Università degli Studi di Cagliari

Facoltà di Ingegneria

Dipartimento di Ingegneria Elettrica ed Elettronica

Corso di Dottorato in Ingegneria Elettronica ed Informatica

XIX Ciclo

Statistical Pattern Recognition Techniques for Intrusion Detection in Computer Networks

Challenges and Solutions

Tesi di Dottorato di

Dott. Ing. Roberto Perdisci

Relatore

Prof. Ing. Giorgio Giacinto

This work was developed in part at the Department of Electrical and Electronic Engineering, University of Cagliari, Italy, and in part at the Georgia Tech Information Security Center, College of Computing, Georgia Institute of Technology, Atlanta, GA, USA.

Alla mia famiglia

Abstract

Intrusion Detection Systems (IDS) aim at detecting and possibly preventing the execution of attacks against computer networks, thus representing a fundamental component of a network defence-in-depth architecture. Designing an IDS can be viewed as a *pattern recognition* problem. Pattern recognition techniques have been proven successful in learning concepts from example data and constructing classifiers that are able to classify new data with high accuracy. In network intrusion detection the main objective is to design a classifier that is able to distinguish between normal and attack traffic, therefore several researchers have used statistical pattern recognition and related techniques to accomplish this task showing promising results.

We explore several aspects of the application of statistical pattern recognition to network intrusion detection from a practitioner point of view. Our intent is to point out significant challenges and possible solutions related to designing statistical pattern classification systems for network intrusion detection. In particular, we discuss three problems: a) Learning from unlabeled traffic; b) Learning in adversarial environment; c) Operating in adversarial environment. Because of the difficulties in constructing labeled datasets of network traffic, *unlabeled learning* techniques have recently been proposed to construct anomaly-based network IDS. In this case, the traffic used for learning is usually directly extracted from the live network to be protected and does not undergo any labeling process. Unfortunately, learning from unlabeled data is inherently difficult. As a consequence *unlabeled anomaly IDS* suffer from a relatively high number of false positives. We propose a new unlabeled anomaly IDS based on a modular Multiple Classifier System (MCS), and show that the proposed approach improves the accuracy performance compared to "monolithic" IDS proposed by other researchers. As the network traffic used for training unlabeled IDS is directly extracted from the network, an adversary may try to pollute the training dataset by sending properfly crafted traffic to the protected network. The adversary's objective is to modify the distribution of training data, so that future attack instances will not be detected. Using a case study we show that such attacks are possible in practice and then we discuss possible coutermeasures. Also, assuming the adversay does not interfere with the learning process, she may try to "evade" the IDS during the oprational phase. We show how an attack may be transformed to blend in normal traffic, thus reaching the protected network unnoticed. Afterwords, we present a strategy which aims to harden anomaly-based network IDS by combining multiple classifiers, making *blending attacks* more difficult to succeed.

Acknowledgments

There are many people I'd like to thank for supporting me during my doctoral studies. First I'd like to thank my advisors, Prof. Giorgio Giacinto, Prof. Fabio Roli, and Prof. Wenke Lee. They are definitely great scientists and great people. They guided me and taught me how to think like an ambitious and effective researcher. They also gave me the possibility to spend half of my studies at the Georgia Institute of Technology, and I will be always extremely grateful for that.

Studying both at the University of Cagliari and at Georgia Tech gave me the opportunity to be part of two great research groups and meet so many wonderful people. I'd like to thank all my colleagues, but I'm afraid the list would be too long. Among them, Luca Didaci, David Dagon, Prahlad Fogla, Guofei Gu, Roberto Tronci, Ignazio Pillai, Monirul Sharif, in particular, are the people with whom I had the privilege to work and exchange ideas about both research and life in general.

Special thanks go to my family. My parents have always supported me and I know they are proud of what I've done, and I hope they know I'm proud of them as much as they are of me, and that I immensely love them. The same love I feel for my sister, my brother, my sister in law, and my nieces. This thesis is dedicated to all of them.

Also, special thanks go to my girlfriend Hannah. She supported me and tolerated me every time I was nervous and stressed out about my thesis. She has always been extremely kind and patient with me, no matter what, and I love her so much because of that.

Least but not last... I'd like to thank my closest friends, Alessandro and Natasha, Danilo, Fabrizio, Cristian, and Gianluca. With them I shared moments I'll never forget. Thank you guys!

Contents

List of Tables

List of Figures

Chapter 1

Introduction to Intrusion Detection

1.1 Computer Security

Ever since the birth of multi-level/multi-user computers in 1960s, computer security has assumed a fundamental role. A multi-level computer supports access control policies which aim to guarantee limited access to resources. The access decision is based on the classification of both the user's privileges and properties of the object the user wants to access [58, 69]. Because of the very limited or absent networking, in multi-level computers the primary concern was related to legitimated users who try to access resources without having proper authorization. Afterwards, along with the growth in popularity of personal computers and the development of the Internet in 1990s, the scenario changed and the primary concern became the potential vulnerability of computer systems in face of attacks from anonymous remote users [69].

The first known internet-wide attack and penetration occurred in November 1988. The attack was in the form of a self-propagating program that spread through the network using a variety of propagation techniques [69]. The attack was named "Morris Worm", after his author Robert T. Morris Jr. The "Morris Worm" exploited common misconfiguration in `sendmail`, a Mail Transfer Agent (MTA) software. In 1996 the online magazine *Phrack* published an article by Aleph One [79] discussing the details of how to perform intrusions by exploiting buffer-overflow vulnerabilities. Today, attacks that exploit buffer-overflow vulnerabilities (usually referred to as *buffer-overflow attacks*) are among the most common and effective, given the large number of new buffer-overflow vulnerabilities discovered every year and that a successful attack has a high probability of yielding administration privileges on the attacked machine [69]. Since early 1990s, attacks against computer systems have significantly increased in number and sophistication. At the same time, the development and release of *automatic attack* tools on the Internet has caused the skills necessary for launching computer attacks to dramatically decrease, as shown in Figure 1.1. On the other hand, whereas in the beginning attacks were performed usually as a way to prove attacker's own skills, today the motivations behind attacks against computer systems are becoming more and more criminal-driven [71]. As a consequence of computer crime, many companies have lately undergone major financial losses [38].

One of the first studies on computer security is "Computer Security Threat Monitoring and Surveillance", by J. P. Anderson, published in 1980 [11]. In his work, Anderson discussed a framework for investigation of intrusions and intrusion detection. In particular, he gave a definition of fundamental terms like *Threat*, *Risk*, *Vulnerability*, *Attack*, and *Penetration*, from a computer security perspective. The definition of these terms as given by Anderson in [11] are reported below:

Threat: The potential possibility of a deliberate, unauthorized attempt to:

(a) Access information

(b) Manipulate information

(c) Render a system unreliable or unusable

Risk: Accidental or unpredictable exposure of information, or violation of operations integrity due to malfunction of hardware or incomplete or incorrect software design.

Vulnerability: A known or suspected flow in the hardware or software design or operation of a system that exposes the system to penetration or its information to accidental disclosure.

Attack: A specific formulation or execution of a plan to carry out a threat.

Penetration: A successful attack; the ability to obtain (unauthorized) access to files and programs or the control state of a computer system.

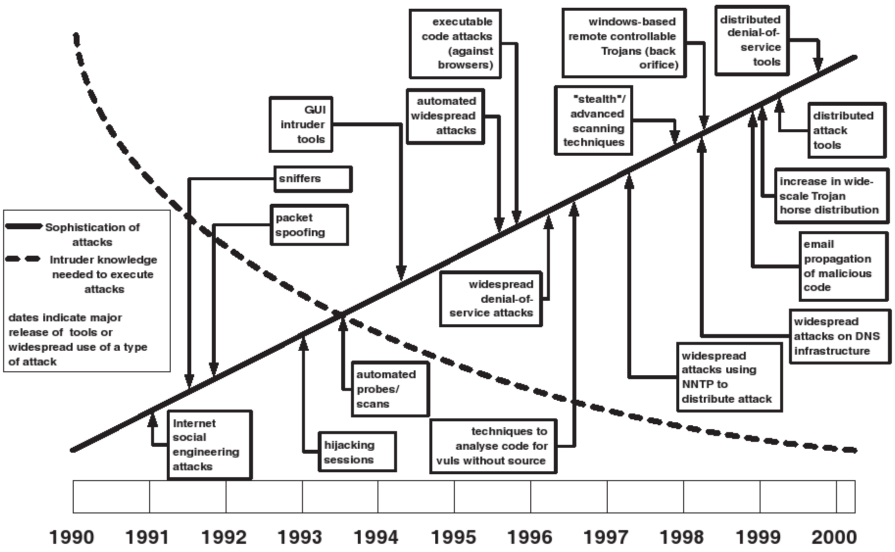

Figure 1.1: The evolution of attack sophistication and devolution of attackers' skills [69].

1.2 Intrusion Detection

According to Anderson's terminology, intrusion detection aims at detecting and possibly preventing the execution of both penetrations (or intrusions) and attacks, i.e., both successful and unsuccessful attacks[1]. In the beginning, intrusion detection was performed through manual analysis of audit records by security experts. This was particularly important for early computer-assisted financial transactions and for the protection of military systems. Afterwards, because of the growth of the volume of financial transactions and military information handled by computer systems, it was evident that an automatic way to perform security audit was needed. Dorothy Denning's paper "An Intrusion Detection Model" [25] is one of the first and most influential papers on intrusion detection. Denning assumes the attacks are distinguishable from normal users' behavior. Therefore, the main task for constructing an effective Intrusion Detection System (IDS) is to find an appropriate way of modeling the normal activities and suitable metrics to measure the distance between attacker's activities and the model of normal behavior. Denning's work has inspired many researchers and represents the base for several commercial products [71].

[1] Although detecting intrusions is the most important objective, gathering information about unsuccessful attacks is in general useful as well because it allowes security experts to estimate where the major threats come from and what are the targeted machines in the protected network

1.2.1 Misuse-based vs. Anomaly-based Intrusion Detection

The intrusion detection problem can be viewed as an instance of the general signal-detection problem [71]. *Intrusive activities* can be viewed as the signal to be detected whereas *normal activities* can be considered to be noise. In classical signal-detection approaches, the distribution of both the signal and the noise is known or approximated. The decision process consists in distinguishing between noise and signal-plus-noise in the communication channel. Unlike classical signal-detection, in which both a model of the noise and the signal are used to make a decision, IDS typically base their decision on either a model of the signal (i.e., a model of attack activities) or a model of the noise (i.e., the normal behavior). IDS which construct a model of the attacks to make a decision are usually referred to as *misuse-based* IDS. Misuse-based IDS often use a set of rules (or signatures) as attack model, with each rule usually dedicated to detect a different attack. In this case IDS are commonly referred to as *signature-based* IDS. A rule can be as simple as a string of consecutive byte values that matches a part of a network packet sent during the execution of the attack the rule refers to. On the other hand, when a model of normal activities is used to make a decision, IDS are usually referred to as *anomaly-based* IDS.

Despite the anomaly-based approach was the first to be introduced by Denning in [25], signature-based IDS are the most deployed. This is in part due to the fact that signature-based IDS are effective in detecting known attacks and usually produce less false alarms (i.e., a normal event erroneously flagged as attack) compared to anomaly-based IDS. An example of widely deployed signature-based IDS is `Snort` [90], an open source software project that has received a lot of attention during the last few years. However, although signature-based IDS have been proven to be quite effective, they are inherently unable to detect unknown attacks or even new variants of known attacks [71]. Moreover, the signature-generation process is usually a slow (semi-)manual process. This means that a window of vulnerability exists even after the attack is released and brought to the attention of the computer security research community. Conversely, anomaly-based IDS are in theory able to detect any know or unknown attack. However, designing a model of normal behavior and suitable metrics that allow to clearly separate attacks or anomalous activities from normal activities is in general non trivial and the resulting IDS are usually prone to false alarms.

1.2.2 Host-based vs. Network-based Intrusion Detection

Based on the type of events or data the IDS analyze in order to detect intrusions, it is also possible to distinguish between host-based and network-based IDS. Host-based IDS are installed on and protect single hosts, usually by inspecting system log data. For example, the audited system log data may be sequences of system calls[2] [114]. Host-based IDS can also monitor single applications. For example, an host-based IDS protecting a web server may monitor the logs produced by the *http server* software looking for anomalous http request patterns. On the other hand, network-based IDS analyze packets crossing an entire network segment. Network-based IDS have the advantage of being able to protect a high number of hosts at the same time. However, they can suffer from performance problems due to the large amount of traffic they need to analyze in real-time and possible attacks that exploit ambiguities in network protocols and cause the exhaustion of the memory and computational resources of the IDS [39]. Furthermore, network-based IDS cannot easily monitor encrypted communications and are inherently unable to monitor intrusive activities that do not produce externally (with respect to a single host) observable evidence. On the other hand, host-based IDS have access to detailed information on system events but may be disabled or made useless by an attacker who successfully gains administrative privileges on the protected machine. Intrusions that bring to the installation of so called *root kits*[3] [49] are an example of such attacks. Once the *root kit* is installed, it is usually possible for the attacker to cover the traces of malicious activities by, for example, cleaning the system logs, hiding information about malicious processes at the kernel level, etc.

1.2.3 Practical Aspects of Intrusion Detection

IDS must be considered as just one piece of the defense-in-depth strategy of an organization. A precise plan is needed before the deployment of any detection sensor. In particular, decisions have to be made regarding where to place the sensors in order to maximize the security of critical network assets, how to configure the IDS so that the general security policies of the organization are

[2] System calls are calls to functions provided by the operating system kernel

[3] A root kit is a piece of software that installs itself as part of the operating system kernel and is able to hide traces of anomalous system activities

respected, and how to react to alarms raised by the IDS [71].

In case of large networks, it may be necessary to deploy multiple IDS to protect different segments of the network. Moreover, given the complementarity of network-based and host-based IDS, deploying multiple IDS of different types may help in raising the bar against network intrusions. However, managing multiple IDS requires a significant effort. Furthermore, although collecting and correlating information from multiple sensors would help in constructing a thorough view of the network, this process is not straightforward. Some work has been done on correlation of alarms generated by multiple IDS, along with information generated by firewalls, authentication servers and other devices used to enforce security [104, 105, 82]. However, alarm correlation is still in its infancy.

1.3 Attacks Against Intrusion Detection Systems

The objective of IDS is to detect attacks against machines hosted by the protected network. However, IDS themselves can be the target of attacks. If the attacker gains information about the IDS that protects the network (e.g., through social engineering, collusive employee, etc.), she may tray to attack the IDS in order to disable it or to make it unable to detect future intrusion attempts. Besides, alike many other kinds of software, commercial and research IDS often have security vulnerabilities resulting from flawed design or coding errors.

In [85] three different attacks against network-based IDS are discussed. By low-level packet forgery, it is possible to construct network traffic in order to produce *insertion*, *evesion* and *Denial of Service* (DoS) attacks. Insertion attacks exploit the fact that network IDS may accept packets that the destination machine will reject. An attacker may use insertion of unrelevant packets to prevent the signature matching algorithm of a signature-based IDS to detect an ongoing attack. Similarly, an evasion attack exploits the fact that the IDS may neglect packets that the real destination machine will accept as valid. For example, an attacker may split IP packets using overlapping fragments. Some IDS may not be able to correctly reconstruct and analyze the original IP packet, whereas the destination machine may be able to reconstruct the original IP packet correctly. If the original packet contains an attack, the destination machine may be violated while the IDS would produce a false negative (i.e., no alarm is raised). On the other hand, the objective of DoS attacks is to cause IDS' resource exhaustion. It is easy to see that reassembling fragmented packets may be a CPU and memory intensive task, given that the fragments of a same packet may arrive unordered and in different time instants. As network-based IDS have to monitor large volumes of traffic in real-time, the attacker may try to generate large amounts of *well-crafted* fragmented packets in order to cause the IDS to reach 100% of CPU usage or to consume the TCP/IP stack buffer, so that the IDS will not be able to process new packets until part of the resources (CPU or memory) are freed. This means that an DoS attack against the IDS may be used to hide an intrusion to a machine in the protected network.

Attackers may also use *polymorphism* and *metamorphism* to evade detection [40]. The objective of these techniques is to modify the code of a given attack every time the attack is launched against a new victim. This makes signature-based IDS ineffective, because the different instances of the attack do not share any common features, or the common features are not sufficient to generate an effective signature. On the other hand, anomaly-based IDS may be able to detect polymorphic attacks because they usually look sufficiently different from normal traffic. However, a number of attacks, often referred to as *mimicry attacks*, have been proposed that can evade both network-based and host-based anomaly detectors [98, 109, 34]. The idea behind mimicry attacks is to craft the attack so that it looks like "normal" from the point of view of the IDS, while still being able to exploit the targeted vulnerability.

Another class of attacks has been recently proposed against the learning phase implemented by some IDS. When machine learning techniques are implemented to construct the IDS, the attacker may try to pollute the learning data on which the IDS is trained. The attack is launched by forcing the IDS to learn from properly crafted data, so that during the operational phase the IDS will not be able to detect certain attacks against the protected network [14, 81, 77].

1.4 Outlook of this Thesis

As anticipated above, and as we hope will be clearer by the end of this thesis, designing a network IDS is a very complicated task. Current commercial and research network IDS often suffer from relatively poor accuracy in detecting attacks against the protected network. Moreover, as discussed in Section 1.3, network IDS themselves may be vulnerable to different types of attacks.

Designing an IDS can be viewed as a *pattern recognition* problem. Pattern recognition techniques have been proven successful in learning concepts from example data and constructing classifiers that are able to classify new data with high accuracy. In particular, statistical pattern recognition has been successfully applied in many fields. In network intrusion detection the main objective is to design a classifier that is able to distinguish between normal and attack traffic. Several researcher have used statistical pattern recognition and related techniques to accomplish this task (e.g., [59, 37, 31, 112]). The obtained results show that the application of statistical pattern recognition is promising and may lead to significant advances in intrusion detection.

1.4.1 Our Contribution

In this thesis we focus on the application of statistical pattern recognition techniques for the development of network-based IDS. We explore several aspects of this task from a practitioner point of view. Our intent is to point out significant challenges and possible solutions related to designing statistical pattern classification systems for network intrusion detection. In particular, we discuss three problems:

a) **Learning from unlabeled traffic**. Many pattern classifiers are constructed using a *supervised learning* approach. To this end, a dataset containing a representative number of labeled examples of both normal and (different types of) attack traffic is needed. Unfortunately, in network intrusion detection it is very hard and expensive to construct such a labeled dataset. In order to overcome this problem, *unsupervised* or *unlabeled learning* techniques have recently been proposed to construct anomaly-based network IDS. However, learning from unlabeled data is inherently difficult. As a consequence, these systems often suffer from a relatively high number of false positives, thus making the IDS unlikely to be used in real scenarios. In order to improve the performance of unlabeled anomaly-based network IDS, we propose a modular Multiple Classifier System (MCS). We discuss how such a system can be designed and we show that the proposed modular approach improves the detection accuracy, compared to unlabeled anomaly detectors proposed by other researchers.

b) **Learning in adversarial environment**. As mentioned above, the process of labeling network data is often very hard and expensive. Therefore, unlabeled learning has been proposed to overcome this problem. Usually, in unlabeled network IDS a certain amount of traffic is collected from a live network and used (after filtering) to fit a statistical model. The obtained model is then used for classification of new traffic, i.e., to distinguish between normal traffic and attacks. In the process of collecting and learning from unlabeled traffic it is fundamental to account for an adversary which may try to interfere with the IDS' learning process. The objective of the adversary is to modify the distribution of the training data used to fit the detection model, so that future attacks will passed unnoticed. We briefly discuss some theoretical scenarios in which this could happen, and present a case study in which we show how this type of attacks are possible in practice. The objective is to point out the weaknesses of the learning phase implemented by certain IDS. We also discuss some possible (partial) solutions to the problem of learning in adversarial environment, although we believe a generic and sound solution to this problem is still to be found.

c) **Operating in adversarial environment**. Assume the adversary is not able or simply chooses not to interfere with the learning process of an IDS. After the IDS has been trained and deployed, the adversary may still be able to evade the IDS. Evasive attacks have been demonstrated to be successful against both misuse-based and anomaly-based systems. In particular, as discussed in Section 1.3, a class of attacks referred to as *mimicry attacks* has been proven successful against host based anomaly detection systems. We will show that such mimicry attacks are also possible against recently proposed network IDS which are designed using statistical pattern recognition techniques. The main reason why such attacks are possible is due to the simplicity of the statistical model used by some IDS. This simplicity derives from a trade-off between the computational performance of the IDS and its accuracy. In order to make mimicry attacks less likely to succeed we propose a new network anomaly IDS based on a multiple classifier system. The proposed architecture increases the robustness of statistical based anomaly IDS, while adding low overhead compared to existing detection systems.

1.4.2 Thesis Outline

This thesis is structured as follows. In Chapter 2 we introduce statistical pattern recognition techniques and briefly report how researchers have so far applied them to network intrusion detection. Supervised and unsupervised learning approaches are briefly presented, and the reasons why unsupervised or unlabeled learning approaches seem to be more suitable for addressing the intrusion detection problem are discussed. In Chapter 3 we present the accuracy challenges to be faced when designing an anomaly detection system based on unlabeled learning approaches. In particular, we discuss the *base rate fallacy* problem and propose a modular approach based on Multiple Classifier Systems (MCS) for improving the accuracy performance of unsupervised anomaly detectors. Chapter 4 presents the problem of learning in adversarial environment. We discuss how the attacker may, in theory, mislead a learning algorithm in order to evade the resulting detection model. We also present a case study, showing how such kind of *misleading attacks* can be successfully devised and launched in practice. Possible ad-hoc solutions to misleading attacks are discussed, although a generic solution remains an open research problem. Chapter 5 presents an attack called *Polymorphic Blending Attack*. The Polymorphic Blending Attack is a mimicry attack devised against anomaly detection systems which use simple payload statistics in order to construct a model of normal network traffic. The objective of the attacker is to transform a generic attack into a polymorphic variant which looks like normal traffic from the point of view of the IDS, yet maintaining the same attack semantic. The Polymorphic Blending Attack points out the difficulties related to operating in adversarial environment. As a possible solution to the Polymorphic Blending Attack we propose an anomaly detector constructed by using an ensemble of one-class SVM classifiers, which makes the attack less likely to succeed. Then, we briefly conclude in Chapter 6.

Chapter 2

Pattern Recognition for Network Intrusion Detection

This chapter is divided in two parts. In the first part we briefly and informally introduce *Pattern Recognition*, with particular focus on *Statistical Pattern Classification*, whereas in the second part we discuss the most significant work on the application of pattern recognition and related techniques to the problem of network intrusion detection.

2.1 Pattern Recognition

Pattern recognition studies "how machines can observe the environment, learn how to distinguish patterns of interest from their background, and make sound and reasonable decisions about the categories of the patterns" [44]. A pattern can be for example a fingerprint image, a human face, a voice signal, a text document, a network packet, etc. Pattern recognition techniques have been proven successful in learning concepts from example data and constructing classifiers that are able to classify new data with high accuracy. In particular, statistical pattern recognition techniques have been the most studied and applied in practice [44]. Based on the learning approach, pattern recognition techniques can be divided in *supervised* and *unsupervised*. Supervised pattern recognition approaches are able to learn concepts from labeled examples[1]. The label attached to each example

[1] In this thesis we do not discuss regression problems, we only focus on classification.

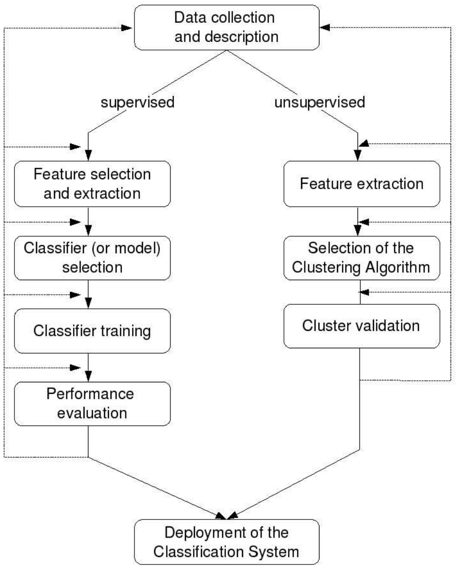
Data collection and description
supervised
unsupervised
Feature selection and extraction
Classifier (or model) selection
Classifier training
Performance evaluation
Feature extraction
Selection of the Clustering Algorithm
Cluster validation
Deployment of the Classification System

Figure 2.1: The process of designing a Pattern Recognition System

pattern represents the class (or category) the pattern belongs to. During the operational phase the supervised pattern recognition system assigns new patterns to a predefined class. On the other hand, unsupervised approaches deal with learning concepts from unlabeled data. In unsupervised classification new patterns are assigned to a hitherto unknown class [44]. An alternative approach, called *semi-supervised*, has also been proposed, whereby both labeled and unlabeled examples are used during the learning process [22].

2.1.1 Designing a Statistical Pattern Recognition System

A high level view of the process of designing a pattern recognition systems is depicted in Figure 2.1. Once the problem to be solved has been defined, the first step is to collect example data that will be pre-elaborated and fed to the learning system. In order for the collected data to be used by the pattern recognition system, a number of *features* have to be measured. The features are used to describe the patterns. For example, *minutiae* points are common features used to describe fingerprint images in fingerprint recognition [42]. As mentioned in Section 2.1, the learning problem may be addressed using the supervised or unsupervised approach.

When the supervised approach is used, the problem of understanding if all the measured features are useful has to be address. In theory, the higher the number of features, the easier to precisely distinguish between patterns belonging to different classes. In practice, when a high number of features is used and a limited number of example patterns is available for learning, the curse of dimensionality problem may arise and the performance of the recognition system may degrade [29, 44]. Feature selection and extraction techniques aim at reducing the dimensionality of the feature space in order to improve the accuracy of the classifier. Afterwards, a suitable learning algorithm has to be chosen. The training and test phase allow the designer to construct and estimate the accuracy of a classifier. As deciding a priori that a certain learning algorithm is the most suitable for the problem at hand is usually not easy, model selection is performed by constructing several classifiers using different learning algorithms, comparing the performance of each of them and choosing the one that performs the best [57]. If the results are not satisfactory, it may be necessary to go back and redesign part of the recognition system in order to improve the performance, for example by using different feature reduction techniques or a new learning algorithm.

Clustering algorithms are usually applied when the available example patterns are not labeled. Clustering aims at identifying and grouping patterns that are close to each other according to a certain metric [43]. The results of the clustering process are then usually validated by an expert on the problem at hand. A cluster may be representative of a formerly unknown class of patterns. For example, clustering is often applied in marketing analysis in order to discover classes (or groups) of customers that have meaningful characteristics (from the point of view of marketing analysts) in common.

2.1.2 Multi-Class Pattern Classification

A pattern classifier can be viewed as a function

$$C : \mathbb{R}^n \mapsto \Omega, \quad \Omega = \{\omega_1, \omega_2, .., \omega_l\} \tag{2.1}$$

where $\omega_i, i = 1, .., l$ represent the possible classes. Given a pattern $\mathbf{x}$, the classifier C assigns it to a class $C(\mathbf{x}) = \omega^* \in \Omega$. Given a dataset of labeled patterns $\mathrm{D} = \{(\mathbf{x}_1, L(\mathbf{x}_1)), (\mathbf{x}_2, L(\mathbf{x}_2)), .., (\mathbf{x}_m, L(\mathbf{x}_m))\}$, where $L : \mathbb{R}^n \mapsto \Omega$ is a (unknown) function that assigns a pattern $\mathbf{x}_i$ to its *true class* ω_{x_i}[2], supervised learning algorithms aim at constructing a classifier C that can correctly classify new patterns. We say that C is trained on D.

