, let us analyze again the program `fdformat`, which was already analyzed in Section 5.2.5. (Table 5.5).

Our system correctly flags `execve` as anomalous (for an excessive length of input). It can be seen that transition probability is 1 (the system call is the one we expected), but the models of the syscall are not matching, generating a very low probability. The localization file opening is also flagged as anomalous for two reasons: scarce affinity with the model (because of the strange filename), and also erroneous transition between the open subgroups `open2` and `open10`. The attack effect (`chmod` and the change of permissions on `/export/home/elmoc/.cshrc`)

Anomalous Syscall	execve0 (START ⇒ execve0)
Argument 1	/usr/bin/fdformat
Argument 2	fdformat\0x20\0x20\0x20\0x20[...]
Model Probability	0.1
Transition Probability	1
Global Prob. (thresh.)	0.1 (1)
Anomalous Syscall	open10 (open2 ⇒ open10)
Argument 1	/usr/lib/locale/iso_8859_1/[...]
Argument 2	-r-xr-xr-x
Model Probability	$5 * 10^{-4}$
Transition Probability	0
Global Prob. (thresh.)	0 (*undefined*)
Anomalous Syscall	open11 (open10 ⇒ open11)
Argument 1	/devices/pseudo/vol@0:volctl
Argument 2	crw-rw-rw-
Model Probability	1
Transition Probability	0
Global Prob. (thresh.)	0 (*undefined*)
Anomalous Syscall	chmod (open11 ⇒ chmod)
Argument 1	/devices/pseudo/vol@0:volctl
Argument 2	crw-rw-rw-
Model Probability	0.1
Transition Probability	0
Global Prob. (thresh.)	0 (*undefined*)
Anomalous Syscall	exit0 (chmod ⇒ exit0)
Argument 1	0
Model Probability	1
Transition Probability	0
Global Prob. (thresh.)	0 (*undefined*)

Table 5.29: `fdformat`: attack and consequences

and various intervening syscalls are also flagged as anomalous because the transition has never been observed. This also helps us understand (while reviewing logs) whether or not the buffer overflow attack had success. A similar observation can be done on the execution of `chmod` on */etc/shadow* ensuing an attack on `eject`.

In the case of `ps`, the system flags the `execve` system call, as usual, for excessive length of input. File `/tmp/foo` is also detected as anomalous argument for `open`. In `LibAnomaly`, this happened just because of the presence of an underscore, and was easy to bypass. In our case, `/tmp/foo` is compared against a sub-cluster of `open` which contains only the `/tmp/ps_data` (see Table 5.23), and therefore will flag as anomalous, with a very high confidence, any other name, even if structurally similar. A sequence of `chmod` syscalls executed inside directory `/home/secret` as a result of the attacks are also flagged as anomalous program flows.

6 Conclusions and future work

In this work we have summarized our researches on the topic of unsupervised learning technologies and their application to the problem of intrusion detection.

We have introduced the key problems of information security, in particular the problem of making computer systems tamper evident: this gives birth to the problem of intrusion detection.

We have analyzed the different technologies and types of intrusion detection systems, the problems and the open issues to be solved, and the state of the art of the research in the field, focusing on earlier studies on the application of unsupervised learning algorithms to intrusion detection.

We have described the challenges we met while implementing an innovative model of anomaly based network intrusion detection system, completely based on unsupervised learning techniques. We have described a novel, two tier architecture for such a system. We have shown how a first tier of clustering (based on Self Organizing Maps) can perform an efficient, unsupervised pattern recognition on packet payloads. We have considered possible alternate metrics for clustering, and shown how the euclidean metric performs overall. We have also shown how the curse of dimensionality requires an appropriate resolution, and proposed various heuristics to improve the runtime efficiency of the algorithm, obtaining a throughput rate almost three times higher than the original one, with marginal misclassification rates, without truncating the number of the bytes of the payload taken into account.

We have described how we combined this first tier with a modified version of the SmartSifter outlier detection algorithm, and we have given results on the detection rate and false positive rate, showing that the system outperforms a similar, state-of-the-art system by almost an order of magnitude in term of false positive reduction. We have also studied how the errors introduced by our heuristics affect the algorithm detection capabilities, and concluded that our modified algorithm works as well as the original version of the SOM.

Future works on this system will strive to further improve its speed, as well as to reduce the false positive rate as much as possible.

We have also described our efforts to implement a host based intrusion detection system based on the sequence of system calls, as well as on their arguments. Firstly, we have introduced a general framework for behavior detection and developed an algorithm for building a Markov-based model of behavior, using concepts from the field of ethology. We have then focused on the problem of detecting anomalies in system calls, by analyzing the only existing framework which takes into account the anomalies in their arguments, by improving its models, and complementing it with a behavioral Markov model in order to capture correlation and aberrant behaviors. We have shown how the resulting model is able to correctly contextualize alarms, giving the user more information to understand what caused any false positive, and to detect variations over the execution flow, as opposed to punctual variations over single instances. The system is also auto-tuning, even if a wide range of parameters can be set by the user to improve the quality of detection.