Once C has been trained, a generic new pattern $\mathbf{z} \notin \mathrm{D}$ can be classified using C. As an intermediate step towards the classification of $\mathbf{z}$, C computes the *supports* or *scores* $\mu_i(\mathbf{z}), i = 1, .., l$, where each $\mu_i(\mathbf{z})$ represents how strongly C believes that $\mathbf{z} \in \omega_i$. Afterwards, a decision rule is applied on the supports $\mu_i(\mathbf{z}), i = 1, ..l$, in order to assign a label to the pattern $\mathbf{z}$. Typically, the decision rule is

$$\arg\max_{i=1,..,l}(\mu_i(\mathbf{z})) = k \quad \Rightarrow \quad \mathbf{z} \in \omega_k \tag{2.2}$$

2.1.3 One-Class Pattern Classification

Most supervised learning algorithms work well when the training dataset is *balanced*, i.e., when it contains approximately the same number of examples from each class. In the presence of unbalanced datasets, techniques such as undersampling of the most represented class, oversampling of the least represented class, and other similar techniques are usually applied [45, 15]. However, in case of two-class problems for which one of the classes of objects is well-sampled, whereas the other one is severely undersampled or not represented (e.g., due to the fact that it is too difficult or expensive to obtain a significant number of training patterns for that class), resampling the dataset might not improve the performance and might not even be possible. In this cases, *one-class classification* approaches may be applied[3].

One-class classifiers aim at constructing a decision surface around the example patterns belonging to the most represented class while leaving out the patterns belonging to the least represented one [100]. The goal is to distinguish between a set of *target objects* and all the other possible objects, referred to as *outliers*. Therefore, during the operational phase, if a new patter $\mathbf{z}$ lies inside the constructed decision surface it will be classified as target, otherwise it will be classified as outlier.

[2] For simplicity, we are not consider the problem of learning from *noisy* examples.

[3] *Novelty detection* and *outlier detection* are other terms used in the pattern recognition literature to refer to one-class classification

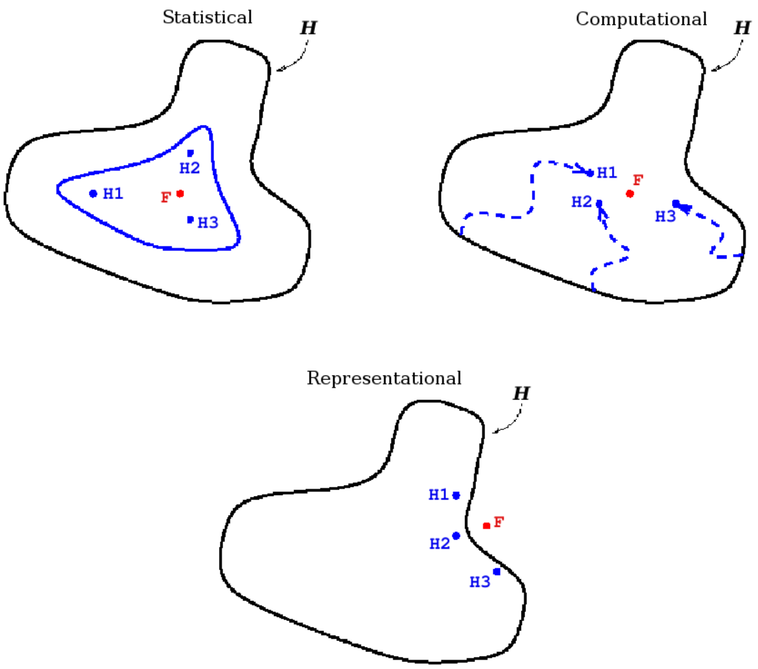

Figure 2.2: Three fundamental reasons why an MCS may work better than the single best classifier in the ensemble [28].

2.1.4 Multiple Classifier Systems

A Multiple Classifier System (MCS) is an ensemble of classifiers whose individual decisions are combined in some way to make a final decision about new patterns [28, 57]. For example, a simple and straightforward way to combine the decision of multiple classifiers is by the application of the majority voting rule [50, 57]. Assume we want to solve a two-class problem for which we constructed three different classifiers, C_1, C_2, and C_3. Let ω_1 and ω_2 be the two classes. We can construct an MCS based on the majority voting rule so that a pattern **z** is assigned to, say, class ω_1 if at least two out of three classifiers assigned **z** to $\omega 1$, otherwise **z** is labeled as belonging to ω_2.

It has been shown in many applications that MCS are much more accurate than any of the single classifiers they combine. As discussed by Dietterich in [28], in order for the MCS to perform better than the single classifiers in the ensemble, the combined classifiers need to be accurate and diverse, in the sense that they need to perform better than the random labeling algorithm and make different errors on new patterns. Dietterich [28] explained three reasons why accuracy and diversity are desired (see Figure 2.2). Let $F : \mathbb{R}^n \mapsto \Omega$, be the (unknown) function that correctly assigns any pattern $\mathbf{z}$ to its true class ω_z, and assume to have n different classifiers $H_1, H_2, .., H_n$, in a hypothesis space $\mathcal{H}$, constructed so that they approximate F. The first reason given by Dietterich is statistical. A learning algorithm can be seen as a search algorithm that tries to find a function $H \in \mathcal{H}$ as close as possible to F. When the size of the training dataset D is small compared to the size of $\mathcal{H}$, the algorithm may find many different functions $\{H_i\}_{i=1..n} \in \mathcal{H}$ which all have the same accuracy on D. By combining the output of the classifiers $\{H_i\}_{i=1..n}$, the obtained MCS reduces the risk of choosing a single classifier that may have poor performance on new data. The second reason is computational. Many learning algorithms have a stochastic component and perform some sort of search with random initialization within $\mathcal{H}$. These algorithms may get stuck in a local optima. An ensemble constructed by running the learning algorithm using different initializations may provide a better approximation of F. The third reason is representational. In many real cases $F \notin \mathcal{H}$, i.e., F cannot be represented by the learning algorithm. By combining the different hypothesis $\{H_i\}_{i=1..n} \in \mathcal{H}$, it may be possible to "expand" the space $\mathcal{H}$ and find a solution which is closer to F.

2.1.5 Classification Performance Evaluation

As mentioned in Section 2.1.1, model selection is an important part of the process of designing pattern recognition systems. In order to perform model selection, we need a way to compute and compare the accuracy of different classification algorithms. The accuracy of a classifier C is defined as $(1 - P_{\text{err}})$, where P_{err} is the probability of error, i.e., the probability of a pattern $\mathbf{x}$ be misclassified by C. The accuracy can be directly estimated as the fraction of correctly classified patters. For example, assuming the classifier C has been trained using a training dataset D, given a labeled dataset of test examples $\text{T} = \{(\mathbf{z}_1, L(\mathbf{z}_1)), (\mathbf{z}_2, L(\mathbf{z}_2)), .., (\mathbf{z}_{N_t}, L(\mathbf{z}_{N_t}))\}$, with $\text{T} \neq \text{D}$, the accuracy can be computed as

$$\text{Accuracy}(C) = \frac{1}{N_t} \sum_{\mathbf{z} \in \text{T}} I(C(\mathbf{z}) = L(\mathbf{z})) \tag{2.3}$$

where I is the indication function, whereby $I(a) = 1$ if a is true, otherwise $I(a) = 0$. This method for estimating the accuracy is called *hold-out*. Another method, called *k-fold cross-validation*, is often used to estimate the accuracy when a limited amount of labeled data is available. In this case the training dataset D is divided in k portions, $\mathrm{D}_1, .., \mathrm{D}_k$, of equal size. The classifier is trained on the union of $(k-1)$ portions and the accuracy is estimated on the one portion that has not been used during training. This process is repeated k times, testing on a different portion for each iteration. The k accuracy measures are then averaged to obtain a more reliable estimate of the real accuracy.

For many classification problems the accuracy is not a suitable measure. For example, consider a two-class problem for which a class is well represented in the test set, whereas the other one is not (see Section 2.1.3). Formally, let N_1 be the number of test patters of the first class, and N_2 be the number of test patterns of the second class. In the considered example we have $N_1 \gg N_2$, and the test dataset is said to be *highly unbalanced*. Suppose now that a classifier C always classifies the patterns as belonging to the first class. In this case

$$\text{Accuracy}(C) = \frac{N_1}{N_1 + N_2} \cong 1 \tag{2.4}$$

It is easy to see that according to the estimated accuracy the classifier performs almost perfectly, although in fact it will always make a mistake on objects from the second class.

Another way to evaluate and compare the performance of a classifier C is by means of the Receiver Operating Characteristic curve (ROC) and the Area Under the ROC Curve (AUC) [17]. Assume we have again a two-class problem. We can refer to one of the classes, say ω_p, as the *positive* class, and the other, say ω_n, as the *negative* class. Let N_p be the number of patterns from class ω_p (i.e., the positive class), N_n be the number of patterns from class ω_n (i.e., the negative class), the *False Positives*, FP, be the number of patterns from the negative class that have been erroneously classified as positive patterns by C, and the *True Positives*, TP, be the number of patterns from the positive class that have been correctly classified as positive patterns. According to the definition above we can compute the false positive and true positive rate as $\frac{FP}{N_n}$, and $\frac{TP}{N_p}$ respectively.

As mentioned in Section 2.1.2, given a test pattern $\mathbf{z}$, C computes the supports $\mu_i(\mathbf{z}), i = p, n$, for the positive and negative class. A possible decision rule is

$$\mathbf{z} \in \omega_p \quad \Leftrightarrow \quad \mu_p(\mathbf{z}) > \theta \tag{2.5}$$

where θ is a constant. By varying the threshold θ it is possible to "tune" the false positive and true positive rates generated by C. The ROC curve is a two dimensional curve. The coordinates (x, y) of the points on the ROC curve represent the false positive and true positive rate generated by C using different values of the threshold θ. The ROC curve represents a good way to visualize a classifier's performance and helps in choosing a suitable operating point, i.e., a suitable value of the decision threshold θ for the classifier C for which the desired trade-off between FP and TP is attained. However, when comparing different classification algorithms it is often desirable to obtain a number, instead of a graph, as a measure of classification performance [17]. Therefore, the AUC is used as an estimate of classification performance. The highest the AUC, the better the classification performance of a classifier C. In particular, the AUC is an estimate of the probability $\mathrm{P}(\mu_p(\mathbf{z}_p) > \mu_p(\mathbf{z}_n))$, where $\mathbf{z}_p \in \omega_p$ represents a generic positive pattern, and $\mathbf{z}_n \in \omega_n$ represents a generic negative pattern. Therefore, the AUC is an estimate of the probability that the classifier scores the positive patterns higher than the negative ones [17].

Many other methods for estimating and comparing the performance of classifiers exist. We suggest the reader to refer to [51, 29, 44, 57] for a more complete discussion and for the details on estimating classification accuracy and performing model selection.

2.2 Application of Statistical Pattern Recognition to Network Intrusion Detection

As mentioned in Section 1.2.1, signature-based IDS are not able to detect really new attacks or even variants of already known attacks. This is mainly due to the fact that the *attack signatures* are usually (semi-)manually written. It is difficult for security experts to write generic signatures capable of detecting variants of attacks against a known vulnerability. Attempts to manually write such signatures may easily make the IDS prone to false alarms. Attackers are aware of this problem and are constantly developing new attack tools with the objective to *evade* signature-based IDS. For example, techniques based on metamorphism and polymorphism are used to generate instances of the same attack that look syntactically different from each other, yet retaining the same semantic and therefore the same effect on the victim [108]. In principle this problem could be solved by designing *vulnerability-specific* signatures [110] that capture the “root-cause” of an attack, thus allowing the

IDS to detect all the attacks that try to exploit the same vulnerability. This type of signatures usually works well when implemented as part of host-based IDS. However, it is difficult to implement vulnerability-specific signatures for network-based IDS due to computational complexity problems related to the high volume of events to be analyzed.

The main motivation in using pattern recognition techniques for IDS is their generalization ability, i.e., the ability to correctly classify new patterns, which may support the recognition of variants of known attacks and unknown attacks that cannot be detected by signature-based IDS. In the following we briefly report how supervised and unsupervised learning approaches have been so far applied to network-based intrusion detection.

2.2.1 Supervised Network IDS

A number of supervised learning approaches for constructing network IDS have been proposed in the literature, for example [59, 30, 74, 80, 37, 36]. In [59] Lee proposed a framework based on data mining techniques to find a suitable set of features for describing network connections and construct network IDS. Lee's analysis brought to the construction of the KDDCup'99 dataset (`http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`), which is part of the UCI KDD archive (`http://kdd.ics.uci.edu/`). In the KDDCup'99 each pattern represents a network connection which is described by 41 features. For example, the number of bytes transmitted by the source and the destination of the connection, the duration in terms of seconds, the number of connections to the same destination machine in a certain time window, are among the used features. The features where derived in order to distinguish between normal connections and a number of different computer attacks grouped in four categories. RIPPER [23], a rule learning algorithm, is used by Lee [59] to infer classification rules that distinguish between normal network connections and different types of attacks. Each RIPPER rule consists of a conjunction of conditions to be tested on values of attributes that describe network connections. In [30] Elkan presented the results of the KDDCup'99 classifier learning competition. The competition consisted in constructing a classification system capable of distinguishing between legitimate and illegitimate network connections. The dataset used for the competition was derived from the work of Lee [59] mentioned above. Among the 24 participants, the winning classification system [83] was an MCS constructed using decision trees and a combination of *bagging* and *boosting* [57]. In [74] artificial neural networks and support vector machine are compared on the task of classifying network connections, whereas [80] compares decision trees and support vector machines on the same task. In [36, 37] Giacinto et al. proposed a modular MCS for supervised misuse-based detection of network connections. Each module is responsible for detecting attacks that use a certain (group of) service(s) as attack vector. Multiple classifiers are trained on descriptions of network connections obtained using different subsets of the set of attributes (or features) proposed by Lee [59]. The output of the obtained classifiers are then combined using various techniques in order to decide if a connection is normal or if it is an attack.

In order to apply supervised learning techniques a dataset containing examples of attack patterns as well as normal traffic is needed. In 1998 and 1999 the MIT Lincoln Laboratory worked on a project funded by DARPA for evaluating IDS [65], which led to the construction of two labeled datasets of network traffic containing both normal and attack traffic. These datasets contain the traces of several weeks of traffic in the (simulated) computer network of an air force base. However, the obtained datasets, usually referred to as *DARPA 1998* and *DARPA 1999*, have been largely criticized [70, 67]. The main critique is concerned with the fact that the simulated traffic contained in the DARPA datasets cannot be considered representative of a real network traffic. In particular, the percentage of attack traffic contained in the dataset is way larger than the percentage of attack traffic expected for a real network. Moreover, a large part of the normal traffic was artificially produced using automatic scripts that cannot accurately simulate the traffic generated as a consequence of human behavior. On the other hand, creating a labeled dataset of traffic directly extracting the raw data form a real network and then analyzing it in order to assign a label to each packet or connection is very hard and expensive. Moreover, traffic traces from different networks have different characteristics, therefore the labeling process should be repeated for each new network we want to protect. For these reasons, the supervised approach is difficult to apply in the general case.

2.2.2 Unlabeled Network IDS

In order to overcome the problems related to supervised network IDS, *unsupervised* or *unlabeled* approaches for constructing anomaly-based network IDS have been recently proposed [84, 31, 55, 66, 116, 118, 112, 63, 113, 111]. Here there is an ambiguity between the terms "unsupervised" and "unlabeled" that should be clarified. In the intrusion detection literature these two terms are often used as synonyms, whereas according to the pattern recognition terminology this may create confusion. Unfortunately, we were not able to find a clear definition of the difference between the two terms and the techniques they refer to. We think a possible definition that solves the ambiguity may be the following. We refer to *unsupervised learning* in those cases when the (unlabeled) training patterns are used to find hitherto unknown classes [44], whereas we refer to *unlabeled learning* in those cases when the set Ω of possible classes is already known and we want do find a classification function $C : \mathbb{R}^n \mapsto \Omega$ using unlabeled examples. As the objective of anomaly-based intrusion detection is to distinguish between two known classes, i.e., normal and anomalous traffic, in the following we will refer to *unlabeled* anomaly detection.

In unlabeled intrusion detection, the traffic is directly extracted from the computer network to be protected and used without the necessity of a labeling process. The only *a priori* knowledge about the data is usually represented by two assumptions that usually hold in practice: a) the extracted dataset contains two classes of data, normal and anomalous traffic; b) the numerosity of the anomalous traffic class is by far less than the numerosity of the normal traffic class. In case when these assumptions are true, novelty detection, outlier-detection or one-class classification techniques can be applied to construct anomaly-based network IDS. As mentioned in Section 1.2.1, anomaly-based IDS have the advantage of being able (at least in theory) to detect both known and never-before-seen attacks. This ability supports the main motivation (discussed above) for applying pattern recognition techniques to develop intrusion detection systems.

NIDES [10] is one of the first unlabeled anomaly detection systems. It monitors TCP/UDP ports and source and destination IP addresses. NIDES builds a model of network behavior over a long-term period, which is assumed to contain a very low amount of (or no) attacks. During the operational phase, NIDES raises and allarm if a network packet significantly deviates from the normal behavior profile. In [68], Mahoney et al. proposed a nonstationary anomaly detection systems. The proposed detection systems is made up of two components, PHAD and ALAD. PHAD monitors the traffic on a packet basis. It constructs a *normal model* of 33 fields from the Ethernet, IP, and transport layer packet header. Instead of monitoring single packets, ALAD monitors TCP connections. It constructs a model of normal connections using information on source and destination IP addresses and TCP ports, TCP header flags, and *keywords*, which represent the first word on a line of input for a certain application protocol. The detection algorithm is devised in order to assign a high score to an event , i.e., an attribute having a particular value, if no novel values have been seen for a long time for that event [68].

In [84] a variant of the single-linkage clustering algorithm is used to discover outliers in the training dataset. A pattern vector represents a connection. Once the normal patterns are separated from outlier patterns, the clusters of normal data are used to construct a supervised detection model. Tests are performed on the KDDCup'99 dataset. In [31], Eskin et al. presented a geometric framework to perform anomaly detection. The patterns are mapped from a feature space F to a new feature space F' and anomalies are detected by searching for patterns that lie in sparse regions of F'. Three different classification technique are used, a clustering algorithm, a k-NN algorithm and a SVM-based one-class classifier. Experiments are performed on the UCI KDD dataset (`http://kdd.ics.uci.edu/`), both on the portion containing network traffic, and on the portion containing sequences of system-calls.

The anomaly detectors proposed in [55, 112, 113, 111] analyze the traffic on a packet basis. The focus is on detecting attack packets that carry executable code. This approach is particularly useful against buffer overflow attacks, which are frequently seen in the wild. In [55] Kruegel et al. proposed a service specific anomaly detection systems. The main idea at the base of the proposed approach is to include information about the protocol into the model of normal traffic. To this end, they describe the network packets using three (sets of) features, namely the type of request (i.e., the protocol and the type of request for that particular protocol), the packet length, and an approximation of the distribution of byte values in the payload. The histogram representing the distribution of the byte values in normal payloads is sorted in a discending order with respect to the occurrence frequency of the byte values, and split in 6 bins. An overall anomaly score is computed by combining anomaly scores computed on the type of the request from the client to the server, the length of the request and by a measure of similarity between the distribution of byte values in the packet under test and the distribution in normal traffic. A variant of the Perason's χ^2 test is used to compute this similarity. Kruegel et al. also proposed a similar model to detect web-based attacks through the analysis of URI strings [56] related to HTTP GET requests. In this case a model is learned for each specific web application hosted on a web server. The length and character distribution of the parameters passed to the web application through the URI are analyzed and a detection model is constructed in a way which is very similar to the one proposed in [55]. The authors also propose to combine this statistical model to a structural model of the URI based on Markov models. Mahoney et al. [66] proposed an anomaly detector that uses the first 48 bytes of IP packets as features to describe the network traffic. Nine different network protocols are considered and a separate model is constructed for each one of them. The detector assigns a high anomaly score to rare events.

In [116], Yang et al. proposed an anomaly detection measure called EWIND and an unsupervised pattern learning algorithm that uses EWIND to find outlier connections in the data, whereas Leung et al. [63] presented a new density-based and grid-based clustering algorithm to perform unlabeled network anomaly detection. In [118] a two-layer IDS is presented. The first layer uses a clustering algorithm to "compress" the information extracted for network packets, whereas the second layer implements an anomaly detection algorithm to classify the patterns received from the first layer.

Wang et al. [112, 113, 111] develop on the idea presented in [55] and propose a more precise way to model the distribution of byte values in the payload of normal packets. In [112, 113] they propose to consider the entire distribution of byte values without any binning. A model is trained for each different service running on different server hosts in the protected network. For each packet, the frequency of the byte values in the payload (i.e., the data portion of the packet) is measured and a simple Gaussian model is trained. The detection of anomalous packets is performed by using a *simplified* Mahalanobis distance between the packets and the model of normal traffic. As the distribution of single byte values in the payload do not extract structural information, they also propose to generalize the detection model by measuring the distribution of n-grams, i.e., sequences of n consecutive bytes in the payload. In [111] a new way to model the normal traffic is presented. The authors propose to use a Bloom filter [18] to store information about the distribution of n-grams [24] in the payload. Compared to [112, 113], the technique presented in [111] provides a way to efficiently store structural information extracted from the payload and improves the classification accuracy. The authors first propose an unlabeled learning approach, whereby a model of normal traffic is trained on a dataset of payloads which are considered mostly normal. Then they propose to improve the detector by adding a supervised learning phase to construct a model of known attacks.

Statistical models of normal traffic are also used to detect Distributed Denial of Service (DDoS) attacks. In [32], for example, the authors propose to measure the distribution of a number of fields in IP packet headers in order to create a model of normal traffic passing through a router. A statistical distribution of the values is estimated for a number of fields during a period of length W. This distribution is considered as a *normal profile* of traffic. Future traffic is compared to the obtained distributions by means of a χ^2 statistical test. Traffic which is significantly different from the normal profile is considered anomalous and possibly generate by a DDoS attack. Therefore, a response is activated in order to reduce the impact of the detected attacks (or anomalies) [32].

It is worth noting that some of the work briefly presented here will be described in more details in the next chapters.

2.2.3 Feature Extraction for Network Intrusion Detection

In the pattern recognition literature, the term “feature extraction” often refers to the process of projecting the patterns from an original feature space $\mathcal{F}$ to a new feature space $\mathcal{F}'$ with the objective of reducing the dimensionality of the feature space. In other cases the term “feature extraction” refers to the measurement of the features themselves. Unless otherwise specified, in the remainder of this thesis we refer to feature extraction as the “feature measurement” process, namely the process through which the features used to describe the patters are measured.

The feature extraction process is a fundamental part of the design of a pattern recognition system. Because the features describe the patterns to be classified, choosing the wrong features usually heavily influences the results of the learning phase, and thus the overall performance of the recognition system. The choice of what features might be the most suitable often involves a broad expertise on the problem at hand. It is worth noting that in network intrusion detection the features to be measured are strictly related to what kind of traffic the IDS is going to analyze and to what kind of attacks we want to detect, as discussed in Section 2.2.1 and in Section 2.2.2.

Chapter 3

Unlabeled Anomaly Detection

As mentioned in Section 2.2.1, the application of supervised learning approaches for designing network IDS is hampered by the problems in obtaining a representative labeled dataset of network traffic. For this reason, unlabeled anomaly detection approaches have been recently proposed, for example in [84, 31, 112, 111].

In this chapter we discuss unlabeled anomaly detection techniques that aim to learn how to distinguish between normal and attack (or anomalous) network connections from unlabeled examples. We will first give a precise definition of the problem. Afterwards, we will present some related work on the topic and some of the challenges related to unlabeled anomaly detection in general. Because learning from unlabeled is inherently hard, recently proposed unlabeled anomaly detectors tend to be prone to a high rate of false alarms. We will propose a possible solution to the unlabeled anomaly detection problem based on a modular Multiple Classifier System (MCS), and show that the proposed approach improves the classification accuracy compared to approaches proposed by other researchers.

3.1 Problem Definition

The traffic over a TCP/IP network consists of packets related to communications between hosts. The exchange of packets between hosts usually fits in the *client-server* paradigm, whereby a client host requests some information offered by a service running on a server host. The set of packets related to the communication established between the client and (the service running on) the server forms a *connection*. Each connection can be viewed as a pattern to be classified and the network-based anomaly detection problem can be formulated as follows [36]:

Given the information about connections between pairs of hosts, assign each connection to the class of either normal or anomalous traffic.

3.2 Difficulties in Unlabeled Anomaly Detection

Learning from unlabeled data is more difficult than learning from labeled data [43]. Due to the inherent difficulties in learning from unlabeled data, unlabeled anomaly detection systems usually suffer from a relatively high false positive rate. It turns out that high false positive rates cause a significant decrease in the Bayesian detection rate, as shown in the following. As we will discuss later in this chapter, it is extremely important to improve the performance of anomaly detectors in order to attain a very low false positive rate and a high detection rate at the same time.

3.2.1 The Base-rate Fallacy

The base-rate fallacy is a logical fallacy that occurs when making a probability judgment without taking into account *a priori* probabilities. As an example, consider e medical test T for a disease D which is 99% accurate, i.e., the probability $P(T = \text{positive}|D)$ that the result of T is positive given that the patient is sick, and the probability $P(T = \text{negative}|\neg D)$ that T is negative given that the patient is not sick are both equal to 0.99. Given a patient who was found to be positive to the medical test T, we want to know what is the probability $P(D|T = \text{positive})$ that the patient really suffer from the disease D. A quick (but wrong) judgment may bring us to believe that the answer is $P(D|T = \text{positive}) = 0.99$. This answer does not take into account the incidence of the disease in the population, i.e., the *a priori* probabilities. Now, let $P(D) = 10^{-4}$ be the rate of incidence of D in the population under study, and $P(\neg D) = 1 - P(D)$. We can easily compute the correct answer by means of the Bayes formula

$$P(D|T = \text{positive}) = \frac{P(T = \text{positive}|D) \cdot P(D)}{P(T = \text{positive}|D) \cdot P(D) + P(T = \text{positive}|\neg D) \cdot P(\neg D)} \tag{3.1}$$

which gives

$$P(D|T = \text{positive}) = \frac{0.99 \cdot 10^{-4}}{0.99 \cdot 10^{-4} + 0.01 \cdot (1 - 10^{-4})} = 0.0098 \tag{3.2}$$

This means that even if the the test is 99% accurate, the probability of the patient being sick is less than 1%.

The same reasoning applies to the intrusion detection problem. In [12], Axelsson presents a simple example considering a hypothetical installation that includes a few tens of computers. Let assume this computers produce 10^6 audit records per day through logging. He also hypothesis that given the limited number of computers, the number of attacks per days is limited to just a few. If we assume, say, 2 intrusions attempts per day and 10 audit record reported per intrusion, the *a priori* probabilities of an audit record being related to an attack, $P(I)$, and to normal activities, $P(\neg I)$, would be $2 \cdot 10^{-5}$ and 0.99998, respectively. Assume we deployed an IDS which monitors the audit records mentioned above. Let, $P(A|I)$ be the probability that the IDS raises an alarm given that an intrusion occurred, i.e., the detection rate, and $P(A|\neg I)$ be the probability that the IDS raises an alarm given that no intrusion occurred, i.e., the false alarm rate. The probability $P(I|A)$ that an intrusion really occurred given that the IDS raised an allarm is

$$P(I|A) = \frac{P(A|I) \cdot P(I)}{P(A|I) \cdot P(I) + P(A|\neg I) \cdot P(\neg I)} \tag{3.3}$$

It is easy to see that the factor governing the detection rate, $P(I) = 2 \cdot 10^{-5}$, is completely overwhelmed by the factor governing the false positive rate, $P(\neg I) = 0.99998$. This is what causes the fallacy to arise [12]. As an example, assume a (unrealistic) perfect detection rate, $P(A|I) = 1.0$, and a very low false positive rate, $P(A|\neg I) = 1 \cdot 10^{-5}$. In this case the Bayesian detection rate is only $P(I|A) = 0.66$, that is there is a 34% chance that no intrusion occurred even though the IDS raised an alarm. Figure 3.1 shows how the base-rate affects $P(I|A)$ for varying detection rate and false allarm rate.

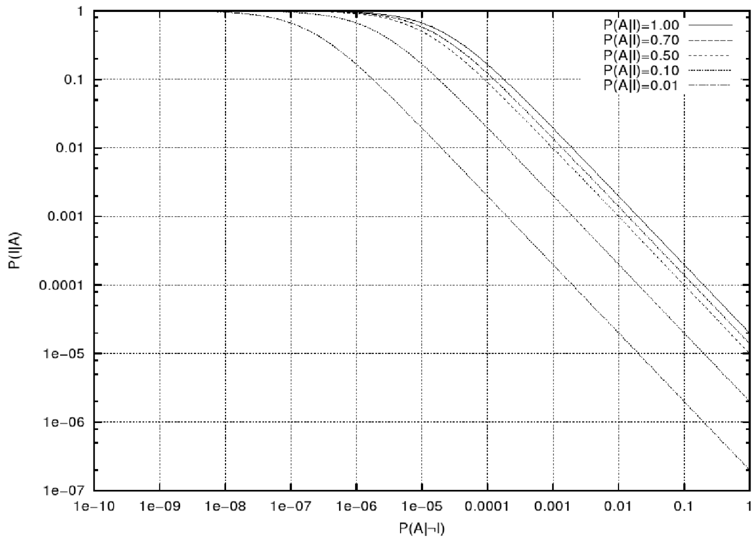

Figure 3.1: The base-rate fallacy problem [12].

In the applications we consider in this chapter we are concerned with analyzing network connections instead of system logs. The problem presented above applies to this case as well, given that we expect the number to network connections related to intrusions to be overwhelmed by the number of normal connections. The only way to mitigate the base-rate problem seems to be the improvement of the classification accuracy of the IDS in order to make it as close as possible to the ideal situation of 100% detection rate and 0% false positive rate, thus maximizing $P(I|A)$.

3.3 State of the Art

In Section 2.2.2 we briefly reported the most relevant work on unlabeled anomaly detection. Among the studies described in Section 2.2.2, the closest to ours were presented in [84] and [31], which are described in more detail in the following. Both in [84] and [31] the authors assume that the traffic is extracted from the computer network to be protected and is unlabeled. The only *a priori* knowledge about the data is represented by two assumptions that usually hold in practice: a) the extracted dataset contains two classes of data, normal and anomalous traffic; b) the numerosity of the anomalous traffic class is by far less than the numerosity of the normal traffic class.

In [84], Portnoy et al. used an online clustering algorithm to group similar network connections. Given a metric M and a cluster width w, instances (i.e., network connections) are picked up one by one from the training dataset. According to M a distance is measured between the instance and the centroid of the already existing clusters. The instance is assigned to the closest cluster if the minimum distance is less than w, otherwise the instance initializes a new cluster. After all the instances in the training dataset have been grouped the obtained clusters are labeled. According to the assumption that the numerosity of the anomalous traffic class is by far less than the numerosity of the normal traffic class, the clusters are labeled by numerosity and the largest ones are labeled as “normal” until a certain percentage of instances are covered, and the rest of the clusters are then labeled as “anomalous”. During the detection phase a distance is measured between the instance under test and the centroids of the clusters obtained during training. The instance is classified according to the label associated to the closest cluster. Experiments are performed on the KDDCup’99 dataset. The approach proposed by Portnoy is “monolithic” in the sense that one detection model is constructed for all the possible network protocols.

In [31], Eskin et al. propose to project the patterns from an original feature space F to a suitable feature space F', and then to apply outlier detection algorithms in F' in order to isolate the attack patterns from the normal ones. The proposed detection algorithms are based on the dot product among pattern vectors, therefore kernel functions may be applied, and there is no need to explicitly map the patterns from F to F'. Assuming that the numerosity of the class of normal connections is by far higher than the numerosity of the class of anomalous connections, the authors propose three different algorithms for anomaly detection. The first algorithm is cluster-based. Given a pattern $\mathbf{x}$, the algorithm estimates the local density around $\mathbf{x}$ by counting the number of patterns in a hypersphere of radius w centered in $\mathbf{x}$. Points that are in low density regions are classified as anomalous. The second algorithm is based on a variant of the k-NN algorithm. If the sum of the distances between $\mathbf{x}$ and its k nearest neighbors is greater than a certain threshold, $\mathbf{x}$ is considered anomalous. The third algorithm is the one-class SVM by Schölkopf et al. [92]. Similarly to [84] the anomaly detection algorithms are applied using a “monolithic” approach, i.e., one detection model is constructed which takes into account all the possible network protocols.

3.4 Performance Improvement Using a Modular MCS

As discussed in Section 3.3, the anomaly detection systems proposed in [84] and [31] are based on a "monolithic" approach. A single classifier is constructed in order to distinguish between normal and attack connections, regardless of the network protocol. As each network protocol has different characteristics, it is hard to construct a precise model of normal traffic by using a single classifier. In the following we propose a modular approach. According to the differences among protocols, one or multiple classifiers are constructed in order to model normal connections related to different (groups of) protocols. We then compare the obtained results with the results obtained by using a "monolithic" approach.

3.4.1 Assumptions

As in [84] and [31], we assume that the traffic is directly extracted from the computer network to be protected, and used without need of a labeling process. The only *a priori* knowledge about the data is represented by two assumptions that usually hold in practice: a) the extracted dataset contains two classes of data, normal and anomalous traffic; b) the numerosity of the anomalous traffic class is by far less than the numerosity of the normal traffic class. The latter assumption is usually true unless Distributed Denial of Service attacks (DDoS) or other coordinated attacks are occurring while the traffic is sniffed from the network. However, as DDoS attacks usually have the objective of exhausting the network resources, their effects are in general easy to detect. We then need to be careful and use only the traffic we believe was not sniffed during such attacks. Assumption b) is also supported by the fact that signature-based IDS can be used to prune known attacks from the sniffed traffic in order to reduce the numerosity of the attack class in the training dataset.

3.4.2 Modular Architecture

As mentioned in Section 3.1, each connection is related to a particular service. Different services are characterized by different peculiarities, e.g., the traffic related to the HTTP service is different from the traffic related to the SMTP service. Besides, as different services involve different software applications, attacks launched against different services manifest different characteristics. We

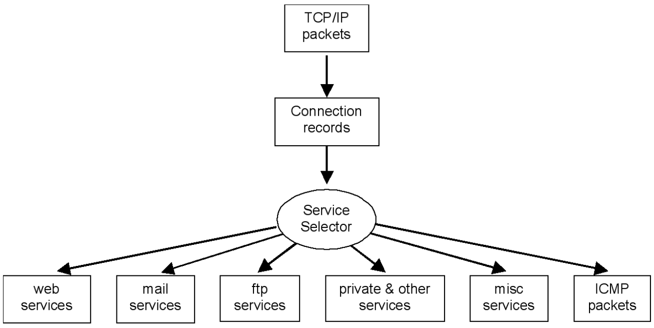

Figure 3.2: Modular architecture

propose to divide the network services into *m* groups, each one containing a number of "similar" services [37]. Therefore, *m* modules are used, each one modeling the normal traffic related to one group of services. The intuitive advantage given by the modular approach is supported by the results in [61], where Lee et al. used information theory to measure the "complexity" of the classification task. The subdivision of services into groups turns into a decrease of the entropy of each subset of data, which in general coincides to the ability to construct a more precise model of the normal traffic. An example of how the services can be grouped is shown in Figure 3.2, where the groupings refer to the network from which the KDD-Cup 1999 dataset was derived (see Section 3.5). In Figure 3.2, a "miscellaneous" group is used to aggregate different services that are rarely used in the computer network at hand. It is worth noting that the number of groups and the type of services in each group depend on the network to be protected, as different networks may provide different services with different characteristics.

3.4.3 Overall vs. Service-Specific False Alarm Rate

Anomaly detection requires setting an acceptance threshold t, so that a traffic pattern $\mathbf{x}$ is labelled as anomalous if its similarity $s(\mathbf{x}, M)$ to the normal model M is less then t. The similarity measure s depends on the particular technique chosen to implement the model of normal traffic M. As we use different modules (i.e., different models) for different services, a method to tune the acceptance threshold for each module is necessary. In order to solve this task, we propose an heuristic approach whereby given a fixed tolerable false alarm rate for the IDS, the overall detection rate is optimized.

Let m be the number of service-specific modules of the IDS; FAR be the overall tolerable false alarm rate; FAR_i be the false alarm rate related to the i-th module; t_i be the acceptance threshold for the i-th module; $P(M_i) = n_i/n$ be the prior distribution of the patterns related to the i-th group of services (i.e., the module) M_i in the training data, where n_i is the number of patterns related to the services for which the module M_i is responsible and n is the total number of patterns in the training dataset. Accordingly, FAR is defined as

$$FAR = \sum_{i=1}^{m} P(M_i) \cdot FAR_i \tag{3.4}$$

Given a fixed value of the tolerable false alarm rate FAR for the IDS, there are many possible ways to "distribute" the overall FAR on the m modules. A value of FAR_i has to be chosen for each module M_i so that Equation (3.4) is satisfied. Once a FAR_i has been set for each module M_i, the thresholds t_i can be chosen accordingly. As a first choice, we could set $FAR_i = FAR$ for each module M_i. This choice satisfies Equation (3.4) and appears to be reasonable, given that no service is seemingly penalized. Nevertheless, this choice presents two drawbacks. One drawback is related to the actual number of false positives generated by each service. As the number of false positives is proportional to $P(M_i)$, the group of services (i.e., the module) accounting for the largest portion of the traffic produces a number of false alarms that is by far larger than the one produced by poorly represented services (i.e., those services which are rarely, or not so often used in the network). This behavior is not adequate as the modules of the IDS that produce an overwhelming number of false alarms could be "turned off" by the network administrator. The other drawback is related to the relation between FAR_i and the detection rate of the i-th service, DR_i. We observed experimentally that for a fixed value of FAR_i, the corresponding value of DR_i strongly dipends on $P(M_i)$. In particular, the larger $P(M_i)$ the larger DR_i. This effect can be explained as follows. Small values of $P(M_i)$ are related to services rarely used in the network, whereby a smaller training set for M_i can be extracted and the corresponding classifier(s) in general will not be able to adequately model the normal traffic.

According to the considerations reported above, given a fixed FAR we propose to compute FAR_i as

$$FAR_i = \frac{1}{m \cdot P(M_i)} FAR \tag{3.5}$$

This choice satisfies Equation (3.4) and allows us to attain an higher overall detection rate DR than that attained by choosing a fixed value $FAR_i = FAR$ for each module.

In order to set an acceptance threshold t_i for the module M_i, to obtain the false alarm rate FAR_i computed as in Equation (3.5), we propose the following heuristic. Let us first note that for a given value of t_i, the fraction $p_{r_i}(t_i)$ of patterns rejected by M_i may contain both patterns related to attacks and false alarms. Let us denote with $p_{a_i}(t_i)$ the fraction of rejected attack patterns using the threshold t_i, and with $far_i(t_i)$ the related fraction of false alarms. It is easy to see that the following relation holds:

$$p_{r_i}(t_i) = p_{a_i}(t_i) + far_i(t_i) \tag{3.6}$$

We want to set t_i so that $far_i(t_i)$ is equal to the desired false alarm rate FAR_i (computed by using (3.5)). As for a given value of t_i the only measurable quantity in Equation (3.6) is the rejection rate $p_{r_i}(t_i)$, we need some hypothesis on $p_{a_i}(t_i)$ so that we can estimate $far_i(t_i) = p_{r_i}(t_i) - p_{a_i}(t_i)$, and therefore we can chose t_i in order to obtain $far_i(t_i) = FAR_i$. We propose to assume $p_{a_i}(t_i) = P_{a_i}$, where P_{a_i} is the expected attack probability for the i-th service[2]. In other words, we assume that for a given threshold value, the rejected patterns are made up of all the attacks related to that service contained in the training set, plus a certain number of normal patterns. Thus, having fixed the value of $p_{a_i}(t_i) = P_{a_i}$, we can tune t_i in order to obtain $far_i(t_i) = FAR_i$.

It is easy to see that the computed thresholds t_i (estimated according to the heuristic described above) produce the required overall FAR (see Equation (3.4)) only if the fraction of patterns rejected by each module actually contains all the attacks $n_i \cdot P_{a_i}$, where n_i is the total number of training patterns for the module M_i. If this is not the case and the rejection rate p_{r_i} includes just a portion of the attacks, a larger number of false alarms $far_i(t_i) > FAR_i$ will occur. However, if the training dataset is a good sample of the real network traffic, we expect most of the attacks will "look" different

[2] In practice, if the network is already protected by "standard" security devices (e.g., firewall, signature-based IDS, etc.), we may be able to estimate P_{a_i} from historical data related to attacks to the network service i that occurred in the past.

from normal traffic and will be likely deemed outliers and rejected by the model.

3.4.4 Service-Specific MCS

Lee et al. [60] proposed a framework for constructing the features used to describe the connections (the patterns). The derived set of features can be subdivided into two groups: i) features describing each single connection; ii) features related to statistical measures on "correlated" connections, namely different connections that have in common either the type of service they refer to or the destination host (i.e., the server host). The latter subset of features is usually referred as *traffic features*. On the other hand, the first group of features can be further subdivided into two subsets, namely *intrinsic features* and *content features*. The intrinsic features are extracted from the *headers* of the packets related to the connection, whereas the content features are extracted from the *payload* (i.e., the data portion of the packets). We call F the entire set of features and I, C and T the subsets of *intrinsic*, *content* and *traffic* features respectively, so that $F = I \cup C \cup T$.

As explained in Section 3.4.2, our IDS is subdivided into a number of modules. Each module implements a model of the normal traffic related to a group of services, so that a module can be viewed as a *service-specific* IDS. The problem of modeling the normal traffic for each module of the IDS can be formulated essentially in two different ways: i) a "monolithic" classifier can be trained using all the available features to describe a pattern; ii) subsets of features from the three groups described above can be used separately to train different classifiers whose outputs can be combined. Depending on the dimensionality of the feature space d and the size of the training set, one approach can outperform the other. In particular, a multiple classifier approach can be effective when the use of a "monolithic" classifier suffers from the "curse of dimensionality" problem, i.e. the training set n_i is too small with respect to d [29]. We propose to use, when needed, a MCS that consists of either two or three classifiers, depending on the module M_i we consider. When a two-classifiers MCS is used, the module is implemented by training two classifiers on two different features subsets, namely $I \cup C$ and $I \cup T$. On the other hand, when a three-classifiers MCS is used, the module is implemented by training a classifier on each single subset of features, namely one classifier is trained by using the subset I, one by using C and one by using T (see Figure 3.3).

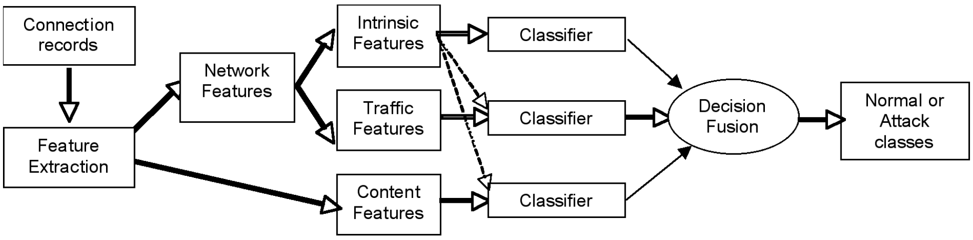

Figure 3.3: Feature subsets for service-specific MCS

3.4.5 One-Class Classification

One-class classification (also referred to as outlier detection) techniques are particularly useful in those two-class problems where one of the classes of objects is well-sampled, whereas the other one is severely undersampled due to the fact that it is too difficult or expensive to obtain a significant number of training patterns. The goal of one-class classification is to distinguish between a set of *target objects* and all the other possible objects, referred as *outliers* [101, 100]. A number of one-class classification techniques have been proposed in the literature. Following the categorization of one-class classifiers proposed by Tax [100], they can be subdivided into three groups, namely density methods, boundary methods and reconstruction methods.

We decided to use one classification method from each category to implement the service-specific MCS modules described in Section 3.4.4 in order to compare different approaches that showed good results in other applications. In particular, we chose the Parzen density estimation [29] from the density methods, the ν-SVC [92] from the boundary methods and the the k-means algorithm [43] from the reconstruction methods. These one-class classifiers exhibited good performance on a number of applications [100]. Besides, the output of the k-means and ν-SVC classifiers can be redefined as class-conditional probability density functions, so that they can be correctly combined with the output of the Parzen classifier (see Section 3.4.6). We also trained the clustering technique proposed by Eskin et al. [31] in order to compare the results of the combination of “standard” pattern recognition techniques with an algorithm tailored to the unlabeled intrusion detection problem.

Parzen Density Estimation

The Parzen-window approach [29] can be used to estimate the density of the target objects distribution

$$p(\mathbf{x}|\omega_t) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{h^d}\,\varphi\left(\frac{\mathbf{x}-\mathbf{x}_i}{h}\right) \tag{3.7}$$

where n is the total number of training patterns belonging to the target class ω_t, $\mathbf{x}_i$ is the i-th training pattern, φ is a kernel function, h is the width of the Parzen-window and $p(\mathbf{x}|\omega_t)$ is the estimated class-conditional probability density distribution. When the Gaussian kernel

$$\varphi(x) = \frac{1}{(2\pi)^{\frac{d}{2}}}\exp\left(-\frac{1}{2}||\mathbf{x}||^2\right) \tag{3.8}$$

is used, $p(\mathbf{x}|\omega_t)$ can be written as

$$p(\mathbf{x}|\omega_t) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{(2\pi s)^{d/2}}\exp\left(-\frac{||x-x_i||^2}{2s}\right), \quad s = h^2. \tag{3.9}$$

and the one-class Parzen classifier can be obtained by simply setting a threshold θ whereby a pattern $\mathbf{z}$ is rejected (i.e., deemed an outlier) if $p(\mathbf{z}|\omega_t) < \theta$ [100].

k-means

The k-means classifier is based on the well-known k-means clustering algorithm [43]. The algorithm identifies k clusters in the data by iteratively assigning each pattern to the nearest cluster. This algorithm can be used as a one-class classifier by clustering the training set and then computing the distance $d(\mathbf{z}, \omega_t)$ of a test pattern $\mathbf{z}$ from the target distribution ω_t as

$$d(\mathbf{z}, \omega_t) = \min_{i=1..k} ||\mathbf{z} - \mu_i|| \tag{3.10}$$

where μ_i represents the i-th cluster center. If the distance is larger than a threshold θ the pattern will be rejected [100]. It is hard to map the distance $d(\mathbf{z}, \omega_t)$ into a probability density distribution and thus the combination of the k-means one-class classifier with density-based classifiers (e.g., the

Parzen classifier) may produce unreliable results, as will be explained in Section 3.4.6. In order to allow this algorithm to produce an output that can be interpreted as a probability density function, we propose to use all the k distances between the test pattern $\mathbf{z}$ and the centroids μ_i as follows

$$\begin{gathered} p(\mathbf{x}|\omega_t) = \frac{1}{k}\sum_{i=1}^{k}\frac{1}{(2\pi s)^{d/2}}\exp\left(-\frac{||\mathbf{x}-\mu_i||^2}{2s}\right) \\ s = \operatorname*{avg}_{i,j}\ ||\mu_i-\mu_j||, \quad i,j=1,2,..,k \end{gathered} \tag{3.11}$$

In other words, we model the distribution of the target class by a mixture of k normal densities, each one centred on a centroid μ_i. An heuristic is used to compute s as the average distance between the k centroids. As for the one-class Parzen classifier, the one-class k-means classifier based on Equation (3.11) can be obtained by setting a threshold θ whereby a pattern $\mathbf{z}$ is rejected if $p(\mathbf{z}|\omega_t) < \theta$. It is worth noting that the same number of distances $||\mathbf{z}-\mu_i||$ have to be computed both in (3.10) and (3.11). Besides, the number of centroids is in general chosen to be low, therefore s can be efficiently computed. This means that the proposed probability density estimate does not add appreciable complexity to the classifier.

ν-SVC

The ν-SVC classifier was proposed by Schölkopf et al. in [92] and is inspired by the Support Vector Machine classifier proposed by Vapnik [107]. The one-class classification problem is formulated to find an hyperplane that separates a desired fraction of the training patterns from the origin of the feature space $\mathbb{F}$. This hyperplane cannot always be found in the original feature space, thus a mapping function $\Phi : \mathbb{F} \to \mathbb{F}'$, from $\mathbb{F}$ to a kernel space $\mathbb{F}'$, is used. In particular, it can be proven that when the Gaussian kernel

$$K(\mathbf{x},\mathbf{y}) = \Phi(\mathbf{x})\cdot\Phi(\mathbf{y}) = exp\left(-\frac{||\mathbf{x}-\mathbf{y}||^2}{2s}\right) \tag{3.12}$$

is used, it is always possible to find a hyperplane that solves the separation problem. The problem is formulated as follows:

$$\min_{\mathbf{w},\xi,\rho}\left(\frac{1}{2}\|\mathbf{w}\|^2-\rho+\frac{1}{\nu l}\sum_i\xi_i\right)$$
$$\mathbf{w}\cdot\phi\left(\mathbf{x}_i\right)\geq\rho-\xi_i,\quad \xi_i\geq 0,\quad \forall i=1,..,l \tag{3.13}$$

where **w** is a vector orthogonal to the hyperplane, ν represents the fraction of training patterns that are allowed to be rejected (i.e., that are not separated from the origin by the hyperplane), $\mathbf{x}_i$ is the i-th training pattern, l is the total number of training patterns, $\xi=[\xi_1,..,\xi_l]^T$ is a vector of slack variables used to "penalize" the rejected patterns, ρ represents the margin, i.e., the distance of the hyperplane from the origin.

The solution of (3.13) brings to the decision function, for a generic test pattern **z**, formulated as

$$f_{svc}(\mathbf{z})=I\left(\sum_i\alpha_iK\left(\mathbf{x}_i,\mathbf{z}\right)\geq\rho\right),\quad \sum_{i=1}^{l}\alpha_i=1 \tag{3.14}$$

where I is the indicator function[3] and the parameters α_i and ρ are provided by the solution of (3.13). According to (3.14), a pattern **z** is either rejected if $f_{svc}(\mathbf{z})=0$ or accepted as target object if $f_{svc}(\mathbf{z})=1$. When the Gaussian kernel (3.12) is used, the output of the ν-SVC can be formulated in terms of a class conditional probability by

$$p(\mathbf{x}|\omega_t)=\frac{1}{(2\pi\cdot s)^{\frac{d}{2}}}\sum_{i=1}^{n}\alpha_i\cdot K(\mathbf{x},\mathbf{x}_i)=\sum_{i=1}^{n}\alpha_i\frac{1}{(2\pi\cdot s)^{\frac{d}{2}}}\cdot e^{-\frac{1}{2}\frac{\|\mathbf{x}-\mathbf{x}_i\|^2}{s}} \tag{3.15}$$

which respects the constraint $\int_{\mathbb{R}^d}p(\mathbf{x}|\omega_t)d\mathbf{x}=1$.

It is worth noting that in general only a small number of coefficients α_i will be different from zero, thus $p(\mathbf{x}|\omega_t)$ can be efficiently computed. The training patterns $\mathbf{x}_i$ whereby the related $\alpha_i\neq 0$ represent the support vectors for the ν-SVC. The acceptance threshold can be rewritten as

$$\rho'=\frac{\rho}{(2\pi\cdot s)^{\frac{d}{2}}} \tag{3.16}$$

so that a pattern **z** will be considered an outlier if $p(\mathbf{z}|\omega_t)<\rho'$.

It is worth noting that Tax et. al [102] independently formulated a SVM-based one-class classi-

[3] $I(x)=1$ if x is true, otherwise $I(x)=0$

fier whose solution is identical to the one of the ν-SVC when the Gaussian kernel is used.

3.4.6 Combining One-Class Classifiers

Traditional pattern classifiers can be combined by using many different combination rules and methods [57]. Among the combination rules, the *min*, *max*, *mean* and *product* rules [50] are some of the most commonly used. These combination rules can be easily applied when the output of the classifiers can be viewed as an *a posteriori* probability $P_i(\omega_j|\mathbf{x})$, where p_i refers to the output of the i-classifier, whereas ω_j is the j-class of objects. In case of a two-class problem, the *a posteriori* probability can be written as

$$P_i(\omega_j|\mathbf{x}) = \frac{p_i(\mathbf{x}|\omega_j)P(\omega_j)}{p_i(\mathbf{x})} = \frac{p_i(\mathbf{x}|\omega_j)P(\omega_j)}{p_i(\mathbf{x}|\omega_1)P(\omega_1) + p_i(\mathbf{x}|\omega_2)P(\omega_2)}, \quad j = 1,2, \quad i = 1,..,L \qquad (3.17)$$

where L is the number of classifiers. Unfortunately, in case of one-class classifiers in general it is not possible to reliably estimate the probability distribution of one of the two classes, namely the probability density of the outlier objects (i.e., one of the terms in the denominator in (3.17)). Tax et al. [101] proposed to consider the distribution of the outlier to be constant in a suitable region of the feature set, so that the *a posteriori* probability for the target class, for example, can be approximated as

$$P_i(\omega_t|\mathbf{x}) = \frac{p_i(\mathbf{x}|\omega_t)P(\omega_t)}{p_i(\mathbf{x}|\omega_t)P(\omega_t) + \theta_i \cdot P(\omega_o)}, \quad i = 1,..,L \qquad (3.18)$$

where ω_t represents the target class, ω_o represent the outlier class and θ_i is the uniform density distribution assumed for the outlier patterns. Let's consider now the traditional *mean* combination rule. We need to compute

$$\begin{aligned} \mu(\omega_t|\mathbf{x}) &= \tfrac{1}{L}\textstyle\sum_{i=1}^{L} P_i(\omega_t|\mathbf{x}) \\ \mu(\omega_o|\mathbf{x}) &= \tfrac{1}{L}\textstyle\sum_{i=1}^{L} P_i(\omega_o|\mathbf{x}) \end{aligned} \qquad (3.19)$$

and the decision criterion is

$$\mathbf{x} \text{ is an outlier} \quad \Leftrightarrow \; \mu(\omega_t|\mathbf{x}) < \mu(\omega_o|\mathbf{x}) \tag{3.20}$$

If we assume $p_i(\mathbf{x}) \simeq p(\mathbf{x}), \forall i$, we can write

$$\mu(\omega_j|\mathbf{x}) = \tfrac{1}{L} \textstyle\sum_{i=1}^{L} \frac{p_i(\mathbf{x}|\omega_j)\cdot P(\omega_j)}{p(\mathbf{x})} = \tfrac{1}{L} \frac{P(\omega_j)}{p(\mathbf{x})} \textstyle\sum_{i=1}^{L} p_i(\mathbf{x}|\omega_j) \tag{3.21}$$

where $j = t, o$ (i.e., (3.21) is applied to both the target and the outlier class). In this case we can compute

$$y_{avg}(\mathbf{x}) = \frac{1}{L} \sum_{i=1}^{L} p_i(\mathbf{x}|\omega_t) \tag{3.22}$$

$$\theta^{'} = \frac{P(\omega_o)}{P(\omega_t)} \cdot \frac{1}{L} \sum_{i=1}^{L} \theta_i \tag{3.23}$$

and the decision criterion (3.20) becomes simply

$$\mathbf{x} \text{ is an outlier} \quad \Leftrightarrow \; y_{avg}(\mathbf{x}) < \theta^{'} \tag{3.24}$$

which means that we can combine the class-conditional probability density functions, instead of the *a posteriori* probabilities estimated by each classifier. The obtained $y_{avg}(\mathbf{x})$ can be used as a standard one-class classifier output and the threshold $\theta^{'}$ can be independently tuned to attain the desired trade-off between false positives (i.e., target objects classified as outliers) and false negatives (i.e., outliers classified as belonging to the target class). This approach is (almost) exactly like the one proposed in [101] and [100] and can be extended to the *min*, *max* and *product* rules.