In the course of our work, we have also outlined a number of shortcomings in the IDEVAL dataset we used in our experiments, which is still the only standard dataset for the validation and evaluation of Intrusion Detection Systems worldwide. The network data suffer of various well known problems, regularities, and characteristic flaws. These characteristics have been carefully considered, and we have tried to minimize their impact on the validity of our results. The execution traces for system call analysis are also flawed. They are too simple and predictable, they do not cover the full range of options of the programs, they are representative of a very small subset of programs and in some cases these programs are executed just a few times. This creates the conditions for overfitting of any anomaly detection algorithm. In addition, the 1999 dataset is hopelessly outdated, both because the protocols, applications and operating systems used are not representative any more of normal network usage; and also because the attack types are not representative of the modern threat scenario. We have outlined how we validated our results in order to obviate to such glaring deficiencies of the dataset.

A theme we are beginning to research on now, and which is the natural evolution of this work, is how to integrate the network and host based systems we designed, in order to use the results of both to automatically filter out false positives and to improve correlation and alert quality.

Another theme we did not deal with in this research is how much information a human operator can get from the system (aside from a generic "threat alert") and how an human expert could help refine the training of the system, with a sort of "semi-supervised" approach. These are surely interesting themes which remain open for future extensions of

this work.

Bibliography

[1] ISO. Information security code of practice. Technical Report ISO 17799, International Standards Organization, Geneva, Switzerland, 2005. revised 1999, 2000, 2002, 2005.

[2] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, J. P. Anderson Co., Ft. Washington, Pennsylvania, Apr 1980.

[3] Stefano Zanero. Detecting 0-day attacks with learning intrusion detection systems. In *Blackhat USA 2004 Briefings*, 2004.

[4] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical Report T2R-0Y6, Secure Networks, Calgary, Canada, 1998.

[5] Ross Anderson. *Security Engineering*. John Wiley & Sons, USA, 2001.

[6] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations model. Technical report, Mitre Corp., Bedford, 1975.

[7] A. Rhodes and W. Caelli. A review paper: Role based access control. Technical report, Information Security Research Centre, 1999.

[8] Gary Stoneburner. Underlying technical models for information technology security: Recommendations of the national institute of standards and technology. Technical Report NIST Special Publication 800-33, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, 2001.

[9] Franois Marie Arouet (Voltaire). *Candide ou l'optimisme*. 1795.

[10] Anthony Boswell. Specification and validation of a security policy model. *IEEE Trans. on Soft. Eng.*, 21(2):63–68, Feb 1995.

[11] Bryce. Security engineering of lattice-based policies. In *Proc. 10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

[12] E.W. Dijkstra. Notes on structured programming. In C.A.R. Hoare O.J. Dahl, E.W. Dijkstra, editor, *Structured Programming*, chapter 1, pages 1–82. Academic Press, London, 1972.

[13] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[14] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Comm. of the ACM*, 33(12):32–44, 1990.

[15] A. K. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proc. of the 1998 IEEE Symp. on Security and Privacy*, pages 104–114, 1998.

[16] Internet security threat report, vol. viii. Technical report, Symantec Corporation, September 2005.

[17] Bruce Schneier. Locks and full disclosure. *IEEE Security and Privacy*, 1(2):88, 2003.

[18] Jeff Bollinger. Economies of disclosure. *SIGCAS Comput. Soc.*, 34(3):1–1, 2004.

[19] ISO. Risk management – vocabulary – guidelines for use in standards. ISO/IEC ISO/IEC Guide 73, International Standards Organization, Geneva, Switzerland, 2002.

[20] ISO. Information technology – security techniques – management of information and communications technology security – part 1: Concepts and models for information and communications technology security management. Technical Report ISO 13335–1, International Standards Organization, Geneva, Switzerland, 2004.

[21] Marcus J. Ranum. The six dumbest ideas in computer security. available online at `http://www.ranum.com/security/computer_security/editorials/dumb/`, September 2005.

[22] The 2002 csi/fbi computer crime and security survey. Technical report, Computer Security Institute – Federal Bureau of Investigations, 2002. available online at http://www.gocsi.com/press/20020407.html.

[23] ISO. Information technology – security techniques – information security incident management. ISO/IEC ISO/IEC TR 18044, International Standards Organization, Geneva, Switzerland, 2004.

[24] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[25] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[26] D. L. Lough. *A taxonomy of computer attacks with applications to wireless networks*. PhD thesis, Virginia Polytechnic Institute and State University, April 2001.

[27] Ulf Lindqvist and Erland Jonsson. How to systematically classify computer security intrusions. In *Proc. of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[28] G. Chakrabarti, A.; Manimaran. Internet infrastructure security: a taxonomy. *IEEE Network*, 16(6):13–21, Nov/Dec 2002.