Another approach is to estimate $P_i(\omega_t|\mathbf{x})$ and $P_i(\omega_o|\mathbf{x})$ so that the decision criterion (3.20) can be used directly. For each one-class classifier i we have

$$\begin{aligned} P_i(\omega_t|\mathbf{x}) &= \tfrac{p_i(\mathbf{x}|\omega_t)P(\omega_t)}{p_i(\mathbf{x}|\omega_t)P(\omega_t)+\theta_i\cdot P(\omega_o)} \\ P_i(\omega_o|\mathbf{x}) &= \tfrac{\theta_i\cdot P(\omega_o)}{p_i(\mathbf{x}|\omega_t)P(\omega_t)+\theta_i\cdot P(\omega_o)} \end{aligned} \tag{3.25}$$

and, setting $\tau_i = \theta_i \cdot \frac{P(\omega_o)}{P(\omega_t)}$, the decision criterion for the classifier i can be written as

$$\mathbf{x} \text{ is an outlier } \Leftrightarrow p_i(\mathbf{x}|\omega_t) < \tau_i \tag{3.26}$$

It is worth noting that τ_i represents the decision threshold applied on the output of classifier i. According to (3.25) we can write

$$P_i(\omega_t|\mathbf{x}) = \frac{p_i(\mathbf{x}|\omega_t)}{p_i(\mathbf{x}|\omega_t) + \tau_i}, \quad i = 1, .., L \tag{3.27}$$

$$P_i(\omega_o|\mathbf{x}) = \frac{\tau_i}{p_i(\mathbf{x}|\omega_t) + \tau_i}, \quad i = 1, .., L \tag{3.28}$$

In practice, we can set the thresholds τ_i so that a given rejection rate is produced by each single one-class classifier. Once the thresholds τ_i, $i = 1, .., L$, have been set, the posterior probabilities can be estimated using (3.27) and (3.28), and the rule (3.20) can be applied. This approach can be extended to the *min*, *max* and *product* rules by computing $\mu(\omega_t|\mathbf{x})$ and $\mu(\omega_o|\mathbf{x})$ according to the new rule and then applying (3.20).

As mentioned in Section 3.4.5, it is not possible to directly make use of the output of one-class classifiers that implement boundary or reconstruction methods in (3.22), (3.27) and (3.28). In order to solve this problem, Tax et al. [101] proposed an heuristic approach to map the output of "distance-based" classifiers to a probability estimate

$$\tilde{P}(\mathbf{x}|\omega_t) = exp\left(-\frac{\rho(\mathbf{x}|\omega_t)}{s}\right) \tag{3.29}$$

where $\rho(\mathbf{x}|\omega_t)$ is the output to be mapped (e.g., $\rho(\mathbf{x}|\omega_t) = \min_{i=1..k} ||\mathbf{x} - \mu_i||$, if the k-means classifier is considered). However, in general $\tilde{P}$ does not repsect the integral constraint for a density probability distribution, whereby

$$\int_{\mathbb{R}^d} \tilde{P}(\mathbf{x}|\omega_t) d\mathbf{x} \neq 1 \tag{3.30}$$

This fact may produce some problems, especially when the output of "distance-based" one-class classifiers is combined with density-based classifiers (e.g., the Parzen classifier described in Section 3.4.5), which respect the integral constraint by definition. On the other hand, the methods proposed in Section 3.4.5 to compute the output of the k-means and ν-SVC classifiers do not suffer from this problem and the decision criterion (3.24) can be used without further output transformations.

3.5 Experimental Results

Experiments were carried out on a subset of the DARPA 1998 dataset distributed as part of the UCI KDD Archive (`http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`). The DARPA 1998 dataset was created by the MIT Lincoln Laboratory group in the framework of the 1998 Intrusion Detection Evaluation Program (`http://www.ll.mit.edu/IST/ideval`). This dataset was obtained from the network traffic produced by simulating the computer network of an air-force base. It consists of seven weeks of traffic data for training purposes, and two weeks of data for testing. A subset of the traffic of the DARPA 1998 dataset has been mapped to a pattern recognition problem and distributed as the KDD-Cup 1999 dataset. The training set is made of 494,020 patterns, and the test set contains 311,029 patterns. Each pattern represents a network connection described by a 41-dimensional feature vector according to the set of features illustrated in Section 3.4.2. In particular, 9 features were of the intrinsic type, 13 features were of the content type, and the remaining 19 features were of the traffic type. Each pattern of the data set is labelled as belonging to one out of five classes, namely *normal traffic* and four different classes of attack: *Probe*, *Denial of Service* (DoS), *Remote to Local* (R2L), and *User to Root* (U2R). The attacks belonging to a certain attack class are designed to attain the same effect by exploiting different vulnerabilities of the computer network.

The DARPA dataset has been widely criticized [67, 70]. The main criticism is related to the fact that the traffic traces reported in the dataset are not representative of a real network scenario. In particular, it is worth noting that the prior probabilities of the attack classes included in the DARPA 1998 dataset (and thus in the KDD-Cup 1999) cannot be considered representative of the traffic in a real network. This fact has been clearly pointed out in a critique of the DARPA corpus of data by McHugh [70]. Although this dataset has been criticized, it is currently used by researchers because it is the only reference dataset that allows the designers to compare results obtained using different intrusion detection techniques.

In order to perform experiments with *unlabeled* intrusion detection techniques, we removed the labels of all the training patterns to simulate the *unlabeled* collection of network traffic. Besides, we relabeled the patterns of the test set as belonging either to the *normal traffic* class or to the “generic” *attack* class, thus discarding the different labels assigned to attack patterns related to different classes of attack. Nominal features have been converted into numerical values according to the procedure in [31]. Given a nominal feature with N possible distinct values, it is mapped to a vector of length N, i.e., the vector contains one coordinate for every possible value of the feature. When a particular value of the feature is mapped on the vector, the coordinate corresponding to the value of the feature is set to $1/N$, whereas the other coordinates (i.e., the ones corresponding to the other $N-1$ possible values for the feature) are set to zero.

According to the description of the modular architecture presented in Section 3.4.2, we divided the traffic of the data set into six subsets, each one related to “similar” services: *HTTP*, containing the traffic related to the HTTP protocol; *FTP*, containing the traffic related to the control flow and data flow for the FTP protocol, and the traffic related to the TFTP protocol; *Mail*, containing the traffic related to the SMTP, POP2, POP3, NNTP, and IMAP4 protocols; *ICMP*, containing the traffic related to the ICMP protocol; *Private&Other*, containing the traffic related to TCP/UDP ports higher than 49,152; *Miscellaneous*, containing all the remaining traffic. For each module, the features taking a constant value for all patterns have been discarded, provided that these features have a constant value by “definition” for that service, and not by chance. For example, the intrinsic feature “protocol_type” is always constant and equal to the value “TCP” for the *http*, and *mail* services, thus for those services it can be discarded. As a result, for each module we used a subset of the 41 available features, namely: 29 features for the *HTTP* module; 34 features for the *FTP* module; 16 features for the *ICMP* module (in particular, the content features were discarded as they have no meaning for the ICMP traffic); 31 features for the *Mail* module; 37 features for the *Miscellaneous* module; 29 features for the *Private&Other* module.

3.5.1 Training Set Undersampling

As mentioned above, the prior probabilities of the attack classes in the training portion of the KDD-Cup 1999 dataset cannot be considered representative of the traffic in a real network. The analysis

	HTTP	**FTP**	**Mail**	**ICMP**	**Private & Others**	**Miscellaneous**
Normal	61,885 (96.55%)	4,172 (78.16%)	9,677 (99.83%)	1,288 (0.46%)	12,998 (81.27%)	7,257 (98.33%)
Attacks	2,208 (3.45%)	1,166 (21.84%)	16 (0.17%)	28,1250 (99.54%)	2,996 (18.73%)	123 (1.67%)

Table 3.1: Composition of the training set before the undersampling phase

	HTTP	**FTP**	**Mail**	**ICMP**	**Private & Others**	**Miscellaneous**
Normal	61,885 (99.83%)	4,172 (96.60%)	9,677 (99.83%)	1,288 (69.66%)	12,998 (96.09%)	7,257 (98.33%)
Attacks	106 (0.17%)	147 (3.40%)	16 (0.17%)	561 (30.34%)	529 (3.91%)	123 (1.67%)

Table 3.2: Composition of the training set after the undersampling phase

of the training set confirmed that it contained a large fraction of attacks compared to normal traffic patterns, as show in Table 3.1. This violates the first assumption behind unlabeled techniques, i.e., connections containing attacks should account for a small portion of the network traffic. As typical network traffic satisfies this assumption [84], we filtered the training set so that the selected data satisfied this assumption. To this end, for each service we retained all the normal connections, while we sampled the attack patterns so that they accounted for 1.5% of the total traffic. This sampling procedure is similar to the one performed by other researchers [84, 31]. Let us recall that each *attack class* is made up of connections related to a number of different *attack types*, each *attack type* designed to produce the same effect of all the attacks in the same class. For each type of attack, a different number of patterns is available because each attack type produces a different number of connections, and because of the simulations carried out during the DARPA programme. A number of techniques can be used to sample a set of data such that the resulting subset is representative of the whole data [15].

In the reported experiments we reduced the percentage of attacks by reducing the number of those attacks accounting for a number of connections larger than 973, which is 1% of the total normal connections. In particular we proceeded as follows:

a) 10 subsets, each one containing 101 patterns, are extracted randomly from each *attack type* (this “magic” number was chosen in order to attain a total percentage of attacks equal to 1.5%);

b) for each subset, we trained a ν-SVC classifier, and computed the error attained by using the

	HTTP	**FTP**	**Mail**	**ICMP**	**Private & Others**	**Miscellaneous**
Normal	39,247 (95.04%)	1,170 (38.22%)	3,222 (25.50%)	380 (0.23%)	12,930 (16.35%)	3,644 (43.53%)
Attacks	2,050 (4.96%)	1,891 (61.78%)	9,412 (74.50%)	164,591 (99.77%)	66,145 (83.65%)	4,727 (56.47%)

Table 3.3: Composition of the test set

		HTTP	**FTP**	**Mail**	**ICMP**	**Private & Others**	**Miscellaneous**
ν-SVC	F	**0.995**	0.894	**0.971**	0.862	**0.992**	0.987
ν-SVC - max rule	$I \cup C + I \cup T$	0.956	0.918	0.960	-	0.911	0.975
	$I + C + T$	0.807	0.566	0.956	**0.929**	0.918	0.939
ν-SVC - min rule	$I \cup C + I \cup T$	0.948	0.967	0.855	-	0.921	0.953
	$I + C + T$	0.773	**0.973**	0.954	0.913	0.904	0.944
ν-SVC - mean rule	$I \cup C + I \cup T$	0.952	0.962	0.970	-	0.957	0.965
	$I + C + T$	0.865	0.972	0.953	0.879	0.921	**0.988**
ν-SVC - product rule	$I \cup C + I \cup T$	0.951	0.961	0.857	-	0.919	0.963
	$I + C + T$	0.865	0.971	0.953	0.879	0.921	0.945

Table 3.4: Performance attained by the ν-SVC classifier on the six modules in terms of AUC. For each module, the best performance is reported in bold.

remaining patterns of that attack type as a test set;

c) the subset with the smallest error is selected, as it can be considered representative of the entire set of available connections for that attack type.

Table 3.2 shows the composition of the training set obtained after the preprocessing phase. It can be observed that attacks are not distributed uniformly among different services. While the overall percentage of attacks has been reduced so that it is equal to 1.5% of all the training traffic, the percentages of attacks related to different services range from the 0.17% of the *HTTP* and *Mail* traffic, to the 30.34% of the *ICMP* traffic. The high percentage of attacks in the *ICMP* traffic can be explained by observing that the available training set contained a very small number of normal *ICMP* connections compared to attacks, so that the proposed reduction of the number of attack patterns left the *ICMP* traffic data unbalanced.

It is worth noting that the distribution of traffic reported in Table 3.2 was used to compute the prior probabilities related to the different modules of the IDS, according to the discussion in Section 3.4.3.

Table 3.3 shows the composition of the test set. As shown in the table, the test set contains a very large fraction of attacks, as it was designed to test the performance of IDS and not to be representative of a realistic network traffic. It is worth noting that we did not apply any changes on

		HTTP	FTP	Mail	ICMP	Private & Others	Miscellaneous
k-means	F	**0.978**	0.820	0.899	0.736	0.918	0.955
k-means - max rule	$I \cup C + I \cup T$	0.864	0.874	0.926	-	0.917	**0.974**
	$I + C + T$	0.872	0.335	**0.930**	**0.913**	0.917	0.889
k-means - min rule	$I \cup C + I \cup T$	0.353	0.830	0.826	-	0.903	0.909
	$I + C + T$	0.814	**0.926**	0.630	0.750	0.907	0.284
k-means - mean rule	$I \cup C + I \cup T$	0.859	0.778	0.913	-	**0.965**	0.932
	$I + C + T$	0.961	0.850	0.929	0.740	0.920	0.947
k-means - product rule	$I \cup C + I \cup T$	0.858	0.777	0.913	-	**0.965**	0.932
	$I + C + T$	0.965	0.851	0.929	0.740	0.920	0.951

Table 3.5: Performance attained by the *k*-means classifier on the six modules in terms of AUC. For each module, the best performance is reported in bold.

		HTTP	FTP	Mail	ICMP	Private & Others	Miscellaneous
Parzen	F	0.977	0.878	0.932	0.743	0.921	**0.982**
Parzen - max rule	$I \cup C + I \cup T$	0.854	0.904	0.568	-	0.905	0.900
	$I + C + T$	0.858	0.368	0.581	**0.872**	0.903	0.909
Parzen - min rule	$I \cup C + I \cup T$	**0.987**	0.868	0.940	-	0.921	0.974
	$I + C + T$	0.982	0.914	0.940	0.704	0.864	0.698
Parzen - mean rule	$I \cup C + I \cup T$	0.854	0.904	0.828	-	**0.991**	0.900
	$I + C + T$	0.858	0.867	0.582	**0.872**	0.910	0.909
Parzen - product rule	$I \cup C + I \cup T$	0.857	0.913	0.839	-	0.977	0.906
	$I + C + T$	0.959	**0.924**	**0.941**	0.725	0.888	0.898

Table 3.6: Performance attained by the Parzen classifier on the six modules in terms of AUC. For each module, the best performance is reported in bold.

the test set.

3.5.2 Performace Evaluation

We divided the performance evaluation experiments into two phases. In the first phase, we evaluated the performance of one module of the IDS at a time. In particular, for each module the performance of a “monolithic” classifier is compared to the performance attained by combining classifiers trained on distinct feature subsets (see Section 3.4.4). In the second phase, the modules related to different services are combined, and the performance of the overall IDS is evaluated. Performance evaluation has been carried out by ROC curve analysis, i.e., by computing the detection rate as a function of the false alarm rate. Different ROC can be compared by computing the Area Under the Curve (AUC). AUC measures the average performance of the related classifier, so that the larger the value of AUC of a classifier the higher the performance [100]. It is worth noting that AUC usually measures the average performance of classifiers considering the entire range of variation of the false positive rate. For some ranges of the false alarm rate the classifier with the smallest AUC value may provide the highest detection rate. Therefore, it may be better to measure the AUC in the interval $[0, a]$, where

		HTTP	FTP	Mail	ICMP	Private & Others	Miscellaneous
Cluster	F	0.967	**0.839**	0.891	0.739	0.847	**0.973**
Cluster - max rule	$I \cup C + I \cup T$	0.965	0.705	0.949	-	0.843	0.253
	$I + C + T$	0.740	0.478	0.949	**0.918**	0.390	0.141
Cluster - min rule	$I \cup C + I \cup T$	0.922	0.782	0.802	-	0.903	0.875
	$I + C + T$	0.970	0.809	0.814	0.856	0.848	0.936
Cluster - mean rule	$I \cup C + I \cup T$	0.932	0.829	0.962	-	**0.915**	0.876
	$I + C + T$	**0.983**	0.874	**0.970**	0.872	0.847	0.958
Cluster - product rule	$I \cup C + I \cup T$	0.924	0.802	0.802	-	0.903	0.875
	$I + C + T$	0.980	0.809	0.814	0.872	0.947	0.943

Table 3.7: Performance attained by the Clustering algorithm proposed in [31] on the six modules in terms of AUC. For each module, the best performance is reported in bold.

	HTTP	FTP	Mail	ICMP	Private & Others	Miscellaneous
Best	ν-SVC F	ν-SVC min rule $I + C + T$	ν-SVC F	ν-SVC max rule $I + T$	Parzen min rule $I \cup C + I \cup T$	ν-SVC mean rule $I + C + T$

Table 3.8: Summary of the best results in terms of AUC attained for each module.

$a < 1$ represents the maximum expected false positive rate. However, it is not always possible to know in advance the working point (or the set of possible working points) on the ROC curve that will be actually used during the operational phase. Moreover, in our application the overall false positive rate is “distributed” in different percentages on different modules in order to optimize the performance of the IDS (see Section 3.4.2). In these cases of unknown a the AUC measured in the interval $[0, 1]$ is a valuable indicator of the performance of the classifier.

Evaluation of Service-Specific Modules

The first phase of the performance evaluation consisted of three experiments for each of the six modules. The first experiment was designed to assess the performance of individual one-class classifiers, i.e., the ν-SVC, the k-means, and the Parzen classifier when the patterns are described by using the entire set of available features F. The performance of the clustering algorithm described in [31] have been also computed for comparison purposes. The second experiment was designed to assess the performance attained by combining classifiers trained on two distinct feature subsets, i.e. the subset of intrinsic and traffic features $I \cup T$, and the subset made of intrinsic and content features $I \cup C$ (see Section 3.4.4). In particular, each classifier has been trained using the two feature subsets, and then they have been combined by using four different combination rules, i.e. the *max* rule, the *min* rule, the *mean* rule, and the *product* rule. The third experiment was designed to assess the performance

Best Modules in terms of AUC		**ν-SVC**		**Best ν-SVC modules in terms of AUC**	
False Alarm Rate	**Detection Rate**	**False Alarm Rate**	**Detection rate**	**False Alarm Rate**	**Detection Rate**
0.87%	75.34%	0.91%	67.31%	0.88%	79.27%
2.10%	80.35%	2.06%	75.61%	2.07%	89.45%
2.64%	80.80%	2.65%	77.10%	2.66%	89.67%
4.00%	85.67%	3.20%	86.31%	3.28%	89.92%
5.49%	94.12%	4.51%	92.25%	4.82%	93.02%
6.86%	94.27%	6.72%	93.91%	6.49%	94.16%
8.25%	94.32%	8.09%	94.12%	8.05%	94.26%
10.44%	94.38%	9.62%	94.25%	9.49%	94.31%

Table 3.9: Results attained by the proposed three modular systems.

attained by combining classifiers trained on three distinct feature subsets, i.e. the intrinsic features I, the traffic features T, and the content features C (see Section 3.4.4) by using again four different combination rules.

When combining classifiers trained on different feature spaces, we used both the combination approaches described in Section 3.4.6. We noted that for the ν-SVC, the k-means, and the clustering algorithm proposed in [31], the best performance was obtained by estimating the posterior probabilities for the target class as in (3.27) and then comparing these probabilities to a varying threshold in order to compute the ROC curves. For the Parzen classifier, the combination of class conditional probabilities, using (3.22) and the decision criteria (3.24), produced the best results.

In the following, we discuss the results obtained by applying the best combination approach for each single classifier. Therefore, we present the results obtained by combining the posterior probabilities for the ν-SVC, the k-means, and the clustering algorithm, and the results obtained by combining the class conditional probabilities for the Parzen classifier.

Tables 3.4, 3.5, 3.6, and 3.7 summarize the performance results on the test set in terms of AUC, for the ν-SVC, the k-means, the Parzen classifier, and the clustering algorithm proposed in [31], respectively. For each algorithm, the parameters have been tuned on the training set. It is worth noting that in the case of the *ICMP* protocol only intrinsic and traffic features were available, thus only the third kind of experiment could be performed by combining two one-class classifiers trained on intrinsic and traffic features, respectively.

The obtained results are discussed in Section 3.6.

Evaluation of Overall IDS

In order to analyze the performance of the overall IDS, we built three systems:

1) An "optimal" system made up, for each module, of the classification techniques that provided the highest value of AUC, according to Table 3.8.

2) A system made up of one "monolithic" ν-SVC for each module. We chose to use ν-SVC classifiers because on average they provide better results than the other considered classifiers, as discussed in Section 3.6.

3) As in the second system, we chose to use ν-SVC classifiers. Then, for each module we chose between a "monolithic" versus a MCS approach, according to best performance results reported in Table 3.4. It is worth noting that for the *Miscellaneous* module the performance of the "monolithic" classifier is really close to the best performance result. Therefore, it is difficult to conclued which approach really performs better than the other. We chose to construct a system made up of one "monolithic" ν-SVC for the *HTTP*, *Mail*, *Miscellaneous* and *Private&Other* modules, and a MCS for the *FTP* and *ICMP* modules (we will further discuss the motivation for this choice in Section 3.6). For the *FTP* module we used an MCS constructed by using three ν-SVC classifiers, namely one trained on the subset of features I, one on the subset C and one on the subset T. For the *ICMP* module we constructed a MCS using two ν-SVC classifiers, namely one trained on the subset of features I and one on the subset T. In particular, for the *FTP* module, the *min* rule was used, whereas the *max* rule was used for the *ICMP* module.

In order to evaluate the performance of the three IDS systems, we computed some working points according to the heuristic proposed in Section 3.4.3. The attained results are reported in Table 3.9. The motivation for the choice of the three proposed IDS systems and the attained results are further discussed in Section 3.6.

3.6 Discussion

The results reported in Section 3.5.2 clearly show that the ν-SVC algorithm provides the highest AUC value for all services, when classifiers are trained using all the available features. The difference between the performance of ν-SVC and that of the other algorithms is very small in the case of the *HTTP* and the *Miscellaneous* traffic, while it is larger for the other services.

Tables 3.4, 3.5, 3.6, and 3.7 show that combining classifiers trained on distinct feature sets does not always improve performance, with respect to those attained by classifiers trained on the entire feature set. In particular, it can be seen that for the ν-SVC, k-means, and Parzen classifiers, the use of distinct feature sets clearly outperforms the use of the entire feature set F only for the *FTP*, and *ICMP* modules. In the case of the clustering algorithm, the use of distinct feature sets clearly outperforms the use of the entire feature set F only for the *Mail*, *ICMP*, and *Private&Others* modules. In all other cases the differences in performance are small, thus the superiority of one technique against the others cannot be concluded. Unfortunately, results show no regularity. For this reason, it is difficult to explain the behavior of different classifiers and combination rules on different modules. On the other hand, results clearly show that each module should be carefully and independently designed by making a decision about the classification algorithm to be used, and by choosing between an individual classification technique and the MCS approach.

Summing up, reported results allow us to conclude that the ν-SVC algorithm performs better than the other ones, on average. Further, it is easy to see that the combination of distinct feature representations usually provides significantly higher performance, with respect to just one classifier trained on the entire feature set, only for the *FTP* and *ICMP* modules. These observations have been used in Section 3.5.2, where three different overall IDS made up of six modules are described.

In order to compare the performance of the modular systems proposed in Section 3.5.2 to the approach used by Eskin et al. [31], we trained the clustering algorithm proposed in [31] and the ν-SVC on the entire training set obtained after subsampling. It is worth noting that this approach is the same used in [31]. Besides, our test set is the same as the one used in [31], and we also used an approach similar to the one proposed in [31] to adjust the training dataset. The performance results obtained on the test set are reported in Tables 3.10 and 3.11, respectively. It is easy to see that if

Clustering	
False Alarm Rate	**Detection Rate**
1%	18.37%
2%	26.80%
3%	27.21%
4%	92.21%
5%	92.24%
6%	92.25%
7%	92.25%
8%	92.29%
9%	92.29%
10%	92.68%

Table 3.10: Results attained by applying the “monolithic” approach using the clustering algorithm proposed in [31].

ν-SVC	
False Alarm Rate	**Detection Rate**
1%	17.91%
2%	66.44%
3%	78.40%
4%	78.85%
5%	86.07%
6%	92.53%
7%	92.57%
8%	92.60%
9%	92.63%
10%	92.91%

Table 3.11: Results attained by applying the “monolithic” approach using the ν-SVC classifier.

the false alarm rate is set to 1%, the algorithms trained on the entire training set provide a detection rate near 18%, while the proposed modular approaches provide detection rates from 67% to 79% (see Table 3.9). As the effectiveness of IDS depends on the capability of providing high detection rates at small false alarms rates, the proposed modular approaches are very effective compared to the “monolithic” approaches. At 4% false alarm rate, the “monolithic” clustering algorithm provides better results than the modular approaches, in terms of detection rates. However, for higher false positive rates, the clustering algorithm does not provide performance improvements, whereas the proposed modular IDS reaches definitely better detection rates with respect to the ones obtained at low false positive rates. It is worth noting that, from a practical point of view, the working point of anomaly detectors are usually tuned to produce a low false alarm rate (e.g., equal to 1% or lower).

Reported results clearly show that the proposed modular approach outperforms the “monolithic” approaches in the range of low false positive rates, due to its capability of allowing different false positive rates on different modules. This result is even more evident if we compare the Bayesian detection rates for the different approaches at a false positive rate $P(A|\neg I) = 0.01$. The *a priori* probabilities are $P(I) = 0.985$ and $P(\neg I) = 0.015$. In case of the monolithic approach using the clustering algorithm proposed in [31], the detection rate $P(A|I) = 0.1837$ and the Bayesian detection rate is $P(I|A) = 0.2186$. In case of the monolithic ν-SVC, $P(A|I) = 0.1791$ and the Bayesian detection rate is $P(I|A) = 0.2143$. On the other hand, in case of the modular approach with ν-SVC classifiers $P(A|I) = 0.6796$[1], and the obtained Bayesian detection rate is $P(I|A) = 0.5085$, which is much higher than the Bayesian detection rate attained using the monolithic approach. Although more work has to be done in order to further increase the Bayesian detection rate, the modular approach is promising and should be considered as a basic scheme for the development of more accurate anomaly detection systems.

[1] This number was obtained by linear interpolation between the points (0.0091, 0.6731) and (0.0206, 0.7561)

Chapter 4

Learning in Adversarial Environment

As discussed in Chapter 3, learning from unlabeled traffic directly extracted from a live network is an inherently difficult pattern recognition problem. Besides the difficulties that characterize the unlabeled learning problem itself, we need to take into account how an adversary (i.e., an attacker) could interfere with the learning process. As the traffic does not undergo any labeling process (e.g., by a human expert) attackers may try to pollute the training traffic with properly crafted data (e.g., packets or connections) in order to mislead the learning algorithm and make the resulting detection model, and therefore the IDS itself, useless. In the following we present different strategies the adversary may use to interfere with the learning process, their theoretical effects, and possible countermeasures. We then present a case study which shows a practical example of how an adversary may affect the accuracy of intrusion detection schemes which are designed to protect against fast propagating worms. We analyze automatic signature generation algorithms which aim at learning "worm signatures" from (unlabeled) examples of worm flows. The generated signatures are used by worm detection systems in order to stop the propagation of the worm. We show how the attacker may inject properly crafted noise in the training dataset in order to mislead the signature learning process and make the generated signatures ineffective. In particular, we present an instance of the noise injection attack that can evade Polygraph [76], a recently proposed signature generation system. Polygraph is of particular interest for two reasons: a) it is able to generate signatures for worms that use a high level of polymorphism, and b) it constructs "Bayes signatures" which represent a statistical model of worm traffic and therefore can be used as the detection model for network

IDS based on statistical pattern recognition. We also present possible ad-hoc countermeasures to the proposed noise injection attack and discuss the reasons why we believe a thorough and robust solution to this type of attacks remains an open research problem.