[29] P. G. Neumann and D. B. Parker. A summary of computer misuse techniques. In *Proceedings of the 12th National Computer Security Conference*, October 1989.

[30] J. D. Howard. *An Analysis Of Security Incidents On The Internet, 1989–1995*. PhD thesis, Carnegie–Mellon University, April 1997. available online http://www.cert.org/research/JHThesis/Start.html.

[31] Sun Tzu. *The Art of War*. available online at `ftp://uiarchive.cso.uiuc.edu/pub/etext/gutenberg/etext94/sunzu10.zip`.

[32] The Honeynet Project. *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*. Addison-Wesley Professional, August 2001.

[33] The Honeynet Research Alliance. Know your enemy - trend analysis. Technical report, The Honeynet Research Alliance, December 2004.

[34] Thorsten Holz. A short visit to the bot zoo. *IEEE Security & Privacy*, 3(3):76–79, 2005.

[35] E. 'Aleph1' Levy. Smashing the stack for fun and profit. *Phrack magazine*, 7(49), Nov 1996.

[36] The SANS Institute. The twenty most critical internet security vulnerabilities. Version 6.01, Available online, `http://www.sans.org/top20/`, November 2005.

[37] D. A. Wheeler. Secure programming for linux and unix howto.

[38] Dorothy E. Denning. *Information warfare and security*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[39] Greg Hoglund and Gary McGraw. *Exploiting Software: how to break code*. Addison Wesley, 2004.

[40] Dixie B. Baker. Fortresses built upon sand. In *Proc. of the 1996 Workshop on New Security Paradigms*, pages 148–153. ACM Press, 1996.

[41] K-2. Admmutate. In *CanSecWest 2001 Conference*, March 2001. the tool is available online at `http://www.ktwo.ca/c/ADMmutate-0.8.4.tar.gz`.

[42] L. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A network security monitor. In *Proc. of the 1990 IEEE Symp. on Research in Security and Privacy*, pages 296–304, May 1990.

[43] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proc. 20th NIST-NCSC Nat'l Information Systems Security Conf.*, pages 353–365, 1997.

[44] Tim Bass. Intrusion detection systems and multisensor data fusion. *Comm. of the ACM*, 43(4):99–105, 2000.

[45] Y.-S. Wu, B. Foo, Y. Mei, and S. Bagchi. Collaborative intrusion detection systems (CIDS): A framework for accurate and efficient IDS. In *Proceedings of the 19th Annual Computer Security Applications Conference*, December 2003.

[46] S. Snapp, J. Bretano, G. Dias, T. Goan, L. Heberlein, C. Ho, K. Levitt, B. Mukherjee, S. Smaha, T. Grance, D. Teal, and D. Mansur. Dids: Motivation, architecture and an early prototype. In *Proc. of the 14th National Computer Security Conference*, pages 167–176, Washington, DC, October 1991.

[47] Z. Zhang, J. Li, C.N. Manikopoulos, J. Jorgenson, and J. Ucles. HIDE: a hierarchical network intrusion detection system using statistical preprocessing and neural network classification. In *Proceedings of IEEE Workshop on Information Assurance and Security*, pages 85–90, West Point, 2001.

[48] Eugene H. Spafford and Diego Zamboni. Intrusion detection using autonomous agents. *Computer Networks*, 34(4):547–570, October 2000.

[49] Jai Balasubramaniyan, Jose Omar Garcia-Fernandez, Eugene H. Spafford, and Diego Zamboni. An architecture for intrusion detection using autonomous agents. Technical Report Coast TR 98-05, Department of Computer Sciences, Purdue University, 1998.

[50] Kymie M. C. Tan, Kevin S. Killourhy, and Roy A Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In Giovanni Vigna Andreas Wespi and Luca Deri, editors, *Fifth International Symposium on Recent Advances in Intrusion Detection (RAID-2002)*, volume 2516 of *Lecture Notes in Computer Science*, pages 54–73, Zurich, Switzerland, October 2002. Springer-Verlag.

[51] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, Dec 1999.

[52] Simonetta Balsamo, Raif O. Onvural, and Vittoria De Nitto Persone. *Analysis of Queueing Networks with Blocking*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[53] Diego Catallo. Una metodologia per la valutazione delle prestazioni di sistemi di intrusion detection. Master's thesis, Politecnico di Milano, 2004. in Italian.

[54] Stefano Zanero. My ids is better than yours... or is it ? In *Blackhat Federal 2006 Briefings*, 2006.

[55] Fragroute. available online at URL `http://www.monkey.org/~dugsong/fragroute/`.

[56] G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits. In *Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS)*, pages 21–30, Washington, DC, October 2004.

[57] Thomas M. Mitchell. *Machine Learning.* McGraw-Hill Higher Education, 1997.

[58] Dana H. Ballard. *An introduction to natural computation.* Massachusetts Institute of Technology, Cambridge, MA, USA, 1997.