4.1 Learning in Presence of Malicious Errors

To the best of our knowledge, the most significant theoretic study on learning in adversarial environment is [47]. Within the context of Valiant's Probably Approximately Correct (PAC) learning [106], Kearns et al. [47] analyze the problem of learning in the presence of an adversary that may introduce malicious errors in the data. The authors study the *optimal malicious error*, i.e., the largest value of the probability of error on the training data that can be tollerated by any learning algorithm for a certain representation class C [47]. They show that there exist representation classes for which the *optimal malicious error* rate can be achieved using simple polynomial-time algorithms. Their analysis is based on two-class problems. They refer to one of the classes as *positive* class and to the other as *negative* class, and prove that algorithms that learn from labeled examples of both the classes can tolerate more errors in the data compared to algorithms that learn form labeled examples of only one of the classes [47].

In [14], Barreno et al. discuss the security of machine learning algorithms applied to the development of IDS. They first propose a taxonomy of attacks against learning algorithms. According to the proposed taxonomy, they distinguish between *causative* and *exploratory* attacks [14]. Causative attacks aim to alter the training process by influencing the distribution of training data. Exploratory attacks do not aim to alter the training process, but aim to discover information about the learning algorithm through probing techniques. Within the class of causative attacks, the authors further distinguish between *integrity* and *availability* attacks. The objective of *causative integrity* attacks is to mislead the learning algorithm in order to prevent the IDS to detect future intrusions. On the other hand, *causative availability* attacks aim to force the IDS to make a sufficient amount of errors, so that it becomes useless and will be likely turned off by the administrator. Afterwards, an example of causative integrity attack against an anomaly detector based on a simple anomaly detection algorithm is described. The simplicity of the learning algorithm allows the authors to analytically

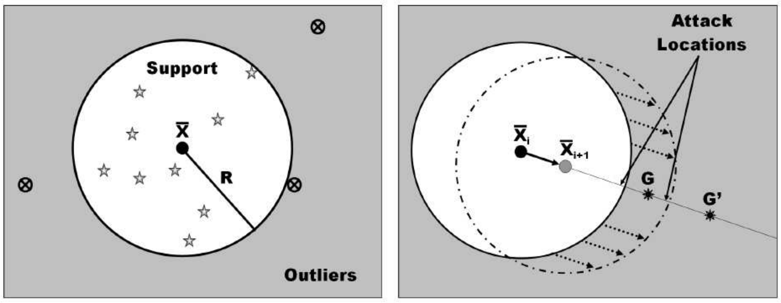

Figure 4.1: Causative attack against a learning algorithm. G and G' represent two different attack instances. The training dataset is polluted by placing data points along the line that connects the center of the sphere to G and G'. Different amounts of well crafted data points are needed to mislead the algorithm and shift the decision surface so that G first, and then G', are not detected as anomalous [14].

study the problem of learning from polluted traffic and find a bound on the effort required by the attacker to mislead the IDS so that future attacks will not be detected. The considered anomaly detection algorithm constructs a hypersphere around the normal data. During the operational phase, the instances that lay outside the sphere are classified as anomalous [14]. As shown in Figure 4.1, the objective of the attacker is to pollute the training data so that eventually an attack instance G will lay inside the sphere, which means that the attack is not detected by the IDS. Assuming the attacker knows the set of features used to describe the traffic, the learning algorithm, and the current state of the IDS, the attack strategy is to inject properly crafted instances in the traffic in order to force the hypersphere to shift towards G, until it lays inside the decision surface [14].

Barreno et al. [14] also propose possible countermeasures to the causative attacks. For example they propose to implement *disinformation* and *randomization* strategies. Disinformations consists in somehow lying to the attacker, whereas randomization etails introducing some level of randomization in the paprameters used to train the model of normal traffic, so that it is difficult for the attacker to learn or guess the actual state of the IDS (i.e., where the decision surface is placed). This may make launching causative attacks more difficult. However, we believe countermeasures are application dependent and are not always applicable or effective, as we discuss in Section 4.6.

4.2 Case Study: Misleading Worm Signature Generators

In the last few years, large worm outbreaks have pointed out the inadequacy of today's network security systems. The now famous Code Red worm, released in July 2001, infected more than 360,000 hosts in less than 14 hours [73], whereas the Slammer worm, released in January 2003, was able to infect more than 90% of the vulnerable population in less then 10 minutes [72]. More recent worms are able to propagate through multiple vectors [87, 86] and to use mutation techniques in an attempt to create variants which are difficult to detect by using traditional signature-based IDS [108, 26]. In 2002, Staniford et al. discussed the risks related to the realistic ability of an attacker to gain control of an enormous number of Internet hosts and anticipated the concept of "flash-worms", which would be able to infect the entire vulnerable population in tens of seconds [97].

A number of techniques have been proposed in order to try to limit the propagation of aggressive worms, including anomaly detection [103, 113], dynamic quarantine [119, 115], automatic signature generation [53, 48, 94, 76, 117, 78, 99], address space and instruction-set randomization [16, 46]. Among these, automatic signature generation systems have recently gained substantial interest within the computer security research community.

Signature generation is a key step in the defense against worm propagation. Most of the signatures used by firewalls or signature-based intrusion detection systems (IDS) are created using a manual analysis of worm traffic flows. This is usually a time-consuming process, and thus cannot keep pace with rapidly spreading worms. Manual analysis becomes even harder and more time-consuming if the worms use metamorphism and polymorphism techniques. Automatic signature generation is a promising alternative. The goal is to automatically, and thus very quickly, learn worm signatures by extracting the invariant parts of examples of worm flows collected in the wild.

Early approaches [53, 48, 94] are based on syntactic analysis of suspicious traffic flows. These approaches have limited abilities to learn (or extract) reliable signatures from truly polymorphic worms. Newsome et al. recently proposed two approaches to address this problem [76, 78]. Polygraph [76] is based on syntactic analysis of suspicious traffic flows, and implements three different types of signature generation algorithms. Taint analysis [78] is a semantic analysis approach based on the execution of possible vulnerable applications inside a protected environment.

We will focus on signature generation systems that aim at automatically learning and deploying signatures that could be used by firewalls or network IDS. Other automatic signature generators are based on the extraction of *host-based* signatures that need to access the execution or application environment they are trying to protect in order to be effective, as proposed for example in [64]. We do not discuss these systems here. We will examine the abilities of syntactic-based automatic signature generators in the face of advanced polymorphic worms that not only spread using a high level of polymorphism but also deliberately *mislead* the learning process in order to prevent the resulting signatures from stopping its propagation.

Using Polygraph [76] as a case study, we introduce a class of attacks whereby a worm can combine polymorphism and misleading behavior in order to interfere with the learning process and disrupt the generation of reliable signatures. We will show that this result can be achieved by intentionally injecting properly crafted noise into the training dataset of suspicious flows used by syntactic-based signature generators to learn worm signatures. We will present a specific instance of the attack that can mislead Polygraph, and then we will discuss how such noise injection attacks are general in that different attacks can be devised to mislead other recently proposed automatic signature generators. According to the taxonomy in [14], the attacks we present are causative attacks against signature-based IDS which use automatically generated worm signatures to stop worm propagation.

The system architecture of Polygraph includes a flow classifier module and a signature generation module [76]. The flow classifier collects the suspicious and the innocuous flows from which the signatures are learned. The authors assumed that the flow classifier can be imperfect and that it can introduce some noise into the pool of suspicious flows, regardless of the classification technique used by the flow classifier. The authors then proposed some techniques to cope with the noise during the signature generation process. This design characteristic is common to most of the syntactic-based automatic signature generators. That is, little or no attention is paid to filtering the noise during the suspicious flow gathering process. This is a serious shortcoming that can be exploited by combining polymorphism and misleading behavior. We will show how a *misleading polymorphic worm* can create and send *fake anomalous* flows during its propagation to deliberately pollute the set of flows used to extract the signatures. Polygraph's authors state that their system is resilient to (at least)

80% of noise into the set of suspicious flows [76]. We will show that by constructing well-crafted *fake anomalous* flows, a worm can mislead the signature generation algorithms by injecting much less than 80% of noise into the set of suspicious flows, thus preventing the generation of useful signatures. We would like to emphasize that although we demonstrate the effects of the noise injection attack on Polygraph, which is used as a case study here, it is a general attack on all the syntactic-based signature generation systems proposed in the literature because they do not addresses directly the problem of intentional pollution of the dataset of suspicious flows. In particular, we will discuss how the attack can be generalized to defeat other recent automatic signature generation systems, and why it cannot always be prevented by even *semantic-based* approaches similar to [78].

4.3 Noise Injection Attack

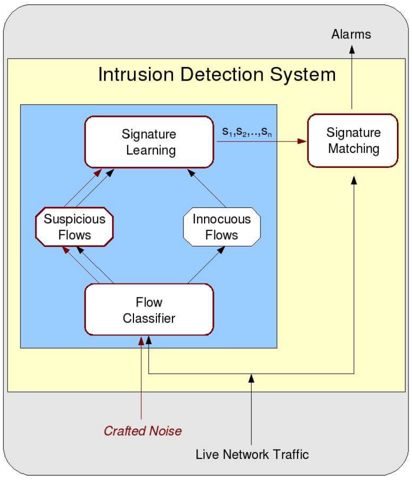

Figure 4.2: Worm signature generation and detection scheme.

Noise injection attack works by polluting the training set of suspicious traffic flows, or *suspicious flow pool* [48, 76], used by automatic signature generators in the signature learning (or extraction) process (see Figure 4.2). The attack aims to mislead the signature generation algorithms by injecting *well-crafted* noise to prevent the generation of useful signatures. In the following sections we briefly survey the most common techniques used by a “flow classifier” to collect the suspicious flows. We then show how the worm can inject noise without *a priori* knowledge about the classification technique in use. To accomplish the task of misleading the signature generation algorithms, the noise has to be crafted in a suitable manner. Different noise injection attacks can be implemented by crafting the noise in different manners. We first demonstrate how this attack can be implemented against Polygraph [76], and then analyze the possible effects of noise injection attack on Nemean [117], another recently proposed automatic signature generator. Different implementations of the attack can be devised to mislead other signature generators.

4.3.1 Collecting Suspicious Flows

A few techniques have been proposed to accomplish the task of collecting the suspicious flows, which represent (part of) the training dataset used for learning the signatures. Honeycomb [53] uses a simulated honeynet. Any flow sent towards the honeynet is inserted into the suspicious flow pool. Nemean [117] uses a similar approach combining real and simulated hosts. In [99] a double honeynet is proposed. In this case a first-layer honeynet is made of real hosts. Whenever a first-layer honeypot is infected by a worm, its outgoing traffic is redirected to a second-layer simulated honeynet and inserted into the suspicious flow pool. Autograph [48] implements a classification approach based on port-scanning detection. Each valid flow sent by a scanner to a valid IP address is inserted into the suspicious flow pool. Anomaly-based IDS can also be used as flow classifiers. For example, PAYL [112] uses the byte frequency distribution of the normal packets to detect anomalies, and can be used as a flow classifier.

There are other techniques that are not considered in our study. Earlybird [94] extracts all the possible substrings of a given fixed length β from each packet to compute the content prevalence. β cannot be reduced to just a few bytes due to computational complexity and memory consumption problems. As shown in [76], a polymorphic worm can contain invariants that are just two or three bytes long, potentially evading Earlybird. Since our study focuses on *misleading* polymorphic worms that try to mislead signature generators, we must assume that the flow classifier can detect polymorphic worm instances as suspicious flows. Approaches for run-time detection of injected code, e.g., [78, 64, 46, 16] are not considered because they are largely limited to *application-based* worms (e.g., CodeRed [73], Slammer [72], etc.) and are not effective against *OS-based* worms (e.g., Sasser [88], Zotob [89], etc.). We are concerned with general-purpose worms. More importantly, these approaches are “host-based” while almost all the automatic signature generators presented in literature use “traffic-based” flow classifiers.

4.3.2 Injecting Noise into The Suspicious Flow Pool

Suppose a worm has infected a host in network *A* and is now trying to infect some hosts in network *B*. Suppose also that each time the worm sends a polymorphic instance to a host in *B*, it also sends a *fake anomalous* flow to the same host, as shown in Figure 4.3. Section 4.3.3 provides details on the creation of *fake anomalous* flows. For now consider that the *fake anomalous* flow does not need to exploit the vulnerability and thus can be crafted in a very flexible manner to appear like the real worm in all but the invariant parts (which are necessary to exploit the vulnerability). For example a *fake anomalous* flow can be crafted so that it contains the same protocol framework as the worm (e.g., a GET request) and the same byte frequency distribution, and at the same time not containing the real worm’s invariants.

Suppose the network B is monitored by a “traffic-based” flow classifier. The worm and its *fake anomalous* flow must both be stored in the suspicious flow pool in order to mislead the signature generation algorithm. This is possible with the flow classifiers we consider (see Section 4.3.1). We describe how this can be accomplished with each of the flow classifiers below:

- **Honeynet**. In this case the vulnerable host that the worm is trying to infect can be a real or simulated honeypot. Since both the real worm and the *fake anomalous* flow are sent to the same destination at (roughly) the same time, they will both be considered suspicious by the honeypot and stored into the suspicious flow pool.

- **Double honeynet**. In this case the real worm will infect a first-layer honeypot, whereas the *fake anomalous* flow will not, and will be disregarded. However, only the outgoing traffic will be redirected to the second-layer simulated honeypot and stored into the suspicious flow pool. Given that the outgoing traffic generated by the worm instance at the first-layer honeypot will again contain both a real worm flow and another fake anomalous flow, they will be stored into the suspicious flow pool together.

- **Port-scanning detection**. If the worm scans more than *s* unused IP addresses, the source of

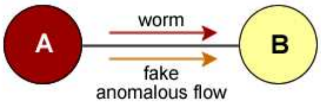

Figure 4.3: Worm propagation

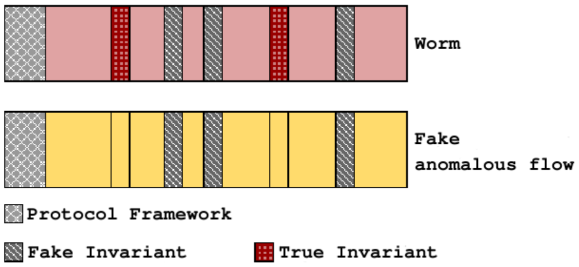

Figure 4.4: Structure of the flows (simplified)

the scanning (i.e., the infected host in *A*) will be considered a scanner. Therefore, each flow sent by the infected host in *A* towards *B* after the scanning phase will be considered suspicious. Given that the real worm and the *fake anomalous* flow originate from the same source host, they will be both inserted into the suspicious flow pool.

- **Byte frequency-based classifier**. The *fake anomalous* flow can be easily crafted to match the byte frequency distribution of the real worm flow (as discussed in Section 4.3.3). This means that if the real worm flow is flagged as anomalous, its *fake anomalous* flow will very likely be flagged as anomalous, too. Thus, both the worm and the *fake anomalous* flow will be stored into the suspicious flow pool.

Note that each copy of the worm could craft and send more than one *fake anomalous* flow at the same time. In this case the real worm flow and all its *fake anomalous* flows will be inserted into the suspicious flow pool together. The discussion above suggests that without a semantic-based analysis it is not possible to distinguish between the real worm flow and its fake anomalous flows.

4.3.3 Crafting the Noise: A Case Study Using Polygraph

In this section we present a noise injection attack devised to mislead Polygraph [76]. In order to explain how the noise can be crafted to mislead Polygraph we first describe the high level structure of a polymorphic worm and how Polygraph extracts worm signatures.

High Level Structure of A Polymorphic Worm

As discussed in [52] and in [76], a polymorphic worm is made of the following components:

- **Protocol framework**. In many cases the vulnerability is associated with a particular execution path in the application code. In turn, this execution path can be activated by one (or just a few) particular request type(s). Therefore, the protocol framework is usually common to all the worm variants. However, in some cases it may still be possible to modify the attack vector, thus reducing the number of invariants.
- **Exploit's invariant bytes**. These bytes have a fixed value that cannot be changed because they are absolutely necessary for the exploit to work.
- **Wildcard bytes**. These bytes can assume any value without affecting the exploit.
- **Worm's body**. It contains the instructions the worm executes once the vulnerability has been exploited. If the worm uses a good polymorphic engine, these bytes can assume different values in each worm copy. Common techniques to achieve body (shellcode) polymorphism include register shuffling, equivalent instruction substitution, instruction reordering, garbage insertions, and encryption. Different keys can be used in encryption for different instances of the attack to ensure that the body's byte sequence is different every time.
- **Polymorphic decryptor**. It contains the first instructions to be executed after the vulnerability has been exploited. The polymorphic decryptor decodes the worm's body and then jumps to it. Obviously, the decryptor itself cannot be encrypted. However, polymorphism of the decryptor can be achieved using various code obfuscation techniques.

Note that this is a simplified view.

Polygraph's Signature Generation Module

Polygraph consists of several modules [76]. A flow classifier performs flow reconstruction and classification on packets received from the network. The flows deemed suspicious are stored into a *suspicious flow* pool, whereas the flows deemed innocuous are stored into an *innocuous flow* pool. The signature generator module uses both pools during the signature generation process. The objective of Polygraph [76] is to extract the invariant parts of a polymorphic worm using three different signature generation algorithms. We briefly summarize how these algorithms work.

- **Conjunction signatures**. During the preprocessing phase the substrings common to all the flows in the suspicious flow pool are extracted. These substrings are called *tokens*. A conjunction signature is made of an unordered set of tokens. A flow matches the signature if it contains all the tokens in the signature.

- **Token-Subsequence signature**. As with the conjunction signatures, the set of tokens in common among all the suspicious flows are extracted. Then, each suspicious flow is rewritten as a sequence of tokens separated by a special character γ. A string alignment algorithm creates an ordered list of tokens that is present in all the suspicious flows. A token-subsequence signature consists of the obtained ordered list of tokens. A flow matches the signature if the ordered sequence of tokens is in the flow.

- **Bayes signatures**. All the tokens of a minimum length α that are common to at least K out of the total number N of suspicious flows are extracted. Then, for each token t_i, $p(t_i|Suspicious\ flow)$ and $p(t_i|Innocuous\ flow)$, the probabilities of finding the token in a suspicious flow and in an innocuous flow, respectively, are computed. A score

$$\lambda_i = log\left[\frac{p(t_i|Suspicious\ flow)}{p(t_i|Innocuous\ flow)}\right]$$

  is then assigned to each token t_i. The probability $p(t_i|Suspicious\ flow)$ is estimated over the suspicious flow pool, whereas $p(t_i|Innocuous\ flow)$ is estimated over the innocuous flow pool. During the match process, the scores λ_i for the tokens t_i contained in the flow under test are summed. The flow matches the signature if the obtained total score Λ exceeds a precomputed threshold θ. This threshold is computed during the signature generation process. Given a predetermined acceptable percentage of false positives r, θ is chosen so that the signature produces less than r false positives and minimizes the number of false negatives at the same time.

The conjunction and token-subsequence signatures are not resilient to noise in the suspicious flow pool. For example, if just one noise flow that does not contain the worm's invariants appears in the suspicious flow pool, the worm's invariants will not be extracted during the preprocessing phase because they are not present in *all* of the flows. For this reason Polygraph [76] applies a hierarchical clustering algorithm during the generation of conjunction and token-subsequence signatures in an attempt to isolate the worm flows from the noise. Each cluster consists of a set of suspicious flows $\{a_1, a_2, .., a_n\}$, and the signature s_a extracted from the set. That is, each cluster can be represented as a pair $(\{a_1, a_2, .., a_n\}, s_a)$. The similarity between two clusters is based on the *specificity* of the signatures, namely, the number of false positives (measured over the innocuous flow pool) produced by the new signature obtained by merging the two clusters. For example, the similarity between two clusters $(\{a_1, a_2, .., a_n\}, s_a)$ and $(\{b_1, b_2, .., b_m\}, s_b)$ is computed as the number of false positives produced by the signature $s_{a,b}$ extracted from the merged set of flows $\{a_1, a_2, .., a_n, b_1, b_2, .., b_m\}$. The algorithm starts with N clusters, one for each suspicious flow, and then proceeds iteratively to merge pairs of the (remaining) clusters. At each step, only the one pair of clusters that upon merging produce the signature with the lowest false positive rate are actually merged. The algorithm proceeds until all the "merged" signatures produce an unacceptable number of false positives or there is only one cluster left.

From a statistical pattern recognition point of view, the tokens represent the features used to describe network flows. In case of conjunction and token-subsequence signatures a flow is described using binary features which encode the presence or absence of tokens in the flow, whereas in case of Bayes signatures the value of each feature represents a score computed according to the probability of finding a token in normal and worm flows, as described above. A signature represents a prototype to which network flows are compared during the detection (or recognition) phase.

Misleading Conjunction and Token-Subsequences Signatures

A signature is useful if it contains at least a subset of the invariant substrings of the worm. The hierarchical clustering algorithm implemented by Polygraph is greedy [76]. This choice is motivated by the fact that a non-greedy clustering algorithm would be computationally expensive. This property can be exploited by injecting well-crafted noise to prevent the generation of a useful signature. Below, we describe how to craft the noise to mislead Polygraph.

Suppose that a polymorphic worm propagates using the scheme described in Section 4.3.2 (see Figure 4.3). Suppose also that the *fake anomalous* flow is crafted so that it has some substrings in common with the real worm, but does not contain the *true* invariant parts of the worm, as shown in Figure 4.4. We call TI (True Invariants) the set of *true* invariant substrings, and FI (Fake Invariants) the set of substrings in common between the worm and its fake anomalous flow. Suppose now that the suspicious flow pool contains three copies of the worm, and then also three corresponding *fake anomalous* flows. We call w_i the *i*-th copy of the worm in the suspicious flow pool and f_i its *fake anomalous* flow. Note that FI_i is different for different pairs of w_i and f_i because each fake anomalous flow is crafted specifically according to a worm flow, and each worm flow is different due to polymorphism.

The clustering algorithm starts (at step 0) by constructing one signature for each (single) flow in the suspicious flow pool. During the first step of the clustering process, whenever a worm flow w_i and the corresponding fake anomalous flow f_i are considered together, a signature containing the common substrings FI_i will be generated. It is worth noting that the generated signature in this case will not contain TI. Whenever two worm flows w_i and w_j are considered together, a signature containing TI will be generated. Whereas, whenever two fake anomalous flows f_i and f_j or a worm flow w_i and a fake anomalous flow f_j ($j \neq i$, i.e, it is from a different worm flow) are considered together, the generated signature will contain just substrings extracted from the protocol framework PF (and possibly other substrings that are in common just by chance). Obviously, a signature containing mostly tokens extracted from the protocol framework would produce a high number of false positives because the normal/innocuous flows will also need to use the protocol/application and thus can also contain substrings of the protocol framework. Therefore, pairs of w_i and f_j and pairs of f_i and f_j ($i \neq j$) will not be merged. Now, the question is whether a pair of w_i and f_i (resulting in a signature containing FI_i) or a pair of w_i and w_j (resulting in a signature containing TI) will be merged.

Let $p(false\ positive|FI_i)$ and $p(false\ positive|TI)$ be the probabilities that a signature containing FI_i and a signature containing TI will produce a false positive, respectively. If the fake invariants FI_i had been "well-crafted" by the worm during propagation so that $p(false\ positive|FI_i) < p(false\ positive|TI)$, the "merged" signature s_1, produced by the first step of the clustering algorithm (see above) will contain FI_i but will not contain TI. That is, a worm flow and its corresponding fake anomalous flow, say, w_1 and f_1 will be merged. Of course, the question is how to obtain $p(false\ positive|FI_i) < p(false\ positive|TI)$. In Section 4.3.3, we will describe how to produce, in practice, a fake anomalous flow that corresponds to a true worm flow. For now, we state that the FI_i tokens are made of random bytes and that the total number and the lengths of tokens in FI_i are greater than the number and the lengths of tokens in TI. As a result, $p(false\ positive|FI_i) < p(false\ positive|TI)$ will be very likely to hold. To show this, let $p_f(b)$ be the probability of a byte b, contained in a fake invariant token, to appear in an innocuous flow, and $p_t(b)$ the probability of a byte b, contained in a true invariant token, to appear in an innocuous flow. Let the cardinalities of the sets FI_i and TI be $x = |FI_i|$ and $y = |TI|$, respectively, and the lengths of a token $t_{f_k} \in FI_i$ and a token $t_{t_k} \in TI$ be l_k and h_k, respectively. Assuming the bytes of a token to be extracted from a uniform random distribution and assuming the tokens to be statistically independent, we can write:

$$\begin{aligned} p(false\ positive|FI_i) &= \textstyle\prod_{k=1}^{x} \prod_{j=1}^{l_k} p_f(b_{k,j}) \\ p(false\ positive|TI) &= \textstyle\prod_{k=1}^{y} \prod_{j=1}^{h_k} p_t(b_{k,j}) \end{aligned} \tag{4.1}$$

where $b_{k,j}$ is the j-th byte of the k-th token. Now, if we assume that the bytes $b_{k,j}$ have the same probability, p, to be present in an innocuous flow, so that $p_f(b_{k,j}) = p_t(b_{k,j}) = p$, $\forall_{k,j}$, it is easy to see that if $x \cdot avg_k(l_k) > y \cdot avg_k(h_k)$ we can obtain $p(false\ positive|FI_i) < p(false\ positive|TI)$.

Now, returning to the clustering process. At this point, there is one cluster, say, $(\{w_1, f_1\}, s_1)$, and two worm flows and two fake anomalous flows. Consider all the candidates for merging. We already know from the above discussion that if we only consider the four clusters containing a single flow, the only acceptable merging will be between a worm flow and its corresponding fake anomalous flow, say w_2 and f_2, resulting in a signature containing FI_2. But w_2 (or f_2) can also merge with the existing cluster, resulting in a set $\{w_1, f_1, w_2\}$ (or $\{w_1, f_1, f_2\}$). By extracting the substrings common to all the three flows the algorithm would obtain only tokens belonging to the protocol framework (and possibly other small substrings that are common to all three flows just by chance). We call CS_{ij} the signature extracted from $\{w_i, f_i, w_j\}$ (or $\{w_i, f_i, f_j\}$). Note that $TI \not\subseteq CS_{ij}$. Again, $p(false\ positive|FI_j) < p(false\ positive|CS_{ij})$ will very likely hold given that CS_{ij} will mostly contain just tokens from the protocol framework. Therefore, the only acceptable cluster is $\{w_2, f_2\}$.

The algorithm continues and finally there will be three clusters, namely $\{w_1, f_1\}$, $\{w_2, f_2\}$ and $\{w_3, f_3\}$, and three corresponding signatures. At this point, the clustering algorithm will consider merging the clusters, say, to form $\{\{w_1, f_1\}, \{w_2, f_2\}\}$. But the set of substrings in common among all the four flows will not contain TI. Once again, the signature will mostly contain invariants related to the protocol framework, and as a result will likely produce a high number of false positives. Thus, this cluster is not acceptable, and the clustering algorithm has to terminate.

In conclusion, the noise injection attack misleads Polygraph to generate signatures containing the fake invariant strings (FI_i), rather than a useful signature containing the true invariants (TI).

Misleading Bayes Signatures

To generate Bayes signatures, Polygraph first extracts the tokens of a minimum length α that are common to at least K out of a total number of N suspicious flows. If $K = 0.2 \times N$, as suggested in [76], an attacker can mislead the Bayes signatures by simply programming the worm so that it sends five fake anomalous flows per worm variant because in this case the true invariants (TI) occur in less than 20% of the suspicious flows and will not be extracted/used. It seems then that for a low value of K the worm needs to flood the suspicious flow pool with a large number of fake anomalous flows. However, we show how the worm can craft the fake anomalous flows so that just a few (one or two) of them per worm variant will be sufficient to mislead the generation of Bayes signatures.

If a worm crafts the fake anomalous flows as described in Section 4.3.3, the Bayes signature generation algorithm will very likely generate a *useful* worm signature containing tokens related to the protocol framework PF and the true invariant tokens TI. The tokens PF will be present in 100% of the suspicious flows, whereas the tokens TI will be present in 50% of the suspicious flows if one fake anomalous flow per worm variant is used. The fake invariants FI are specific for each worm variant and its fake anomalous flow. This means each FI_i will, in general, be present less than K times in the suspicious flow pool (unless K is very small) and will not be used to generate the Bayes signatures. In short, the technique described in Section 4.3.3 cannot mislead Bayes signatures.

As described in Section 4.3.3, during the generation of a Bayes signature a score λ_i is computed for each token t_i in the signature. During the matching process, the scores of matched tokens are summed. The technique we develop here is to insert a set of strings in the fake anomalous flows in such a way that the generated signatures contains tokens that will score an innocuous flow higher than a true worm flow, thus making it very hard to set a proper threshold value (θ) to obtain both low false positive and false negative rates.

Consider now a length n string of bytes $v = (v_1, v_2, ..v_n)$ that appears in the innocuous flow pool (but does not appear in the worm flows) with a probability p that is neither too low nor too high, for example $p_1 = 0.05 < p(v|Innocuous\ flow) < 0.20 = p_2$. If v is injected into the fake anomalous flows generated by each variant of the worm, this string will appear in at least 50% of the suspicious flows. This means that the string v will be considered as a token in the Bayes signature. We have $p(v|suspicious\ flow) \geq 0.5$ and $p_1 < p(v|Innocuous\ flow) < p_2$, thus the token v would receive a score λ_v between $log(0.5/p2)$ and $log(0.5/p_1)$. If we split the string v to all the possible substrings of length $m < n$, we will obtain $n{-}m{+}1$ different substrings $v_{1,m} = (v_1, v_2, ..v_m)$, $v_{2,m+1} = (v_2, v_3, ..v_{m+1})$, ..., $v_{n-m+1,n} = (v_{n-m+1}, v_{n-m+2}, ..v_n)$. Suppose now the worm injects all of the $n - m + 1$ substrings randomly (with respect to the position for each substring) in each fake anomalous flow, instead of injecting the entire string v. All of the substrings of v will be present in at least 50% of the suspicious flows in the suspicious flow pool and will therefore be added as tokens into the Bayes signature.

If m is not much lower than n, we can expect that $p(v_{j,j+m-1}|Innocuous\ flow)$ will be not much higher than $p(v|Innocuous\ flow)$. In turn, we expect the score $\lambda_{v_{j,j+m-1}}$ associated with each of the $n - m + 1$ substrings of v to be not much lower than the score λ_v. This results in a multiplying effect on the score of v because a flow that contains v also contains all of its substrings. We will refer to the strings $v_{j,j+m-1},\ j = 1..(n - m + 1)$ as *score multiplier strings*.

The Bayes signatures now include PF, TI and the score multiplier strings. During the matching phase, the total score for a real worm flow is:

$$S = \sum_l \lambda_{PF_l} + \sum_h \lambda_{TI_h} \tag{4.2}$$

Here λ_{TI_h} is the score of a worm's true invariant token TI_h and λ_{PF_l} is the score of a protocol framework token PF_l (note that the worm will not contain v).

On the other hand, the total score for an innocuous flow containing v is *at least*:

$$\Lambda = \sum_{j=1}^{n-m+1} \lambda_{v_{j,j+m-1}} \tag{4.3}$$

The innocuous flow contains v and thus all of its substrings, which are tokens in the Bayes signatures (the flow can also contain PF tokens etc.) If the attacker chooses v and m such that $\Lambda > S$, it will be impossible to set a threshold θ for the Bayes signatures that will produce a low false positive rate and low false negative rate at the same time. This is because if $\theta < S$ (and then also $\theta < \Lambda$) the signature will generate a high number of false positives (from around 5% to 20% for the proposed example $0.05 < p(v|Innocuous\ flow) < 0.20$), due to the presence of v, and then of all its substrings, into a non-negligible percentage of normal traffic. On the other hand, if $\theta > \Lambda$ (and then also $\theta > S$) the Bayes signature will produce around 100% false negatives.

In conclusion, the attacking technique described here prevents the generation of a useful signature. We will discuss in Section 4.4 how the attacker can automatically extract a set of *candidate* strings v (and therefore its *score multiplier* substrings) from network traffic traces. The obtained candidate strings can be used to obtain the multiplying effect explained above.

Crafting The Noise

Before propagating to the next victim the worm must first create a polymorphic copy of itself w_i. Then it can create the associated fake anomalous flow f_i using the following algorithm:

a) $f_i^{(0)}$ = **clone**(w_i): Create a copy of w_i.

b) $f_i^{(1)}$ = **randomlyPermuteBytes**($f_i^{(0)}$): Permute the bytes of $f_i^{(0)}$ but leaving the protocol framework bytes unchanged.

c) a[ ] = **extractFakeInvariants**(w_i,k,l): Copy k substrings of length l from w_i into an array a, choosing them at random, but do not copy substrings that contain protocol framework or true invariant bytes.

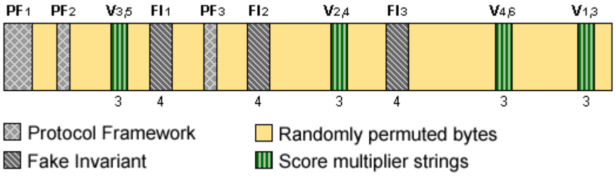

Figure 4.5: An example of fake anomalous flow

d) $f_i^{(2)}$ = **injectFakeInvariants**($f_i^{(1)}$,a[ ]): Copy the fake invariant substrings into $f_i^{(1)}$ but do not overwrite bytes belonging to the protocol framework (see Figure 4.4).

e) $f_i^{(3)}$ = **injectScoreMultiplierStrings**($f_i^{(2)}$,v): Inject *score multiplier strings* in $f_i^{(2)}$ by splitting a string v as explained in Section 4.3.3. The string v can be chosen from a set of candidate strings obtained by means of an analysis of normal network traffic traces performed using the algorithm explained in [81]. The attacker could embed a subset of the candidate strings into the worm's code. The decision on which string v to use can be based on time. For example, the worm could embed the time of its first infection into its code and then use a different string v periodically (e.g., every 10 minutes for a fast-propagating worm). This is necessary because the worm and its fake anomalous flows can arrive at the flow classifiers from multiple infected hosts. Given that the *score multiplier strings* have to be present in a high fraction of the total number of fake anomalous flows into the suspicious pool, the worm cannot just pick v at random each time it propagates to a new victim. Instead, each v has to be used for a period of time.

f) $f_i^{(4)}$ = **obfuscateTrueInvariants**($f_i^{(3)}$): This is necessary because $f_i^{(3)}$ could still contain some true invariant strings, even though just by chance. The obfuscation process assures that $f_i^{(4)}$ will not contain the worm's true invariants.

Here $f_i^{(h)}$ represents an "update" of $f_i^{(h-1)}$. The final fake anomalous flow $f_i^{(4)}$ and the worm variant w_i are sent together to the next victim. An example of the application of the above algorithm is reported in Figure 4.5. The fake anomalous flow has been crafted using $k = 3$ fake invariants of length $l = 4$. The string v is 6 bytes long and the length of the *score multiplier substrings* is $m = 3$. It is worth noting that the resulting *fake anomalous* flow does not contain the true invariant tokens.

If the byte frequency distribution of w_i and f_i are not very close (due to the injection of the score multiplier strings) a simple padding technique could be applied to make the two byte frequency distribution closer.

Combining Noise Injection and Red Herring Attacks

In Section 4.3.3 we presented how the fake anomalous flows can be crafted to mislead the generation of Conjunction and Token-subsequences signatures. For such attack to be successful, fake anomalous flows generated by different worm variants should not contain common substrings. The attacking method presented in Section 4.3.3 to mislead the generation of Bayes signatures violates this constraint because all the fake anomalous flows in the suspicious flow pool have to contain the same *score multiplier strings*. However, this turns out not to be a problem. During the application of the hierarchical clustering algorithm, whenever two fake anomalous flows f_i and f_j are involved in a merge, the extracted tokens will be either part of the protocol framework or *score multiplier* substrings. Therefore, the generated signature will very likely produce a high number of false positives and the flows will not be kept in the same cluster. It is then very likely to see (following the analysis in Section 4.3.3) that the only acceptable clusters are $\{w_i, f_i\}$. Thus, the attack against Bayes signatures described in Section 4.3.3 does not interfere with the attack against Conjunction or Token-subsequence signatures. It follows that crafting the fake anomalous flows as described in Section 4.3.3, the attack is effective against the three different types of Polygraph signature generation algorithms.

However, the results of the attack are not deterministically predictable. As mentioned in Section 4.3.3 it is possible that a set of flows contains some substrings that are common just by chance to all the flows in the set. For example it could happen that two worm variants w_i and w_j present (by chance) a common substring $c_{i,j}$, besides the protocol framework and true invariant tokens. This means that to avoid w_i and w_j being kept in the same cluster, the constraint $p(false\ positive|FI) < p(false\ positive|TI, c_{i,j})$ needs to be verified. Given that $c_{i,j}$ is unknown, it is not easy to craft the set of fake invariants FI to assure that this constraint is satisfied. Besides, even if the worm crafts FI so that $p(false\ positive|FI)$ is close to zero, it can also happen that $p(false\ positive|TI, c_{i,j}) = 0$. In this case there is no way to determine which signature is more specific than the other, and we assume the merged cluster to be kept is chosen at random.

We will show in Section 4.4 that in practice the probability of success for the noise injection attack is fairly high. To further increase the success chance of the noise injection attack, it is possible to combine it with the *red herring* attack discussed by Polygraph's authors in [76]. The worm variants could include some *temporary true invariants* that change over time. If the Conjunction and Token-subsequence signature generation algorithms produce (by chance) a useful signature, this signature would become useless over a certain period of time. After this period of time Polygraph could try to generate again new Conjunction and Token-subsequence signatures to detect the worm. Nevertheless, this time Polygraph may not be as "fortunate" as the first time in generating a useful signature. Besides, if the *temporary true invariants* were chosen among high frequency strings (e.g., extracted from network traces using the algorithm presented in [81] setting the probability between 0.8 and 1), the related tokens would receive a low score during the generation of the Bayes signature and therefore would not interfere with the noise injection attack against Bayes signatures. The final result is that the attacker has a very high probability to succeed in misleading all the three types of signatures at the same time.

4.3.4 Effects of the Noise on Other Automatic Signature Generators

We have performed experiments only on Polygraph. However, it is possible to evaluate the effects of different noise injection attacks on other systems basing the analysis on the description of the signature generation algorithms. We present an analysis of the possible effects of noise injection attack on Nemean [117].

Nemean is a recently proposed automatic signature generator that uses a semantic analysis of the network protocols and two types of signatures, namely connection and session signatures [117]. It uses a honeynet to collect the suspicious flow pool. Then it applies a clustering algorithm to group similar connections in *connection clusters* and similar sessions in *session clusters*. Each cluster contains the observed variants of the same worm. Even though Nemean is suitable for generating signatures for worms that use limited polymorphism [117], it introduces interesting features such as semantic protocol analysis and connection and session clustering. For this reason, it is interesting to discuss how it could be misled using the noise injection attack.

Nemean represents a connection by a vector containing the distribution of bytes, the request type and the response codes that occurred in the network data [117]. The *fake anomalous* flows can be injected into the suspicious flow pool as explained in Section 4.3.2. Given that the *fake anomalous* flows can be crafted to have the same protocol framework and (almost perfectly) the same distribution of bytes as the worm variant they derive from, the *fake anomalous* flows and the worm variants will be very likely considered in the same connection cluster. If the *fake anomalous* flows are crafted by applying a random permutation of the worm's bytes (see Section 4.3.3), the signature generation algorithm will not be able to discover significant invariant parts common to the flows in a cluster, and the extracted connection signatures will be useless because they will likely produce a high number of false positives. This noise injection attack will affect the session signatures as well, given that they are constructed based on the results produced by the connection clustering process [117].

4.4 Experiments

In our experiments we tried to have an experimental setup similar to the one reported in [76] in order to make the results comparable. Polygraph software is not publicly available, therefore we implemented our own version following the description of the algorithms in [76].

4.4.1 Experimental Setup

Polygraph setup. We performed all the experiments setting the minimum token length $\alpha = 2$ and the token-extraction threshold for Bayes signature generation to be 20% of the total size of the suspicious flow pool. We also set the minimum cluster size to 3 and the maximum acceptable false positive rate for a signature to be 0.01 during the application of the hierarchical clustering algorithm for Conjunction and Token-subsequences signatures.

Polymorphic worm. We considered the Apache-Knacker exploit reported in [76] as the attack vector for the worm. We simulated an ideal polymorphic engine following the same idea used by Polygraph's authors, keeping the protocol framework of the attack and the first two byte of the return address fixed and filling the wildcard and code bytes uniformly at random. Each worm variant matches the regular expression:

```
GET .* HTTP/1.1\r\n.*\r\nHost: .*\r\n.*\r\nHost: .*\xFF\xBF.*\r\n
```

Datasets. We collected several days of HTTP requests taken from the backbone of a busy aggregated /16 and /17 academic network (i.e., CIDR [35] blocks of the form `a.b.0.0/16` and `c.d.e.0/17`) hosting thousands of machines. The collected traffic contains requests towards thousands of different public web-servers, both internal and external with respect to our network. The network traffic traces were collected between October and November 2004. We split the traffic traces to obtain three different datasets which are described below.

Innocuous flow pool. The innocuous flow pool was made of 100,459 flows related to HTTP requests towards 898 different web-servers[1]. Among these, 7 flows matched the same regular expression as the polymorphic worm. Thus, in absence of noise in the suspicious flow pool, a generated signature that matched the worm invariants would result in around 0.007% of false positives on the innocuous flow pool. These 7 flows were the only ones to contain the \xFF\xBF string. Very similar to our traffic data, the \xFF\xBF string was present in 0.008% of the evaluation flows used by Polygraph's authors to perform their experiments [76]. In [76] the \xFF\xBF token caused the Bayes signature to produce 0.008% of false positives.

Test flow pools. We used two sets of test flows in our experiments. The first set was made of 217,164 innocuous flows[1] extracted from the traffic traces. We inspected this test set to ensure that it did not include any flow containing the \xFF\xBF string. The second test set was made of 100 simulated worm variants. We used the first test set to measure the false positive rate and the second to measure the false negative rate produced by the signatures. Note that we obtained the innocuous flow pool and the test set made of innocuous flows from two different slices of the network traces.

Score multiplier strings. We used a dataset made of 5,000 flows to extract the score multiplier strings. We analyzed the flows using the algorithm presented in [81]. We extracted all the substrings of length from 6 to 15 bytes having an occurrence frequency between 0.05 and 0.2, obtaining around 300 different strings. Many of them were strings related to HTTP-header fields introduced by certain browsers, such as “Cache-Control’, “Modified-Since”, “Firefox/0.10.1”, “Downloader/6.3”,

[1] The flows were “innocuous” in the sense that they did not contain the considered worm.

etc. The extracted strings are the candidate strings that can be used to obtain a score multiplying effect to force Bayes signatures to generate a high number of false positives, as explained in Section 4.3.3. It is worth noting that the flows used to extract the score multiplier strings contained both inbound and outbound HTTP requests taken from the perimeter of our network. The flows were related to requests among a large number of different web-servers and clients. For these reasons we expect the obtained strings and occurrence frequencies to be general and not specific just to our network[2].

Fake anomalous flows. We crafted the fake anomalous flows using the algorithm presented in Section 4.3.3. We used $k = 2$ fake invariants of length $l = 5$ for all the fake anomalous flows. We used several combinations of score multiplier strings v by splitting them in different ways to obtain a different number of substrings for each test. For each fake anomalous flow, we chose $\frac{2}{3}$ of the obtained substrings at random and injected them into the flow[3].

4.4.2 Misleading Bayes Signatures

In [15] Polygraph's authors state that Bayes signatures are resilient to the presence of noise into the suspicious flow pool until the noise level reaches at least 80% of the total number of flows. In our experiments we found that if the fake anomalous flows are properly crafted, just 50% of noise in the suspicious flow pool (i.e., 1 fake anomalous flow per worm variant) can make the generated signature useless. We performed several experiments using 10 worm variants and 1 or 2 fake anomalous flows per variant in the suspicious flow pool. The fake anomalous flows were crafted as explained in Section 4.3.3 and 4.4.1. We report the results of two group of tests below.

Case 1. We obtained the best result using "Firefox/0.10.1" (12.2%) and "shockwave-flash" (11.9%) as score multiplier strings. The percentages between parenthesis represent the occurrence frequencies of the strings (see Section 4.4.1). We split the two score multiplier strings to obtain all the possible substrings of size $m = 9$ (e.g., "Firefox/0", "irefox/0.", "refox/0.1", etc.).

As described above, we simulated two attack scenarios using 1 and 2 fake anomalous flows per

[2] The extracted strings could obviously present different occurrence frequencies over time. Nevertheless it is reasonable to assume that the attacker could perform a similar analysis on traffic traces collected just a few weeks or even days before launching the attack.

[3] Thus, the fake anomalous flows did not always contain the same set of substrings.

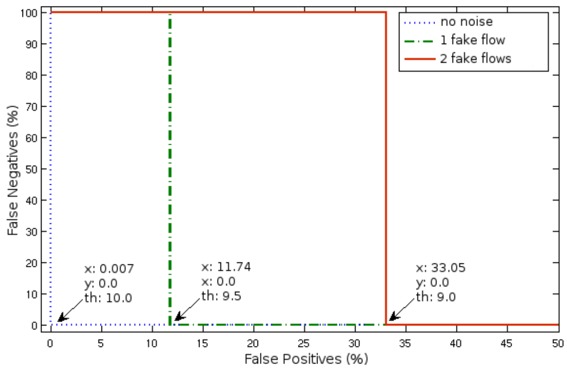

Figure 4.6: Case 1. The false positives are measured over the innocuous flow pool

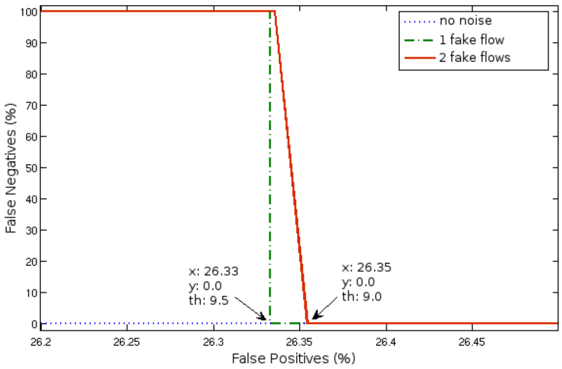

Figure 4.7: Case 1. The false positives are measured over the test flow pool

worm variant, respectively. Therefore, the suspicious flow pool was made of 20 flows during the first attack scenario and of 30 flows during the second one. We generated the Bayes signature on the suspicious flow pool and measured the false positive rates on the innocuous flow pool and the test flow pool made of innocuous traffic. The results are shown in Figure 4.6 and Figure 4.7. Please note that the graphs are represented on different ranges of false positives to highlight the difference between the two attack scenarios. The plots represent the false positives and false negatives produced by the signature while varying the threshold θ starting from 0.0 and incrementing it using a 0.5 increment step. A threshold equal to 0.0 obviously produces 100% of false positives and 0% of false negatives. By incrementing the threshold, the percentage of false positives decreases. The arrows indicate the coordinates related to the maximum value of the threshold that produces no false

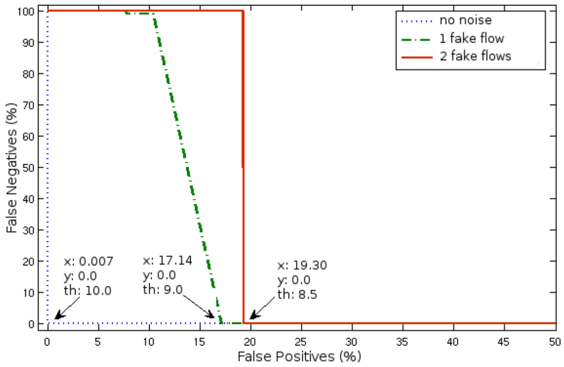

Figure 4.8: Case 2. The false positives are measured over the innocuous flow pool

negatives. The Bayes signature generated during the second scenario is reported in [81].

In Section 4.3.3 we discussed how Polygraph optimizes the threshold θ for Bayes signatures. It is easy to see from Figure 4.6 that the noise injection attack prevents the threshold θ to be optimized. Consider for example the graph related to the injection of 1 fake anomalous flow per worm variant. If $\theta = 9.5$, the signature generates 11.74% of false positives and 0% of false negatives. In order to decrease the number of false positives the threshold would need to be incremented further. However, as soon as the threshold exceeds 9.5 the signature produces 100% of false negatives.

Case 2. In this case "Pragma: no-cache" (9.4%) and "-powerpoint" (7.0%) were used as score multiplier strings. We split these two strings to obtain all the substrings of length $m = 4$. Again, the suspicious flow pool contained 10 worm variants and 1 or 2 fake anomalous flows per variant. The results are reported in Figures 4.8 and 4.9. Please note that, again, the graphs are represented on different ranges of false positives to highlight the difference between the two attack scenarios. The Bayes signature generated during the second scenario (2 fake anomalous flows per worm variant) is reported in [81].

4.4.3 Misleading All The Three Signatures at The Same Time

The objective of the noise injection attack is to prevent the generation of useful signatures. In order to achieve this result the attack needs to prevent the generation of useful conjunction, token-subsequences, and Bayes signatures at the same time. As discussed in Section 4.3.3, the results of

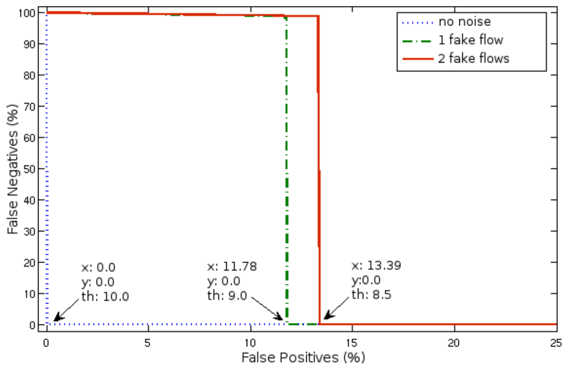

Figure 4.9: Case 2. The false positives are measured over the test flow pool

	1 fake anomalous flow	2 fake anomalous flows
Conjunction	73.3%	88.9%
Token-subsequences	60.0%	73.3%
Bayes	100%	100%
All three signatures	44.4%	**62.2%**

Table 4.1: Percentage of successful attacks (using “Forwarded-For” and “Modified-Since”)

the attack are not deterministically predictable. In order to estimate the probability of success we simulated the noise injection attack multiple times. We considered an attack successful if Polygraph did not generate a conjunction or token-subsequence signature that would match the worm and if the Bayes signature produced more than 1% of false positives measured over the innocuous flow pool. Even though a false positive rate around 1% is seemingly low, we consider it intolerable for a *blocking* signature. We report the results with fake anomalous flows crafted using two different combinations of score multiplier strings. We divided the tests into two groups. The first group of tests were performed using “Forwarded-For” (11.3%) and “Modified-Since” (15.2%) as score multiplier strings, splitting them into substrings of length $m = 5$. The second group of test were performed using “Cache-Control” (15.1%) and “Range: bytes” (11.9%), splitting them in substrings of length $m = 4$. For each group of tests we simulated two noise injection attack scenarios using 1 and 2 fake anomalous flows per worm variant, respectively. We used 5 worm variants in the suspicious flow pool for both the first and the second scenario. We generated the signatures 45 times for the first group of tests and 20 times for the second group. The results are shown in Table 4.1 and Table 4.2. The reported percentages represent how many times the attack was successful in

	1 fake anomalous flow	2 fake anomalous flows
Conjunction	65%	95%
Token-subsequences	40%	90%
Bayes	90%	100%
All three signatures	20%	**85%**

Table 4.2: Percentage of successful attacks (using "Cache-Control" and "Range: bytes")

avoiding the generation of useful signatures. The first three rows report the percentage of success computed for each type of signatures, individually. The last row represents the percentage of attacks that succeeded in misleading Polygraph so that it could not generate any useful signature, regardless of the signature type. It is worth noting that in both experiments, when using 2 fake anomalous flows per worm variant, the attack has a higher probability to succeed, and further, it prevents Polygraph from generating a useful Bayes signature 100% of the time.

4.4.4 Analysis of the Results

Polygraph's authors showed that their system is resilient to the presence of as much as 80% of "normal" noise in the suspicious flow pool [76]. However, we showed that if the noise is properly crafted, just 50% of noise could prevent Polygraph from generating useful signatures a majority of the times. As shown above, if the detection threshold for Bayes signatures is set in order to produce a low amount of false positives, we obtain almost 100% of false negatives. According to the attack taxonomy in [14], we can interpret this as the result of a successful *causative integrity attack*, because the learning phase is influenced so that future attacks will not be detected. On the other hand, if the detection threshold is set so that a low number of false negatives are produced, the signatures generate too many false positives. If the signatures were deployed they would produce a *self-Denial of Service* attack. We can interpret this as the result of a successful *causative availability attack*.

In addition, as explained in Section 4.3.3, the noise injection attack can be easily combined with the red herring attack discussed in [76]. The combination of the two attacks increases the probability that the worm will prevent the generation of a useful signature.

We also conducted preliminary experiments on NETBIOS traffic to extract score multiplier strings that can be used by a worm that uses this protocol as attack vector. We chose NETBIOS because it is an attack vector for most of the *OS-based* worms. We analyzed more than 5,000 NETBIOS flows, searching for strings of length from 6 to 15 bytes and an occurrence frequency between 0.05 and 0.2. We found 29 candidate strings in “TCP-based” NETBIOS traffic and 58 candidate strings in “UDP-based” requests. This experiment suggests that our noise injection technique using “score multiplier” strings can work for a variety of protocols.

4.5 Attack Against Semantic-based Signature Generators

In [78] Newsome et al. propose dynamic taint analysis for automatic signature generation. The idea consists in running (potentially) vulnerable network applications in a virtual machine. This gives full control on the instructions executed by the application. The method aims to detect the memory location to which the execution of the application is hijacked while under attack. Assume, for example, an application running in the virtual machine for taint analysis has a buffer overflow vulnerability [79]. Assume also a new worm has been developed in order to exploit this vulnerability. While running in the virtual machine, all the data arriving to the vulnerable application from the network are labeled as tainted, and every attempt to hijack the application’s execution flow to execute code contained in tainted data is detected. As soon as the worm flow tries to force the application to execute the worm’s code, the application is stopped and the taint analysis engine registers the address where the worm code resides in memory. As there are just a limited number of possible address locations the worm could use, part of the registered address (e.g., the first two bytes) is likely shared by all the (polymorphic) variants of the worm and may be used to help Polygraph in generating a more robust worm signature [78]. According to the description of fake anomalous flows given above, the noise injection attack we presented would not work in this case, because the fake anomalous flows do not attempt to exploit the vulnerability and are then filtered out by the taint analysis engine. On the other hand, we can imagine of a noise injection attack for which the fake anomalous flows are constructed to actually exploit the vulnerability and hijack the application’s execution flow to a random memory address. In this case it is difficult to distinguish between a real worm and a fake anomalous flow. This means that the fake anomalous flows cannot be easily filtered out, unless further and more expensive semantic analysis is performed. The only shortcoming of this attack is represented by the fact that the worm propagation would slow down. Given that each new worm instance and its fake anomalous flows are sent to the next victim in a random order, it might happen that the first flow to be received by the application is a fake anomalous flow. In this case, because the fake anomalous flow actually exploits the vulnerability, the attacked application may crush due to the attempt to hijack the execution to a random memory address. This prevents the worm flow, which arrives later, to be executed and infect the machine. On the other hand, whenever the worm flow is the first to be received, the machine is infected and contributes to the worm propagation.