[59] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proc. of LISA 99*, 1999.

[60] Christopher Krügel and Thomas Toth. Using decision trees to improve signature-based intrusion detection. In *RAID*, pages 173–191, 2003.

[61] Marcus J. Ranum, Kent Landfield, Mike Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall. Implementing a generalized tool for network monitoring. In *LISA'97.*

[62] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 262–271, New York, NY, USA, 2003. ACM Press.

[63] S. Eckmann, G. Vigna, and R. Kemmerer. STATL: An attack language for state-based intrusion detection. In *Proceedings of the ACM Workshop on Intrusion Detection*, Atene, November 2000.

[64] Ludovic Me. Gassata, a genetic algorithm as an alternative tool for security audit trails analysis. In *Proceedings of RAID'98*, September 1998.

[65] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, February 1987.

[66] H. S. Javits and A. Valdes. The NIDES statistical component: description and justification. Technical report, SRI International, March 1993.

[67] M. Theus and M. Schonlau. Intrusion detection based on structural zeroes. *Statistical Computing & Graphics Newsletter*, 9:12–17, 1998.

[68] Mark Burgess, Hârek Haugerud, Sigmund Straumsnes, and Trond Reitan. Measuring system normality. *ACM Trans. Comput. Syst.*, 20(2):125–160, 2002.

[69] N. Ye and Q. Chen. An anomaly detection technique based on a chi-square statistic for detecting intrusions into information systems. *Quality and Reliability Engineering International*, 17(2):105–112, 2001.

[70] Juan Carlos Galeano, Angélica Veloza-Suan, and Fabio A. González. A comparative analysis of artificial immune network models. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 361–368, New York, NY, USA, 2005. ACM Press.

[71] Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer immunology. *Commun. ACM*, 40(10):88–96, 1997.

[72] Rebecca Gurley Bace. *Intrusion detection*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 2000.

[73] Steven A. Hofmeyr and Stephanie A. Forrest. Architecture for an artificial immune system. *Evol. Comput.*, 8(4):443–473, 2000.

[74] Xiaoshu Hang and Honghua Dai. Applying both positive and negative selection to supervised learning for anomaly detection. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 345–352, New York, NY, USA, 2005. ACM Press.

[75] Marcus J. Ranum. artificial ignorance: how-to guide. Firewall Wizards mailing list, available online at `http://lists.insecure.org/firewall-wizards/1997/Sep/0096.html`, September 1997.

[76] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[77] Christian Kreibich and Jon Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *SIGCOMM Comput. Commun. Rev.*, 34(1):51–56, 2004.

[78] Lance Spitzner. Honeypots: Catching the insider threat. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, page 170, Washington, DC, USA, 2003. IEEE Computer Society.

[79] Jake Ryan, Meng-Jang Lin, and Risto Miikkulainen. Intrusion detection with neural networks. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[80] A. K. Gosh, J. Wanken, and F. Charron. Detecting anomalous and unknown intrusions against programs. In *ACSAC '98: Proceedings of the 14th Annual Computer Security Applications Conference*, page 259, Washington, DC, USA, 1998. IEEE Computer Society.

[81] J. Ryan, L. Meng-Jane, and R. Miikkulainen. *Intrusion Detection with Neural Networks*, chapter 8. Mit Press, May 1998.

[82] H. Debar, M. Becker, and D. Siboni. A neural network component for an intrusion detection system. In *Proc. IEEE Symposium on Research in Computer Security and Privacy*, 1992.

[83] Wenke Lee and Wei Fan. Mining system audit data: opportunities and challenges. *ACM SIGMOD Rec.*, 30(4):35–44, 2001.

[84] Wenke Lee and Salvatore Stolfo. Data mining approaches for intrusion detection. In *Proc. of the 7th USENIX Security Symp.*, San Antonio, TX, 1998.

[85] T.D. Lane. *Machine Learning Techniques For The Computer Security Domain Of Anomaly Detection*. PhD thesis, Purdue University, 1998.

[86] L. Me. Genetic algorithms, a biologically inspired approach for security audit trails analysis. In *1996 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1996. short paper.

[87] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 1996. IEEE Computer Society.

[88] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, volume XIII, pages 134–144. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[89] Stephanie Forrest, Alan S. Perelson, Lawrence Allen, and Rajesh Cherukuri. Self-nonself discrimination in a computer. In *SP '94: Proceedings of the 1994 IEEE Symposium on Security and Privacy*, page 202, Washington, DC, USA, 1994. IEEE Computer Society.

[90] J. B. D. Cabrera, L. Lewis, and R.K. Mehara. Detection and classification of intrusion and faults using sequences of system calls. *ACM SIGMOD Record*, 30(4), 2001.

[91] G. Casas-Garriga, P. Díaz, and J.L. Balcázar. ISSA: An integrated system for sequence analysis. Technical Report DELIS-TR-0103, Universitat Paderborn, 2005.

[92] Intrusion Detection Using Sequences of System Calls. S. hofmeyr and s. forrest and a. somayaji. *Journal of Computer Security*, 6:151–180, 1998.

[93] Anil Somayaji and Stephanie Forrest. Automated response using system–call delays. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 2000.

[94] William W. Cohen. Fast effective rule induction. In Armand Prieditis and Stuart Russell, editors, *Proc. of the 12th International Conference on Machine Learning*, pages 115–123, Tahoe City, CA, Jul 1995. Morgan Kaufmann.

[95] Y. Chevaleyre, N. Bredeche, and J. Zucker. Learning rules from multiple instance data : Issues and algorithms. In *Proceedings of the 9th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU02)*, Annecy, France, 2002.

[96] Wei Fan, Matthew Miller, Salvatore J. Stolfo, Wenke Lee, and Philip K. Chan. Using artificial anomalies to detect unknown and known network intrusions. In *ICDM*, pages 123–130, 2001.

[97] N. Provos. Improving host security with system call policies. Technical Report 02-3, CITI, November 2002.

[98] Suresh N. Chari and Pau-Chen Cheng. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Trans. Inf. Syst. Secur.*, 6(2):173–200, 2003.

[99] C. C. Michael and Anup Ghosh. Simple, state-based approaches to program-based anomaly detection. *ACM Trans. Inf. Syst. Secur.*, 5(3):203–237, 2002.

[100] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2001. IEEE Computer Society.

[101] David Wagner and Drew Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Se-*

curity and Privacy, page 156, Washington, DC, USA, 2001. IEEE Computer Society.

[102] Dirk Ourston, Sara Matzner, William Stump, and Bryan Hopkins. Applications of hidden markov models to detecting multi-stage network attacks. In *HICSS*, page 334, 2003.

[103] Dit-Yan Yeung and Yuxin Ding. Host-based intrusion detection using dynamic and static behavioral models. *Pattern Recognition*, 36:229–243, January 2003.

[104] Christina Warrender, Stephanie Forrest, and Barak A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. pages 133–145, 1999.

[105] Anup K. Ghosh, Aaron Schwartzbard, and Michael Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 51–62, April 1999.

[106] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 255–264, New York, NY, USA, 2002. ACM Press.

[107] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In *Proceedings of the 2003 European Symposium on Research in Computer Security*, Gjvik, Norway, October 2003.

[108] G. Tandon and P. Chan. Learning rules from system call arguments and sequences for anomaly detection. In *ICDM Workshop on Data Mining for Computer Security (DMSEC)*, pages 20–29, 2003.

[109] Dave Aitel. Resilience. Available online at `http://www.immunitysec.com/resources-papers.shtml`, February 2006.

[110] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-based anomaly detection: a new approach for detecting network intrusions. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 265–274, New York, NY, USA, 2002. ACM Press.

[111] Wenke Lee, Salvatore Stolfo, and Kui Mok. Mining in a data-flow environment: Experience in network intrusion detection. In Surajit Chaudhuri and David Madigan, editors, *Proc. of the 5th Int'l Conf. on Knowledge Discovery and Data Mining*, pages 114–124, 1999.

[112] Daniel Barbará, Julia Couto, Sushil Jajodia, and Ningning Wu. Adam: a testbed for exploring the use of data mining in intrusion detection. *SIGMOD Rec.*, 30(4):15–24, 2001.

[113] D. Barbar, N. Wu, and S. Jajodia. Detecting novel network intrusions using bayes estimators. In *Proceedings of the First SIAM International Conference on Data Mining*, 2001.

[114] Anukool Lakhina, Mark Crovella, and Christophe Diot. Mining anomalies using traffic feature distributions. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 217–228, New York, NY, USA, 2005. ACM Press.

[115] Debra Anderson, Teresa F. Lunt, Harold Javitz, Ann Tamaru, and Alfonso Valdes. Detecting unusual program behavior using the statistical component of the next-generation intrusion detection expert system (nides). Technical report, Computer Science Laboratory SRI-CSL, May 1995.

[116] Paul Barford, Jeffery Kline, David Plonka, and Amos Ron. A signal analysis of network traffic anomalies. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 71–82, New York, NY, USA, 2002. ACM Press.

[117] M.V. Mahoney and P.K. Chan. Detecting novel attacks by identifying anomalous network packet headers. Technical Report CS-2001-2, Florida Institute of Technology, 2001.

[118] Matthew V. Mahoney and Philip K. Chan. Learning nonstationary models of normal network traffic for detecting novel attacks. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 376–385, New York, NY, USA, 2002. ACM Press.

[119] Dit-Yan Yeung and Calvin Chow. Parzen-Window network intrusion detectors. In *Proc. of the 16th Int'l Conf. on Pattern Recognition*, volume 4, pages 385–388, aug 2002.