4.6 Possible Countermeasures

A possible defense against our implementation of the noise injection attack is to use a white list to attempt to filter out flows that contain the score multiplier substrings. However, this is not straightforward and may not even be possible. As shown in Section 4.4.1, there are a very large number of strings that a worm can potentially use. The set of candidate strings extracted from the traffic are determined by the occurrence frequency ranges, and the sets of substrings are determined by the string length value. These are chosen by the attacker and are not known *a priori* to the signature generator. Further, the strings actually used by a worm instance to create fake anomalous flows can change over time. As a result, a reliable way to filter out the fake anomalous flows is to look for occurrences of all possible substrings of a very large set of strings. This can be very expensive. Further, such aggressive filtering may prevent the system from producing useful worm signatures that happen to contain such substrings.

Another possible countermeasure against the score multiplier strings technique is to modify the detection algorithm for Bayes signatures. For example, every time a test flow matches a token, the related bytes in the flow should be marked to prevent them from “participating” in matching another token of the same signature. This means that the score multiplier effect described in Section 4.3.3 cannot be achieved anymore. However, the attack may still work if multiple candidate strings *v* (see Section 4.3.3) are carefully chosen and if they are split without overlap, although now the induced false positive rate may be much less than the one obtained during the experiments reported in Section 4.4.2.

Even if the above countermeasures happen to work in some cases, the fundamental problem still exists: without an accurate and robust flow classifier that can prevent the injection of fake anomalous flows, syntactic-based automated signature generators are vulnerable. The noise injection attack we have described above is proof-of-concept. We suspect there are many other similar attacks which may also defeat semantic-based signature generators, as described in Section 4.5, and believe that a robust solution to the noise injection attack is still an open research problem.

Chapter 5

Operating in Adversarial Environment

In the previous chapter we discussed how an adversary may try to interfer with the learning process used by IDS. In this chapter we describe another challenge. Assume no interference was present during the training of the IDS. After deployment, the adversary may still try to launch sophisticated attacks which are crafted in order to "evade" the IDS, so that no alarm is raised. In the following we focus on *evasive attacks* against anomaly-based IDS. Evasive attacks of this type are usually referred to as mimicry attacks. We present a recently proposed mimicry attack against payload-based anomaly IDS, first, and then we present a possible solution to make payload-based anomaly IDS more robust by means of a Multiple Classifier System (MCS).

5.1 Payload-based Anomaly Detection

Recent work on unlabeled anomaly detection focused on *high speed* classification based on simple payload[1] statistics [55, 66, 112, 113, 111]. For example, PAYL [112, 113] extracts 256 features from the payload. Each feature represents the occurrence frequency in the payload of one of the 256 possible byte values. A simple model of normal traffic is then constructed by computing the average and standard deviation of each feature. A payload is considered anomalous if a *simplified Mahalanobis distance* between the payload under test and the model of normal traffic exceeds a

[1] The payload is the data portion of a network packet.

predetermined threshold. Wang et al. [112] also proposed a more generic n-gram[2] version of PAYL. In this case the payload is described by a pattern vector in a 256^n-dimensional feature space. The n-grams extract byte sequence information from the payload, which helps in constructing a more precise model of the normal traffic compared to the simple byte frequency-based model. The extraction of n-gram statistics from the payload can be performed efficiently and the IDS can be used to monitor high speed links in real time. However, given the exponentially growing number of extracted features, the higher n the more difficult it may be to construct an accurate model because of the curse of dimensionality and possible computational complexity problems. In order to overcome the high dimensionality problem, Wang et al. recently proposed ANAGRAM [111], an anomaly IDS that uses Bloom filters to “compress” the dimensionality of the feature space. First they propose a single Bloom filter to model only (unlabeld) normal traffic, and then they propose a second filter which models known attacks. During detection, the n-grams are extracted from the payload and matched against both the normal and attack models. The authors also discuss the ability of ANAGRAM to detect polymorphic blending attacks (which we discuss in Section 5.2.3) constructed to evade 1-gram PAYL.

Other anomaly detection systems based on more complex features have been proposed [103, 21]. These anomaly detectors involve the extraction of syntax and semantic information from the payload, which is usually a computationally expensive task. Therefore, it may not be possible to use this approaches in order to analyze network traffic on high speed links in real time.

5.2 Evading Detection

Since IDS started to become popular, researchers began studying the robustness of IDS against sophisticated attacks which are constructed with the objective of exploiting the targeted vulnerability without being detected. This type of attacks are usually referred to as *evasive attacks*. Researchers in this area have used TCP/IP transformations to demonstrate IDS evasions [85], and address weaknesses created by ambiguities in network protocols [39]. Numerous tools have been created for evading IDS, including `fragroute` [96], `snot` [95], and `mucus` [75]. Some authors have inves-

[2] Here an n-gram represents n consecutive bytes in the payload

tigated techniques to automate the generation of evasive attacks. For example, in [108], the authors identified mutation operations to generate variations on known exploits. Similarly, the authors in [91] modeled attack transformations to derive new variations on known attacks.

5.2.1 Polymorphic Attacks

In Section 4.3.3 we presented a high level structure of a polymorphic worm. Polymorphism can be applied to generic attacks using the same high level structure. As a consequence, a polymorphic attack is an attack that is able to change its appearance with every instance. Therefore, there may be no fixed or predictable signature for the attack which could be used by signature-based IDS. As a result, polymorphic attacks have a high chance of evading detection because most of the current intrusion detection systems and anti-virus systems are signature-based.

5.2.2 Mimicry Attacks

It has been demonstrated that many anomaly detection systems can be evaded by *mimicry* attacks [109, 54, 26, 34]. A mimicry attack is an evasive attack against a network or system vulnerability. The attack is carefully crafted so that the attack pattern, i.e., the representation of the attack used during the classification process, lies inside the decision surface that separates the normal patterns from the anomalous ones (i.e., the *outliers*). A successful mimicry attack is able to exploit the targeted vulnerability while causing the anomaly IDS to produce a false negative (i.e., no alarm is raised). Mimicry attacks by means of *evasive polymorphism* have been recently explored [26, 34]. These attacks aim to evade payload-based anomaly detectors. CLET [26], an advanced polymorphic engine, performs spectrum analysis on the payload in order to evade IDS. Given an attack payload, CLET adds padding bytes in a separate *cramming bytes* zone (of given length) to make the byte frequency distribution of the attack close to the model of normal traffic. In [34], Fogla et al. showed how to construct a mimicry attack, called *polymorphic blending attack*, that can evade 1-gram (i.e., the *single-byte frequency* version) and 2-gram PAYL. Using byte substitution and padding techniques, the polymorphic blending attack encodes the attack payload so that the obtained *transformed* attack is classified as normal by PAYL, while still being able to exploit the targeted vulnerability. We discuss the details of the polymorphic blending attack in Section 5.2.3.

5.2.3 Polymorphic Blending Attack

Polymorphic attack instances usually look very different from normal traffic. For example, the polymorphic decryptor and encrypted shellcode (Section 4.3.3) may contain characters that have very low probability of appearing in normal packets. Thus, an anomaly-based IDS may detect the polymorphic attack instances by recognizing their deviation from the normal profile. For example, Wang et al. [112, 113] showed that the byte frequency distribution of an (polymorphic) attack is quite different from that of normal traffic, and can thus be used by PAYL to detect polymorphic attacks.

Clearly, if a polymorphic attack can “blend in” with (or look like) normal, it can evade detection by an anomaly-based IDS. Normal traffic contains a lot of syntactic and semantic information, but only a very small amount of such information can be used by *high speed* network-based anomaly IDS. This is due to fundamental difficulties in modeling complex systems and performance overhead related to real-time monitoring. For example, the network traffic profile used by PAYL [112, 113] includes simple statistics such as maximum or average size and rate of packets, frequency distribution of bytes in packets, and range of tokens at different offsets. The simplicity of PAYL makes it fast and suitable for real-time detection in high speed links. However, very low structural information is extracted from the payload and used to construct the model of normal traffic.

Given the incompleteness and imprecision of the normal profiles based on simple traffic statistics, it is quite feasible to launch what we call *polymorphic blending attacks*. The main idea is that, when generating a polymorphic attack instance, care can be taken so that its payload characteristics, as measured by the anomaly IDS, will match the normal profile. For example, in order to evade detection by PAYL [112, 113], the polymorphic engine can carefully choose the characters used in encryption and pad the attack payload with a chosen set of characters, so that the resulting byte frequency of the attack instance will closely match the normal profile and thus be considered as normal by PAYL.

From the point of view of statistical pattern recognition, the polymorphic blending attack can be seen as a transformation **T** which modifies an attack in order to move its representation (i.e., its pattern vector) inside the decision surface constructed by the IDS, as depicted for example in Figure 5.1.

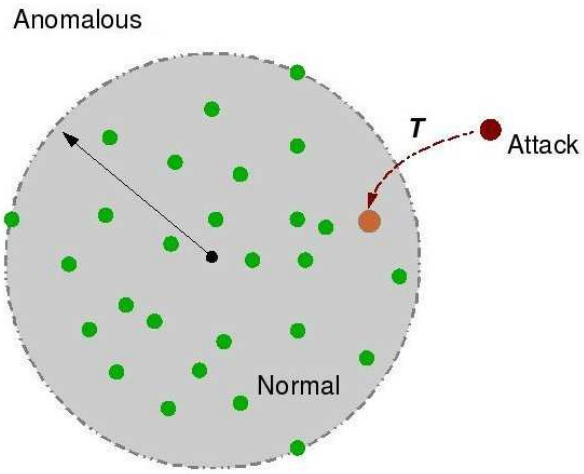

Figure 5.1: Polymorphic Blending Attack. After transformation the attack lies inside the decision surface constructed around normal traffic.

Attack Scenario

Figure 5.2 shows a possible scenario for the polymorphic blending attack. There are a few assumptions behind this scenario:

- The attack program has already compromised a host X inside a network A which communicates with the target host Y inside network B. Network A and host X may have poor security so that the attack can penetrate without getting detected, or there is a colluding insider.

- The attack program has knowledge of IDS_B. This might be possible using a variety of approaches, e.g., social engineering (e.g., company sales or purchase data), or fingerprinting, or trial-and-error. We argue that one cannot assume that the IDS deployment is a secret and security by obscurity is never reliable. We assume IDS_B is a payload-based anomaly detection system (e.g., PAYL [112]).

- Given some packet data from X to Y, the attack program will be able to generate its own version of the statistical normal profile used by IDS_B. This is feasible if we assume that the IDS_B is known and hence its algorithm for learning a normal profile is also known.

- A typical anomaly IDS has a threshold setting that can be adjusted to obtain a desired false

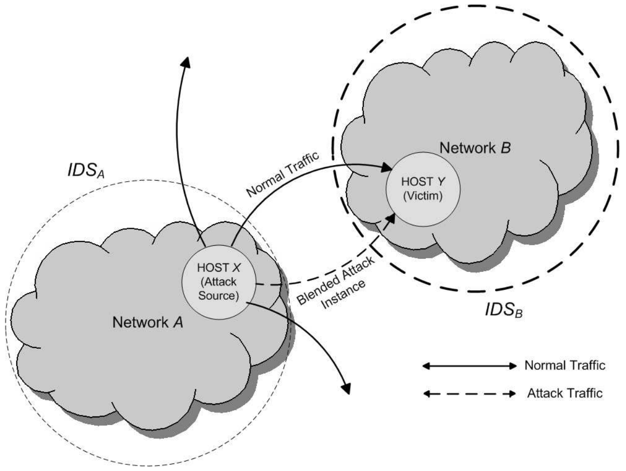

Figure 5.2: Polymorphic Blending Attack scenario [34]

positive rate. We assume that the attack program does not know the exact value of the threshold used by IDS_B, but has an estimation of the generally acceptable false positive and false negative rates. With this knowledge, the attack program can estimate the error threshold when crafting a new attack instance to match the IDS profile.

Once the attack program has control of host X, it observes the normal traffic going from X to Y. The attacker builds (estimates) a normal profile for this traffic using the same modeling technique that IDS_B uses. This profile is called *artificial profile* [34]. With it, the attack program creates a mutated instance of itself in such a way that the statistics of the mutated instance matches the artificial profile. When IDS_B analyzes these mutated attack packets, it is unable to discern them from normal traffic because the artificial profile can be very close to the actual profile in use by IDS_B. Thus, the attack successfully infiltrates the network B and compromises host Y.

The polymorphic blending attack has three basic steps: (1) learn the IDS normal profile; (2) encrypt the attack body; (3) and generate a polymorphic decryptor.

Learning the Normal Profile

The task at hand for the attack program is to observe the normal traffic going from a host, say X, to another host in the target network, say Y, and generate a normal profile close to the one used by the IDS at the target network, say IDS_B, using the same algorithm used by the IDS.

A simple method to get the normal data is by sniffing the network traffic going from network A to host Y. This can be easily accomplished in a bus network. In a switched environment it may be harder to obtain such data. But the attack program knows the type of service running at the target host. It may then simply generate normal request packets and learn the artificial profile using these packets.

In theory, even if the attack program learns a profile from just a single normal packet, and then mutates an attack instance so that it matches the statistics of the normal packet perfectly, the resulting polymorphic blended attack packet should not be flagged as an anomaly by IDS_B if the normal packet does not result in a false positive in the first place. On the other hand, it is beneficial to generate an artificial profile that is as close to the normal profile used by IDS_B as possible so that if a polymorphic blended attack packet matches the artificial profile closely, it has a high chance of evading IDS_B. In general, if more normal packets are captured and used by the attack program, it will be able to learn an artificial normal profile that is closer to the normal profile used by IDS_B.

Attack Body Encryption

After learning the normal profile, the attack program creates a new attack instance and encrypts (and blends) it to match the normal profile. For simplicity, a straightforward byte substitution scheme followed by padding can be used for encryption. The main idea here is that every character in the attack body can be substituted by a character(s) observed from the normal traffic using a substitution table. The encrypted attack body can then be padded with some more garbage normal data so that this polymorphic blended attack packet can match the normal profile even better. To keep the padding (and hence the packet size) minimal, the substituted attack body should already match the normal profile closely. We can use this design criterion to produce a suitable substitution table.

To ensure that substitution algorithm is reversible (for decrypting and running the attack code), a one-to-one or one-to-many mapping can be used. A single-byte substitution is preferred over multi-byte substitution because multi-byte substitution will inflate the size of attack body after substitution. An obvious requirement of such encryption scheme is that the encrypted attack body should contain characters from only the normal traffic. Although this may be hard for a general encryption technique (because the output typically looks random), it is an easy requirement for a simple byte substitution scheme. However, finding an optimal substitution table that requires minimal padding is a complex problem. In [34], the authors show that for certain cases this is a very hard problem. Therefore, a greedy method is proposed to find an acceptable substitution table. The main idea is to first sort the statistical features in the descending order of the frequency for both the attack body and normal traffic. For each unassigned entry with the highest frequency in the attack body, map it to an available (not yet mapped) normal entry with the highest frequency. Repeat this until all entries in the attack body are mapped. The feature mapping can be translated to a character mapping. Then a substitution table can be created for encryption and decryption purposes. For the details of the greedy algorithm see [34].

Polymorphic Decryptor

Once the vulnerability has been exploited, the decryptor first removes all the extra padding from the encrypted attack body and then uses a reverse substitution table (or decoding table) to decrypt the attack body to produce the original attack code (shellcode).

The decryptor is not encrypted but can be mutated using multiple iterations of shellcode polymorphism processing (e.g., mapping an instruction to an equivalent one randomly chosen from a set of candidates). To reverse the substitution done during blending, the decryptor needs to look up a decoding table that contains the required reverse mappings. The decoding table for one-to-one mapping can be stored in an array where the *i*-th entry of the array represents the normal character used to substitute attack character *i*. Such an encoding table contains only normal characters. Unused entries in the table can be used for padding. On the other hand, storage of decoding tables for one-to-many mapping or variable-length mapping is complicated and typically requires larger space [34].

Incorporating Attack Vector and Polymorphic Decryptor in Blending

The attack vector, decryptor and decryption table are not encrypted. Their addition to the attack packet payload alters the packet statistics. The new statistics may deviate significantly from the normal profile. If the changes are significant, the normal profile has to be adjusted through an iterative blending process [34].

5.3 Hardening Payload-based Anomaly Detection Systems

In order to make it harder for the attacker to evade the IDS, a comprehensive model of the normal traffic is needed. Furthermore, the modeling technique needs to be also practical and efficient. We address these challenges using an ensemble of classifiers. Classifier ensembles, often referred to as Multiple Classifier Systems (MCS), have been proved to achieve better accuracy in many applications, compared to the best single classifier in the ensemble. A number of security related applications of MCS have been proposed in the literature. For example, MCS are used in multimodal biometrics for hardening person identification [19, 41], and in misuse-based IDS [37, 36] to improve the detection accuracy. To the best of our knowledge, no work has been presented so far that explicitly addresses the problem of increasing the *hardness of evasion* of anomaly-based IDS using multiple classifier systems. We propose a new approach to construct a *high speed* payload-based anomaly IDS by combining multiple one-class SVM classifiers. Our approach is intended to improve both the detection accuracy and the hardness of evasion of high speed anomaly detectors. MCS attain accuracy improvements when the combined classifiers are "diverse", i.e., they make different errors on new patterns [28]. A way to induce diversity is to combine classifiers that are based on descriptions of the patterns in different feature spaces [57]. We propose a new technique to extract the features from the payload that is similar to the 2-gram technique. Instead of measuring the frequency of the pairs of consecutive bytes, we propose to measure the features by using a sliding window that "covers" two bytes which are ν positions apart from each other in the payload. We refere to this pairs of bytes as 2_ν-grams. The proposed featrue extraction process do not add any complexity with respect to the traditional 2-gram technique and can be performed efficiently. We also show that the proposed technique allows us to "summarize" the occurrence frequency of n-grams, with $n > 2$, thus capturing byte sequence information while limiting the dimensionality of the feature space. By varying the parameter ν, we construct a representation of the payload in different feature spaces. Then we use a feature clustering algorithm originally proposed in [27] for text classification problems to reduce the dimensionality of the different feature spaces where the payload is represented. Detection accuracy and hardness of evasion are obtained by constructing our anomaly-based IDS using a combination of multiple one-class SVM classifiers that work on these different feature spaces. Using multiple classifiers forces the attacker to devise a mimicry attack that evades multiple models of normal traffic at the same time, which is intuitively harder than evading just one model. We compare our payload-based anomaly IDS to the original implementation of 1-gram PAYL by Columbia University, to an implementation of 2-gram PAYL, and to an IDS constructed by combining multiple one-class classifiers based on the *simplified Mahalanobis distance* used by PAYL.

In the following, we present two different one-class classification algorithms that we used to perform our experiments, namely a classifier inspired to the Support Vector Machine (SVM) [107], and a classifier based on the Mahalanobis distance [29]. As we discuss in Section 5.3.3, there is an analogy between anomaly detection based on *n*-gram statistics and text classification problems. We chose the one-class SVM classifier because SVM have been shown to achieve good performances in text classification problems [93, 62]. We also describe the Mahalanobis distance based classification algorithm because it is the same classification algorithm used by PAYL [112], a recently proposed anomaly detector based on *n*-gram statistics.

5.3.1 One-Class SVM

We use the classifier described in Section 3.4.5, which was proposed by Schölkopf et al. in [92]. Because we combine multiple classifiers using the simple majority voting rule, as described in Section 5.3.3, here we do not use Equation 3.15 to transform the output of the classifier into class conditional probabilities estimates. Therefore, we simply use the Gaussian kernel

$$K(\mathbf{x}, \mathbf{y}) = \Phi(\mathbf{x}) \cdot \Phi(\mathbf{y}) = exp\left(-\gamma\|\mathbf{x} - \mathbf{y}\|^2\right) \tag{5.1}$$

and compute the output of the classifier according to Equation 3.14.

5.3.2 Mahalanobis Distance-based Classifier

Given a training dataset $D = \{\mathbf{x}_1, \mathbf{x}_2, .., \mathbf{x}_m\}$, the average ϕ_i and standard deviation σ_i are computed for each feature as

$$\begin{aligned} \phi_i &= \tfrac{1}{m} \textstyle\sum_{k=1}^{m} x_{k_i} \\ \sigma_i &= \sqrt{\tfrac{1}{m-1} \textstyle\sum_{k=1}^{m} (x_{k_i} - \phi_i)^2} \end{aligned} \tag{5.2}$$

where x_{k_i} is the i-th feature of a pattern $\mathbf{x}_k \in D$ and m is the total number of training patterns. We call $M(\phi, \sigma)$ the model of normal traffic, where $\phi = [\phi_1, \phi_2, .., \phi_l]$ and $\sigma = [\sigma_1, \sigma_2, .., \sigma_l]$. Assuming the features to be uncorrelated, a *simplified Mahalanobis distance*[3] [112] $\Delta(\mathbf{z}, M(\phi, \sigma))$ between a generic pattern $\mathbf{z} = [z_1, z_2, .., z_l]$ and the model $M(\phi, \sigma)$ can be computed as

$$\Delta(\mathbf{z}, M(\phi, \sigma)) = \sum_{i=1}^{l} \frac{|z_i - \phi_i|}{\sigma_i + \alpha} \tag{5.3}$$

where α is a constant *smoothing factor* introduced in order to avoid division by zero. Given a threshold θ, the decision rule for the classifier can be written as

$$\Delta(\mathbf{z}, M(\phi, \sigma)) > \theta \;\Rightarrow\; \mathbf{z} \text{ is an outlier} \tag{5.4}$$

The threshold θ can be computed during training so that a chosen rejection rate r of patterns in D is left outside the decision surface, i.e., the classifier produces a false positive rate r on the training dataset D, if we assume D contains only examples extracted from the target class.

5.3.3 Payload Classification

Feature Extraction

The detection model used by PAYL [112] is based on the frequency distribution of the n-grams (i.e., the sequences of n consecutive bytes) in the payload. The occurrence frequency of the n-grams is

[3] The *simplified Mahalanobis distance* do not involve square operations, which would slow down the computation of the distance.

measured by using a sliding window of length n. The window slides over the payload with a step equal to one byte and counts the occurrence frequency in the payload of the 256^n possible n-grams. Therefore, in this case the payload is represented by a pattern vector in a 256^n-dimensional feature space. It is easy to see that the higher n, the larger the amount of structural infomation extracted from the payload. However, using $n = 2$ we already obtain 65,536 features. Larger values of n are impractical given the exponentially growing dimensionality of the feature space and the curse of dimensionality problem [29]. On the other hand, by measuring the occurrence frequency of pairs of bytes that are ν positions (i.e., ν bytes) apart from each other in the payload, it is still possible to extract some information related to the n-grams, with $n > 2$. We call such pairs of bytes 2_ν*-grams*. In practice, the occurrence frequency of the 2_ν-grams can be measured by using a $(\nu+2)$ long sliding window with a “gap” between the first and last byte.

Consider a payload $B = [b_1, b_2, .., b_l]$, where b_i is the byte value at position i. The occurrence frequency in the payload B of an n-gram $\beta = [\beta_1, \beta_2, .., \beta_n]$, with $n < l$, is computed as

$$f(\beta|B) = \frac{\text{\# of occurrences of } \beta \text{ in } B}{l - n + 1} \tag{5.5}$$

where the number of occurrences of β in B is measured by using the sliding window technique, and $(l - n + 1)$ is the total number of times the window can “slide” over B. $f(\beta|B)$ can be interpreted as an estimate of the probability $p(\beta|B)$ of finding the n-gram β (i.e., the sequence of consecutive bytes $[\beta_1, \beta_2, .., \beta_n]$) in B. Accordingly, the probability of finding a 2_ν-gram $\{\beta_1, \beta_{\nu+2}\}$ can be written as

$$p(\{\beta_1, \beta_{\nu+2}\}|B) = \sum_{\beta_2,..,\beta_{\nu+1}} p([\beta_1, \beta_2, .., \beta_{\nu+1}, \beta_{\nu+2}]|B) \tag{5.6}$$

It is worth noting that for $\nu = 0$ the 2_ν-gram technique reduces to the “standard” 2-gram technique. When $\nu > 0$, the occurrence frequency in the payload of a 2_ν-gram $\{\beta_1, \beta_{\nu+2}\}$ can be viewed as a marginal probability computed on the distribution of the $(\nu + 2)$-grams that start with β_1 and end with $\beta_{\nu+2}$. In practice the frequency of a 2_ν-gram somehow “summarizes” the occurrence frequency of 256^ν n-grams, with $n = \nu + 2$.

From the occurrence frequency of the n-grams it is possible to derive the distribution of the $(n - 1)$-grams, $(n - 2)$-grams, etc. On the other hand, measuring the occurrence frequency of the

2_ν-grams does not allow us to automatically derive the distribution of $2_{(\nu-1)}$-grams, $2_{(\nu-2)}$-grams, etc. The distributions of 2_ν-grams with different values of ν give us different structural information about the payload. The intuition is that, ideally, if we could somehow combine the structural information extracted using different values of $\nu = 0, .., N$ we would be able to reconstruct the structural information given by the distribution of n-grams, with $n = (N + 2)$. This motivates the combination of classifiers that work on different descriptions of the payload obtained using the 2_ν-gram technique with different values of ν.

Feature Reduction

Payload anomaly detection based on the frequency of n-grams is analogous to a text classification problem for which the bag-of-words model and a simple unweighted raw frequency vector representation [62] is used. The different possible n-grams can be viewed as the words, whereas a payload can be viewed as a document to be classified. In general for text classification only the words that are present in the documents of the training set are considered. This approach is not suitable in case of a one-class classification problem. Given that the training set contains (almost) only target examples (i.e., “normal” documents), we cannot conclude that a word that have a probability equal to zero to appear in the training dataset will not be discriminant. As a matter of fact, if we knew of a word w that has probability $p(w|d_t) = 0,\ \forall d_t \in C_t$, of appearing in the class of target documents C_t, and $p(w|d_o) = 1,\ \forall d_o \in C_o$, of appearing in documents of the outlier class C_o, it would be sufficient to measure just one binary feature, namely the presence or not of w_t in the document, to construct a perfect classifier. This is the reason why we choose to take into account all the 256^n n-grams, even though their occurrence frequency measured on the training set is equal to zero. Using the 2_ν-gram technique we still extract 256^2 features. This high number of features could make it difficult to construct an accurate classifier, because of the curse of dimensionality [29] and possible computational complexity problems related to learning algorithms.

In order to reduce the dimensionality of the feature space for payload anomaly detection, we apply a feature clustering algorithm originally proposed by Dhillon et al. in [27] for text classification. Given the number of desired clusters, the algorithm iteratively aggregates the features until the information loss due to the clustering process is less than a certain threshold. This clustering algorithm has the property to reduce the within cluster and among clusters Jensen-Shannon divergence [27] computed on the distribution of words, and has been shown to help obtain better classification accuracy results with respect to other feature reduction techniques for text classification [27]. The inputs to the algorithm are:

1. The set of distributions $\{p(C_i|w_j) : 1 \le i \le m,\ 1 \le j \le l\}$, where C_i is the i-th class of documents, m is the total number of classes, w_j is a word and l is the total number of possible different words in the documents.

2. The set of all the priors $\{p(w_j),\ 1 \le j \le l\}$.

3. The number of desired clusters k.

The output is represented by the set of word clusters $W = \{W_1, W_2, .., W_k\}$. Therefore, after clustering the dimensionality of the feature space is reduced from l to k. The information loss is measured as

$$Q(\{W_h\}_{h=1}^{k}) = \sum_{h=1}^{k} \sum_{w_j \in W_h} p(w_j) KL(p(C|w_j), p(C|W_h)) \tag{5.7}$$

where $C = \{C_i\}_{i=1..m}$, and $KL(p_1, p_2)$ is the Kullback-Leibler divergence between the probability distributions p_1 and p_2.