[120] P. Lichodzijewski, A.N. Zincir-Heywood, and M.I. Heywood. Dynamic intrusion detection using self organizing maps. In *14th Annual Canadian Information Technology Security Symp.*, May 2002.

[121] K. Labib and R. Vemuri. NSOM: A real-time network-based intrusion detection system using self-organizing maps. Technical report, Dept. of Applied Science, University of California, Davis, 2002.

[122] L. Girardin. An eye on network intruder-administrator shootouts. In *Proc. of the Workshop on Intrusion Detection and Network Monitoring*, pages 19–28, Berkeley, CA, USA, 1999. USENIX Association.

[123] M. Ramadas, S. Osterman, and B. Tjaden. Detecting anomalous network traffic with self-organizing maps. In Giovanni Vigna, Christopher Kruegel, and Erland Jonsson, editors, *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, volume 2820, pages 36–54, Pittsburgh, PA, USA, September 2003. Springer-Verlag.

[124] M. Ramadas. Detecting anomalous network traffic with self-organizing maps. Master's thesis, Ohio University, March 2003.

[125] L. Ertoz, E. Eilertson, A. Lazarevic, P. Tan, J. Srivastava, V. Kumar, and P. Dokas. *Next Generation Data Mining*, chapter 3. MIT Press, 2004.

[126] Kenji Yamanishi, Jun ichi Takeuchi, Graham J. Williams, and Peter Milne. On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms. In *Proc. of the 6th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pages 320–324, Aug 2000.

[127] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In *RAID Symposium*, September 2004.

[128] B. C. Rhodes, J. A. Mahaffey, and J. D. Cannady. Multiple self-organizing maps for intrusion detection. In *Proceedings of the 23rd National Information Systems Security Conference*, Baltimore, 2000.

[129] Matthew V. Mahoney and Philip K. Chan. Learning rules for anomaly detection of hostile network traffic. In *Proc. of the 3rd IEEE Int'l Conf. on Data Mining*, page 601, 2003.

[130] J. Snyder. Taking aim: Target-based IDSes squelch network noise to pinpoint the alerts you really care about. *Information Security Magazine*, January 2004.

[131] M. V. Mahoney and P. K. Chan. A machine learning approach to detecting attacks by identifying anomalies in network traffic. Technical Report CS-2002-08, Florida Institute of Technology, 2002.

[132] M. V. Mahoney. Network traffic anomaly detection based on packet bytes. In *Proceedings of the 19th Annual ACM Symposium on Applied Computing*, 2003.

[133] Matthew V. Mahoney and Philip K. Chan. Learning nonstationary models of normal network traffic for detecting novel attacks. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 376–385, New York, NY, USA, 2002. ACM Press.

[134] Stefano Zanero and Sergio M. Savaresi. Unsupervised learning techniques for an intrusion detection system. In *Proc. of the 2004 ACM Symposium on Applied Computing*, pages 412–419. ACM Press, 2004.

[135] K.M.C. Tan and B.S. Collie. Detection and classification of TCP/IP network services. In *Proc. of the Computer Security Applications Conf.*, pages 99–107, 1997.

[136] Stefano Zanero. Analyzing tcp traffic patterns using self organizing maps. volume 3617 of *Lecture Notes in Computer Science*, pages 83–90, Cagliari, Italy, September 2005. Springer.

[137] J. A. Hartigan. *Clustering Algorithms*. Wiley, 1975.

[138] J. Han and M. Kamber. *Data Mining: concepts and techniques*. Morgan-Kauffman, 2000.

[139] D. Hawkins. *Identification of Outliers*. Chapman and Hall, London, 1980.

[140] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surv.*, 31(3):264–323, 1999.

[141] D. Boley, V. Borst, and M. Gini. An unsupervised clustering tool for unstructured data. In *IJCAI 99 Int'l Joint Conf. on Artificial Intelligence*, Stockholm, Aug 1999.

[142] T. Kohonen. *Self-Organizing Maps*. Springer-Verlag, Berlin, 3 edition, 2001.

[143] Sergio Savaresi, Daniel L. Boley, Sergio Bittanti, and Giovanna Gazzaniga. Cluster selection in divisive clustering algorithms. In *Proc. of the 2nd SIAM Int'l Conf. on Data Mining*, pages 299–314, 2002.

[144] R. Larsen. *Lanczos bidiagonalization with partial reorthogonalization*. PhD thesis, Dept. Computer Science, University of Aarhus, DK-8000 Aarhus C, Denmark, Oct 1998.

[145] Stefano Zanero. Improving the principal direction divisive partitioning algorithm. Technical Report TR-2006-02, Dipartimento di Elettronica e Informazione, Politecnico di Milano, January 2006.

[146] A. Likas, N. Vlassis, and J. J. Verbeek. The global k-means clustering algorithm. *Pattern Recognition*, 36(2), 2003.

[147] Sergio Savaresi and Daniel L. Boley. On the performance of bisecting k-means and PDDP. In *Proc. of the 1st SIAM Conf. on Data Mining*, pages 1–14, 2001.

[148] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[149] T. F. Cox and M. A. A. Cox. *Multidimensional Scaling*. Monographs on Statistics and Applied Probability. Chapman & Hall, 1995.

[150] I. T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 1986.

[151] S. Zanero. Improving self organizing map performance for network intrusion detection. In *SDM 2005 Workshop on "Clustering High Dimensional Data and its Applications"*, 2005.

[152] S. McCreary and K. Claffy. Trends in wide area ip traffic patterns - a view from ames internet exchange. In *Proc. of ITC'2000*, 2000.

[153] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? *Lecture Notes in Computer Science*, 1540:217–235, 1999.

[154] Alexander Hinneburg, Charu C. Aggarwal, and Daniel A. Keim. What is the nearest neighbor in high dimensional spaces? In *The VLDB Journal*, pages 506–515, 2000.

[155] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim. On the surprising behavior of distance metrics in high dimensional space. *Lecture Notes in Computer Science*, 1973, 2001.

[156] Charu C. Aggarwal. On effective classification of strings with wavelets. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 163–172, New York, NY, USA, 2002. ACM Press.

[157] The exploittree repository. available online, `http://www.securityforest.com`.

[158] The metasploit framework. available online, `http://www.metasploit.com`.

[159] J. Frank. Artificial intelligence and intrusion detection: Current and future directions. In *Proc. of the 17th Nat'l Computer Security Conf.*, Baltimore, MD, 1994.

[160] Victoria Hodge and Jim Austin. A survey of outlier detection methodologies. *Artif. Intell. Rev.*, 22(2):85–126, 2004.

[161] Jessica Lin, Eamonn Keogh, and Wagner Truppel. Clustering of streaming time series is meaningless. In *DMKD '03: Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 56–65, New York, NY, USA, 2003. ACM Press.

[162] T. Lane and C.E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Trans. on Information and System Security*, 2(3):295–331, 1999.

[163] B.-K. Yi, N. D. Sidiropoulos, T. Johnson, A. Biliris, H. V. Jagadish, and C. Faloutsos. Online data mining for co-evolving time sequences. In *Proceedings of the IEEE 16th International Conference on Data Engineering*, pages 13–22, 2000.

[164] Sergey Kirshner. *Modeling of multivariate time series using hidden Markov models*. PhD thesis, Department of Computer Science, University of California, Irvine, March 2005.

[165] T. Kailath. *Linear Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1980.

[166] Spiros Papadimitriou, Jimeng Sun, and Christos Faloutsos. Streaming pattern discovery in multiple time-series. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 697–708. VLDB Endowment, 2005.

[167] K. Yamanishi, J.-I. Takeuchi, G. J. Williams, and P. Milne. Online unsupervised outlier detection using finite mixtures with discounting learning algorithms. *Knowledge Discovery and Data Mining*, 8(3):275–300, 2004.

[168] Kenji Yamanishi and Jun ichi Takeuchi. Discovering outlier filtering rules from unlabeled data: combining a supervised learner with an unsupervised learner. In *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 389–394, New York, NY, USA, 2001. ACM Press.

[169] Isabelle Guyon and Andr Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[170] S.K. Pal and P. Mitra. *Pattern Recognition Algorithms for Data Mining: Scalability, Knowledge Discovery, and Soft Granular Computing*. Chapman Hall/CRC Press, Boca Raton, FL, May 2004.

[171] Hyunjin Yoon and Kiyoung Yang. Feature subset selection and feature ranking for multivariate time series. *IEEE Transactions on Knowledge and Data Engineering*, 17(9):1186–1198, 2005. Member-Cyrus Shahabi.

[172] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. Analysis and results of the 1999 DARPA off-line intrusion detection evaluation. In *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, pages 162–182, London, UK, 2000. Springer-Verlag.

[173] Darpa intrusion detection evaluation. available online, `http://www.ll.mit.edu/IST/ideval/data/dataindex.html`.

[174] KDD Cup '99 Dataset. available online at `http://kdd.ics.uci.edu/databases/kddcup99/kddcu99.html`.

[175] Christopher Kruegel, Thomas Toth, and Engin Kirda. Service specific anomaly detection for network intrusion detection. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 201–208, New York, NY, USA, 2002. ACM Press.

[176] Y. Tang and S. Chen. Defending against internet worms: a signature-based approach. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2005)*, volume 2, pages 1384–1394. IEEE, March 2005.

[177] The shmooo group capture the ctf project. available online, `http://www.shmoo.com/cctf/`.

[178] K. Kendall. A database of computer attacks for the evaluation of intrusion detection systems. Master's thesis, Massachussets Institute of Technology, 1999.

[179] John McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by lincoln laboratory. *ACM Trans. on Information and System Security*, 3(4):262–294, 2000.

[180] M. V. Mahoney and P. K. Chan. An analysis of the 1999 DARPA / Lincoln laboratory evaluation data for network anomaly detection. In *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, pages 220–237, Pittsburgh, PA, USA, September 2003.

[181] Stefano Zanero. Behavioral intrusion detection. In Cevdet Aykanat, Tugrul Dayar, and Ibrahim Korpeoglu, editors, *Proceedings of ISCIS 2004*, volume 3280 of *Lecture Notes in Computer Science*, pages 657–666, Kemer-Antalya, Turkey, October 2004. Springer.

[182] P. Martin and P. Bateson. *Measuring Behaviour: An Introductory Guide*. Cambridge University Press, Cambridge, UK, 2 edition, 1993.

[183] K. Z. Lorenz. The comparative method in studying innate behaviour patterns. In *Symposia of the Society for Experimental Biology*, page 226, 1950.

[184] N. Tinbergen. The hierarchical organization of nervous mechanisms underlying instinctive behaviour. In *Symposium for the Society for Experimental Biology*, pages 305–312, 1950.

[185] Mark Humphrys. Action selection methods using reinforcement learning. In Pattie Maes et al., editor, *From Animals to Animats 4: Proc. of the 4th Int'l Conference on Simulation of Adaptive Behavior*, pages 135–144, 1996.

[186] A. K. Seth. Evolving action selection and selective attention without actions, attention or selection. In R. Pfeifer, B. Blumberg, J. Meyer, and S. Wilson, editors, *Proc. of SAB'98*, pages 139–147. MIT Press, 1998.

[187] G. W. Barlow. *Ethological units of behavior*, pages 217–237. Chicago University Press, Chicago, 1968.

[188] S. Jha, K. Tan, and R. A. Maxion. Markov chains, classifiers, and intrusion detection. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations (CSFW'01)*, pages 206–219, Washington, DC, USA, June 2001. IEEE Computer Society.

[189] Wen-Hua Ju and Y. Vardi. A hybrid high-order Markov chain model for computer intrusion detection. *J. of Computational and Graphical Statistics*, 10:277–295, 2001.

[190] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. In *Proc. of the IEEE*, volume 77, pages 257–286, 1989.

[191] L. E. Baum and J. A. Eagon. An inequality with applications to statistical prediction for functions of Markov process and to a model of ecology. *Bull. American Math. Soc.*, pages 360–363, 1967.

[192] N. Merhav, M. Gutman, and J. Ziv. On the estimation of the order of a Markov chain and universal data compression. *IEEE Trans. Inform. Theory*, 35:1014–1019, Sep 1989.

[193] P. Haccou and E. Meelis. *Statistical analysis of behavioural data. An approach based on timestructured models.* Oxford university press, 1992.

[194] Yiu-Ming Cheung and Lei Xu. An RPCL-based approach for Markov model identification with unknown state number. *IEEE Signal Processing Letters*, 7:284–287, Oct 2000.

[195] L. Baum. An inequality and associated maximization technique in statistical estimation for probalistic functions of Markov processes. *Inequalities*, pages 1–8, 1972.

[196] J. B. Moore and V. Krishnamurthy. On-line estimation of hidden Markov model based on the Kullback-Leibler information measure. *IEEE Trans. on Signal Processing*, pages 2557–2573, August 1993.

[197] B.-H. Juang and L. Rabiner. A probabilistic distance measure for hidden Markov models. *AT&T Technical Journal*, 64:391–408, 1985.

[198] Andreas Stolcke and Stephen Omohundro. Hidden Markov Model induction by bayesian model merging. In *Advances in Neural Information Processing Systems*, volume 5, pages 11–18. Morgan Kaufmann, 1993.

[199] A. Stolcke and S. M. Omohundro. Best-first model merging for hidden Markov model induction. Technical Report TR-94-003, 1947 Center Street, Berkeley, CA, 1994.

[200] I. Ren J.A. te Boekhorst. Freeing machines from Cartesian chains. In *Proceedings of the 4th International Conference on Cognitive Technology*, number 2117 in LNCS, pages 95–108. Springer-Verlag, Aug 2001.

[201] Andreas Stolcke and Stephen M. Omohundro. Inducing probabilistic grammars by bayesian model merging. In *Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, pages 106–118, London, UK, 1994. Springer-Verlag.

[202] LibAnomaly Project. available online at `http://www.cs.ucsb.edu/~rsg/libAnomaly`.

[203] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[204] Andrew P. Kosoresow and Steven A. Hofmeyr. Intrusion detection via system call traces. *IEEE Softw.*, 14(5):35–42, 1997.

[205] Jonathon T. Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P. Miller. Environment-sensitive intrusion detection. In *RAID*, pages 185–206, 2005.

[206] S. Jha, K. Tan, and R. A. Maxion. Markov chains, classifiers, and intrusion detection. In *CSFW '01: Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, page 206, Washington, DC, USA, 2001. IEEE Computer Society.

Index