In the original l-dimensional feature space, the j-th feature of a pattern vector $\mathbf{x}_i$ represents the occurrence frequency $f(w_j|d_i)$ of the word w_j in the document d_i. The new representation $\mathbf{x}'_i$ of d_i in the k-dimensional feature space can be obtained by computing the features according to

$$f(W_h|d_i) = \sum_{w_j \in W_h} f(w_j|d_i), \quad h = 1, .., k \tag{5.8}$$

where $f(W_h|d_i)$ can be interpreted as the occurrence frequency of the cluster of words W_h in the document d_i.

In case of a one-class problem, $m = 2$ and we can call C_t the target class and C_o the outlier class.

The posterior probabilities $\{p(C_i|w_j) : i = t, o,\ 1 \leq j \leq l\}$ can be computed as

$$p(C_i|w_j) = \frac{p(w_j|C_i)p(C_i)}{p(w_j|C_t)p(C_t)+p(w_j|C_o)p(C_o)} \qquad i = t, o, \quad 1 \leq j \leq l \tag{5.9}$$

and the priors $\{p(w_j),\ 1 \leq j \leq l\}$ can be computed as

$$p(w_j) = p(w_j|C_t)p(C_t) + p(w_j|C_o)p(C_o),\ \ 1 \leq j \leq l \tag{5.10}$$

The probabilities $p(w_j|C_t)$ of finding a word w_j in documents of the target class C_t can be reliably estimated on the training dataset, whereas it is difficult to estimate $p(w_j|C_o)$, given the low number (or the absence) of examples of documents in the outlier class C_o. Similarly, it is difficult to reliably estimate the prior probabilities $p(C_i) = \frac{N_i}{N}$, $i = t, o$, where N_i is the number of training patterns of the class C_i and $N = N_t + N_o$ is the total number of training patterns. Given that $N_o \ll N_t$ (or even $N_o = 0$), the estimated priors are $p(C_o) \simeq 0$ and $p(C_t) \simeq 1$, which may be very different from the real prior probabilities.

In our application, the words w_j are represented by the 256^2 possible different 2_ν-grams (with a fixed ν). In order to apply the feature clustering algorithm, we estimate $p(w_j|C_t)$ by measuring the occurrence frequency of the 2_ν-grams w_j on the training dataset and we assume a uniform distribution $p(w_j|C_o) = \frac{1}{l}$ of the 2_ν-grams for the outlier class. We also assume $p(C_o)$ to be equal to the desired rejection rate for the one-class classifiers, and accordingly $p(C_t) = 1 - p(C_o)$.

Combining One-Class Classifiers

Multiple Classifier Systems (MCS) have been proved to improve classification performaces in many applications [28]. MCS achieve better performance than the best single classifier when the classifiers of the ensemble are accurate and diverse, i.e., make different errors on new patterns [28]. Diversity can be intuitively induced for example by combining classifiers that are based on descriptions of the patterns in different feature spaces [57].

Several techniques have been proposed in the literature for combining classifiers [57]. To the best of our knowledge, the problem of combining one-class classifiers has been addressed only by Tax et al. in [101] and in [100]. We use a simple majority voting rule [57] to combine one-class classifiers that work on different descriptions of the payload. Suppose we have a dataset of payloads $T = \{\pi_1, \pi_2, .., \pi_m\}$. Given a payload π_k, we extract the features as discussed in Section 5.3.3 obtaining L different descriptions $\{\mathbf{x}_k^{(1)}, \mathbf{x}_k^{(2)}, .., \mathbf{x}_k^{(L)}\}$ of π_k. L one-class classifier are constructed. The h-th classifier is trained on a dataset $D^{(h)} = \{\mathbf{x}_1^{(h)}, \mathbf{x}_2^{(h)}, .., \mathbf{x}_m^{(h)}\}$, obtained from T using the h-th description for the payloads. During the operational phase, a payload is classified as target (i.e., normal) if it is labeled as target by the majority of the classifiers, otherwise it is classified as outlier (i.e., anomalous).

5.4 Experiments

In this section we compare and discuss the classification performance of four different anomaly IDS. We compare the performace obtained using the original implementation of 1-gram PAYL [112] developed at Columbia University, an implementation of 2-gram PAYL, and two anomaly IDS we built by combining multiple one-class classifiers. One of these two IDS was implemented using an ensemble of one-class SVM classifiers, whereas the other was implemented using an ensemble of Mahalanobis Distance-based (MD) one-class classifiers. We also show and discuss the performance of the single classifiers used to construct the ensembles. To the best of our knowledge, no public implementation of 2-gram PAYL exists. We implemented our own (simplified) version of 2-gram PAYL in order to compare its performance to the other considered anomaly IDS.

5.4.1 Experimental Setup

It is easy to see that the accuracy of the anomaly detection systems we consider can be considerably influenced by the values assigned to a number of free parameters. Tuning all the free parameters in order to find the optimal configuration is a difficult and computationally expensive search task. We did not perform a complete tuning of the parameters, but we used a number of reasonable values that should represent an acceptable suboptimal configuration. For 1-gram PAYL we used the default configuration provided along with the software. For all the MD classifiers and our 2-

gram PAYL we set the smoothing factor $\alpha = 0.001$, because this is the same default value for α used by 1-gram PAYL (which also uses the MD classification algorithm). We used `LibSVM` [20] to perform the experiments with one-class SVM. For all the one-class SVM classifiers we used the gaussian kernel in (5.1). Some techniques for the optimization of the parameter γ have been proposed in the literature [13]. Simple tuning is usually performed iteratively changing the value of γ and retraining the classifier [20], which results in a computationally expensive process in case of multiple classifiers. Therefore, in order to choose a suitable value for γ we performed a number of pilot experiments. We noted that setting $\gamma = 0.5$ the one-class SVM classifiers performed well in all the different feature spaces obtained by varying the parameters ν and the number of feature clusters k during the feature extraction and reduction processes, respectively (see Section 5.3.3). Having fixed the values for some of the parameters as explained above, we performed several experiments varying the "gap" ν and the number of feature clusters k. The values we used for this parameters and the obtained results are discussed in detail in Section 5.4.2.

We performed all the experiments using 5 days of HTTP requests towards our department's web server collected during October 2004. We assumed this unlabeled traffic to contain mainly normal requests and possibly a low fraction of noise, i.e., anomalous packets. We used the first day of this traffic to train the IDS and the last 4 days to measure the false positive rate (i.e., the false alarm rate). In the following we refer to the first day of traffic as *training dataset*, and to the last 4 days as *test dataset*. The training dataset contained 384,389 packets, whereas the test dataset contained 1,315,433 packets. In order to estimate the detection rate we used 18 HTTP-based buffer overflow attacks. We collected the first 10 attacks from the Internet (e.g., exploits for `IIS 5.0 .printer ISAPI Extension` [8], `ActivePerl perlIIS.dll` [7], `UnixWare's Netscape FastTrack 2.01a` [9], and also [6, 2, 4, 3, 1, 5, 73]). Each of these attacks is made up of a different number of attack packets. The latter 8 attacks were represented by some of the attacks used in [34], where Fogla et al. constructed a number of mimicry attacks against PAYL. These attacks were derived from an exploit that targets a vulnerability in `Windows Media Services (MS03-022)` [33]. In particular, we used the original `Windows Media Services` exploit used in [34] before transformation, 6 mimicry attacks derived from this original attack using a polymorphic shellcode engine called CLET [26], and one polymorphic blending attack obtained using the *single byte encoding* scheme for the 2-grams presented in [34]. The 6 mimicry attacks obtained using CLET were created setting different combinations of packet length and total number of attack packets. The polymorphic blending attack consisted of 3 attack packets and the payload of each packet was 1460 bytes long. In the following we will refer to the set of attacks described above as *attack dataset*. Overall, the attack dataset contained 126 attack packets.

5.4.2 Performance Evaluation

In order to compare the performace of PAYL, the constructed single classifiers, and the overall anomaly IDS, we use the Receiver Operating Characteristic (ROC) curve and the Area Under the Curve (AUC). We trained PAYL and the single classifiers for different operational points, i.e., we constructed different "versions" of the classifiers setting a different rejection rate on the training dataset each time. This allowed us to plot an approximate ROC curve for each classifier. Assuming the training dataset contains only normal HTTP requests, the rejection rate can be interpreted as a *desired false positive* rate. In the following we refere to this desired false positive rate as DFP. If we also assume the test dataset contains only normal HTTP requests, we can use it to estimate the *"real" false positive* rate, or RFP. Each point on an ROC curve represents the RFP and the detection rate (DR) produced by the classifier. The detection rate is measured on the attack dataset and is defined as the faction of detected attack packets, i.e., the number of attack packets that are classified as anomalous divided by the total number of packets in the attack dataset (regardless of the specific attack the detected packets come from).

We measured the performance of the classifiers for 7 different operational points to compute an (partial) ROC curve for each classifier. These points are obtained by training each classifier using 7

DFP(%)	RFP(%)	Detected attacks	DR(%)
0.0	0.00022	1	0.8
0.01	0.01451	4	17.5
0.1	0.15275	17	69.1
1.0	0.92694	17	72.2
2.0	1.86263	17	72.2
5.0	5.69681	18	73.8
10.0	11.05049	18	78.6

Table 5.1: Performance of 1-gram PAYL.

		k				
		10	**20**	**40**	**80**	**160**
ν	**0**	0.9660 (0.4180E-3)	0.9664 (0.3855E-3)	0.9665 (0.4335E-3)	0.9662 (0.2100E-3)	**0.9668** (0.4686E-3)
	1	0.9842 (0.6431E-3)	0.9839 (0.7047E-3)	**0.9845** (0.7049E-3)	0.9833 (1.2533E-3)	0.9837 (0.9437E-3)
	2	0.9866 (0.7615E-3)	0.9867 (0.6465E-3)	0.9875 (0.6665E-3)	**0.9887** (2.6859E-3)	0.9862 (0.7753E-3)
	3	0.9844 (1.2207E-3)	0.9836 (1.1577E-3)	**0.9874** (1.0251E-3)	0.9832 (1.0619E-3)	0.9825 (0.6835E-3)
	4	0.9846 (0.5612E-3)	0.9847 (1.5334E-3)	0.9846 (0.9229E-3)	0.9849 (1.5966E-3)	**0.9855** (0.4649E-3)
	5	0.9806 (0.8638E-3)	0.9813 (0.9072E-3)	0.9810 (0.5590E-3)	0.9813 (0.8494E-3)	**0.9818** (0.3778E-3)
	6	0.9809 (0.7836E-3)	0.9806 (1.1608E-3)	**0.9812** (1.6199E-3)	0.9794 (0.3323E-3)	0.9796 (0.4240E-3)
	7	0.9819 (1.6897E-3)	0.9854 (0.8485E-3)	0.9844 (1.2407E-3)	0.9863 (1.9233E-3)	**0.9877** (0.7670E-3)
	8	0.9779 (1.7626E-3)	0.9782 (1.9797E-3)	0.9787 (2.0032E-3)	**0.9793** (1.0847E-3)	0.9785 (1.7024E-3)
	9	0.9733 (3.1948E-3)	**0.9775** (1.9651E-3)	0.9770 (1.0803E-3)	0.9743 (2.4879E-3)	0.9722 (1.2258E-3)
	10	0.9549 (2.7850E-3)	0.9587 (3.3831E-3)	0.9597 (3.8900E-3)	0.9608 (1.2084E-3)	**0.9681** (7.1185E-3)

Table 5.2: Performance of single one-class SVM classifiers. The numbers in bold represent the best average AUC for a fixed value of ν. The standard deviation is reported between parentheses.

DFP, namely 0%, 0.01%, 0.1%, 1.0%, 2.0%, 5.0% and 10.0%. The AUC is estimated by integrating the ROC curve in the interval of RFP between 0% and 10.0%. The obtained result is then normalized so that the maximum possible value for the AUC is 1. According to how the AUC is computed, the higher the value of the AUC, the better the performance of the classifier in the considered interval of false positives. For each DFP, we also measured the number of detected attacks. We consider an attack as detected if at least one out of the total number of packets of the attack is detected as anomalous. It is worth noting that the number of detected attacks is different from the detection rate used to computed the ROC curve.

1**-gram PAYL.** Our baseline is represented by the performance of 1-gram PAYL. As mentioned before, PAYL measures the occurrence frequency of byte values in the payload. A separate model is generated for each different payload length. These models are clustered together at the end of the training to reduce the total number of models. Furthermore, the length of a payload is also monitored for anomalies. Thus, a payload with an unseen or very low frequency length is flagged as an anomaly [112].

We trained PAYL using the entire first day of collected HTTP requests. We constructed the ROC curve by estimating the RFP on the entire test dataset, i.e., the other 4 days of collected HTTP requests, and the detection rate on the attack dataset. The obtained AUC was equal to 0.73. As shown in Table 5.1, for DFP=0.1% PAYL produced an RFP=0.15% and was able to detect 17 out of 18 attacks. In particular it was able to detect all the attacks except the polymorphic blending attack.

Table 5.1 also shows that the polymorphic blending attack remained undetected until RFP > 1.86%. By performing further experiments, we found out that the minimum amount of RFP for which PAYL is able to detect all the attacks, included the polymorphic blending attack, is equal to 4.02%, which is usually considered intolerably high for network intrusion detection.

Single One-Class SVM Classifiers. We constructed several one-class SVM classifiers. We extracted the features as described in Section 5.3.3 varying the parameter ν from 0 to 10, thus obtaining 11 different descriptions of the patterns. Then, for each fixed ν, we applied the feature clustering algorithm described in Section 5.3.3 fixing the prior probability $P(C_o) = 0.01$ and setting the number of desired clusters k equal to 10, 20, 40, 80 and 160. We used a random initialization for the algorithm (i.e., at the first step each feature is randomly assigned to one of the k clusters). The feature clustering algorithm stops when the information loss in (5.7) becomes minor than 10^{-4}.

For each pair (ν, k) of parameter values we repeated the experiment 5 times. For each round we applied the feature clustering algorithm (using a new random initialization), and we trained a classifier on a sample of the training dataset obtained from the original training dataset by applying the bootstrap technique without replacement and with a sampling ratio equal to 10%. We estimated the AUC by measuring the false positives on a sample of the test dataset obtained using again the bootstrap technique with sampling ratio equal to 10%, and measuring the detection rate on the entire attack dataset. Table 5.2 reports the estimated average AUC. The numbers between parentheses represent the standard deviation computed over the 5 rounds. We discuss the obtained results later in this section comparing them to the results obtained using the MD classification algorithm.

		k				
		10	**20**	**40**	**80**	**160**
ν	**0**	**0.9965** (0.5345E-3)	0.9948 (1.4455E-3)	0.9895 (3.9813E-3)	0.9785 (5.1802E-3)	0.9718 (9.9020E-3)
	1	**0.9752** (0.5301E-3)	0.9729 (0.7921E-3)	0.9706 (1.0940E-3)	0.9664 (2.2059E-3)	0.9653 (0.3681E-3)
	2	**0.9755** (0.2276E-3)	0.9743 (0.4591E-3)	0.9741 (0.9121E-3)	0.9676 (0.1084E-3)	0.9661 (0.4246E-3)
	3	**0.9749** (0.7496E-3)	0.9736 (0.8507E-3)	0.9726 (1.8217E-3)	0.9714 (1.2729E-3)	0.9708 (2.6994E-3)
	4	**0.9761** (0.4269E-3)	0.9743 (0.3552E-3)	0.9735 (0.7998E-3)	0.9737 (0.3827E-3)	0.9722 (0.9637E-3)
	5	**0.9735** (1.0645E-3)	0.9692 (0.3607E-3)	0.9694 (1.0499E-3)	0.9626 (2.4574E-3)	0.9606 (1.9866E-3)
	6	**0.9737** (0.6733E-3)	0.9709 (1.5523E-3)	0.9687 (2.9730E-3)	0.9699 (4.1122E-3)	0.9717 (0.5427E-3)
	7	**0.9687** (3.3302E-3)	0.9545 (9.6519E-3)	0.9505 (7.3100E-3)	0.9258 (19.923E-3)	0.8672 (50.622E-3)
	8	**0.9731** (0.7552E-3)	0.9721 (0.6001E-3)	0.9717 (0.6799E-3)	0.9715 (0.6367E-3)	0.9678 (1.5209E-3)
	9	**0.9719** (1.5743E-3)	0.9695 (1.9905E-3)	0.9700 (2.2792E-3)	0.9662 (2.9066E-3)	0.9611 (1.5542E-3)
	10	0.9641 (1.6604E-3)	**0.9683** (2.5370E-3)	0.9676 (1.2692E-3)	0.9635 (1.1016E-3)	0.9598 (0.6209E-3)

Table 5.3: Performance of single MD classifiers. The numbers in bold represent the best average AUC for a fixed value of ν. The standard deviation is reported between parentheses.

ν										
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0.9744	0.9665	0.9711	0.9393	0.9170	0.8745	0.8454	0.8419	0.8381	0.9556	0.9079

Table 5.4: Performance of single MD classifiers for varying ν. No feature clustering is applied. The number in bold represents the best result.

Single MD Classifiers. Similarly to the experiments with the one-class SVM classifiers, we constructed several MD classifiers. For each pair (ν, k) of parameter values, we applied the feature clustering algorithm with random initialization, and we trained a classifier on a 10% sample of the training set (using again the bootstrap technique without replacement). The AUC was estimated by measuring the false positives on a 10% sample of the test dataset and the detection rate on the entire attack dataset. We repeated each experiment 5 times. Table 5.3 reports the average and the standard deviation for the obtained AUC. The MD classifier performs extremely well for $\nu = 0$ and $k = 10$. In this case the MD classifier is able to detect all of the 18 attacks for an RFP around 0.1% and reaches 100% of detection rate for an RFP around 1%. However, the use of only one classifier does not improve the hardness of evasion, as discussed in Section 5.5.

We also estimated the performance of the MD classifiers without applying the feature clustering algorithm. In this case each pattern is described by 65,536 features. We trained a classifier for each value of $\nu = 0, .., 10$ on the entire training dataset and estimated the AUC measuring the false positives and the detection rate on the entire test and attack dataset, respectively. The obtained results are reported in Table 5.4. As can be seen from Table 5.3 and Table 5.4, the best performance for a fixed value of ν are always reached using $k = 10$. The only exception is when $\nu = 10$. In this case the best performance is obtained using $k = 20$. The good performance obtained for low values of k are probably due to the fact that the MD classification algorithm suffers from the curse of dimensionality problem. By reducing the dimensionality of the feature space the MD classifier is able to construct a tighter decision surface around the target class. For each fixed k the best results in terms of AUC were obtained using $\nu = 0$. The only exception is when $k = 160$. In this case the best AUC is obtained for $\nu = 4$. Nevertheless, the AUC obtained for $\nu = 4$ and for $\nu = 0$ are really close, and considering the standard deviation it is not possible to say which classifier performs better than the other. As we discuss in Section 5.5, the amount of structural information extracted from the payload decreases when ν grows. The MD classifier seems to be sensitive to this effect.

By comparing the best results in Table 5.2 and 5.3 (the numbers in bold), it is easy to see that SVM classifiers perform better than MD classifiers in all the cases except when $\nu = 0$ and $\nu = 10$. When $\nu = 10$ the best performance are really close, and considering the standard deviation it is not possible to say which classifier performs better than the other. It is also easy to see that, differently from the MD classification algorithm, the one-class SVM seems not to suffer from the growing dimensionality of the feature space obtained by increasing k. This is probably due to the fact that by using the gaussian kernel the patterns are projected in an infinite-dimensional feature space, so that the dimensionality of the original feature space becomes less important.

2-**gram PAYL.** The MD classifier constructed without applying the feature clustering and setting $\nu = 0$ represents an implementation of 2-gram PAYL that uses one model for all the possible packet lengths. Table 5.5 reports the results obtained with this classifier. It is easy to see that 2-gram PAYL performs better that 1-gram PAYL, if we consider the detection rate DR. This is due to the fact that the simple distribution of 1-grams (i.e., the distribution of the occurrence frequency of the byte values) does not extract structural information from the payload, whereas the distribution of 2-grams conveys byte sequence information. Nevertheless, 2-gram PAYL is not able to detect the polymorphic blending attack even if we are willing to tolerate an RFP as high as 11.25%. This is not surprising given that the polymorphic blending attack we used was specifically tailored to evade 2-gram PAYL.

DFP(%)	RFP(%)	Detected attacks	DR(%)
0.0	0.00030	14	35.2
0.01	0.01794	17	96.0
0.1	0.12749	17	96.0
1.0	1.22697	17	97.6
2.0	2.89867	17	97.6
5.0	6.46069	17	97.6
10.0	11.25515	17	97.6

Table 5.5: Performance of an implementation of 2-gram PAYL using a single MD classifier, $\nu = 0$ and $k = 65,536$.

Classifier Ensembles. We constructed several anomaly IDS by combining multiple classifiers using the simple majority voting rule. We first combined one-class SVM classifiers. For a fixed

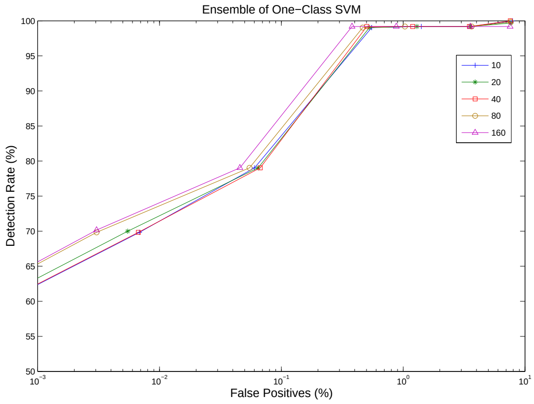

Figure 5.3: ROC curves obtained by combining One-Class SVM classifiers using the majority voting rule. Each curve is related to a different value of *k*. Notice that the scales have been adjusted in order to highlight the differences among the curves.

value of the number of feature clusters k, the output of the 11 classifiers constructed for $\nu = 0, .., 10$ were combined. The experiments were repeated 5 times for each value of k. We also applied the same approach to combine MD classifiers. The obtained ROC curves are reported in Figure 5.3 and Figure 5.4. The average and standard deviation for the obtained AUC are reported in Table 5.6. The last row reports the results obtained by combining single MD classifiers for which no feature clustering was applied (i.e., all the 65,536 features are used). The combination works really well in case of one-class SVM. As shown in Table 5.6, the overall IDS constructed using ensembles of one-class SVM always performs better than the best single classifier. The only exception is when $k = 160$, but in this case the results are so close that considering the standard deviation it is not possible to say which one is the best. On the other hand, the combination of MD classifiers is not as effective as for the ensemble of one-class SVM, and does not improve the performance of the single best classifier. This is probably due to the fact that although we constructed MD classifiers

k	Ensemble of SVM	Ensemble of MD
10	0.9885 (0.3883E-3)	**0.9758** (0.4283E-3)
20	0.9875 (2.0206E-3)	0.9737 (0.1381E-3)
40	**0.9892** (0.2257E-3)	0.9736 (0.2950E-3)
80	0.9891 (1.6722E-3)	0.9733 (0.5144E-3)
160	0.9873 (0.4209E-3)	0.9701 (0.6994E-3)
65,535	-	0.9245

Table 5.6: Average AUC of classifier ensembles constructed using the majority voting rule. The numbers in bold represent the best result for varying *k*. The standard deviation is reported between parentheses.

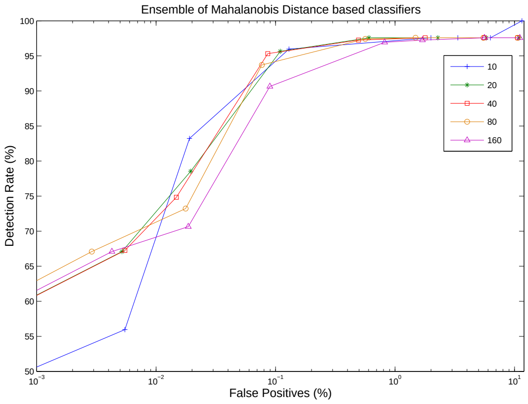

Figure 5.4: ROC curves obtained by combining MD classifiers using the majority voting rule. Each curve is related to a different value of k. Notice that the scales have been adjusted in order to highlight the differences among the curves.

that work on different feature spaces, the obtained classifiers are not sufficiently diverse and make the same errors for new patterns.

DFP(%)	RFP(%)	Detected attacks	DR(%)
0.0	0.0	0	0
0.01	0.00381	17	68.5
0.1	0.07460	17	79.0
1.0	0.49102	18	99.2
2.0	1.14952	18	99.2
5.0	3.47902	18	99.2
10.0	7.50843	18	100

Table 5.7: Performance of an overall IDS constructed using an ensemble of one-class SVM and setting $k = 40$. The DFP is referred to the single classifiers of the ensemble.

Table 5.7 shows the results obtained with an overall IDS implemented by combining the 11 single one-class SVM constructed using $\nu = 0, .., 10$ and $k = 40$. The IDS is able to detect all the attacks except the polymorphic blending attack for an RFP lower than 0.004%. The IDS is also able to detect all the attacks, including the polymorphic blending attack, for an RFP lower than 0.5%.

In conclusion, the experimental results reported above show that our IDS constructed using an ensemble of one-class SVM classifiers and using $k = 40$ performs better than any other IDS or single classifiers we considered. The only exception is the single MD classifier obtained setting $\nu = 0$ and $k = 10$. However, as mentioned before and as discussed in Section 5.5, this single MD classifier may still be easy to evade, whereas our MCS based IDS is much harder to evade.

5.5 Discussion

2_ν-grams. We discussed in Section 5.3.3 how to extract the features using the 2_ν-gram technique. We also argued that the occurrence frequency of 2_ν-grams somehow "summarizes" the occurrence frequency of n-grams. This allows us to capture some byte sequence information. In order to show that the 2_ν-grams actually extract structural information from the payload, we can consider the bytes in the payload as random variables and then we can compute the relative mutual information of bytes that are ν positions apart from each other. That is, for a fixed value of ν we compute the quantity

$$RMI_{\nu,i} = \frac{I(B_i; B_{i+\nu+1})}{H(B_i)} \tag{5.11}$$

where $I(B_i; B_{i+\nu+1})$ is the mutual information of the bytes at position i and $(i + \nu + 1)$, and $H(B_i)$ is the entropy of the bytes at position i. By computing the average for $RMI_{\nu,i}$ over the index $i = 1, .., L$, with L equal to the maximum payload length, we obtain the average relative mutual information for the 2_ν-grams along the payload. We measured this average relative mutual information on both the training and the test set varying ν from 0 to 20. The results are shown in Figure 5.5. It is easy to see that the amount of information extracted using the 2_ν-gram technique is maximum for $\nu = 0$ (i.e., when the 2-gram technique is used) and decreases for growing ν. However the decreasing trend is slow and the average RMI is always higher than 0.5 until $\nu = 10$. This is probably due to the fact that HTTP is a highly structured protocol. Preliminary results show that the same property holds for other text based protocols.

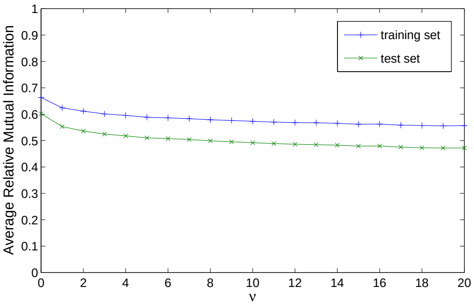

Figure 5.5: Average relative mutual information for varying ν.

Polymorphic Blending Attack. The polymorphic blending attack we used for our performance evaluation was presented in [34] as an attack against 2-gram PAYL. The polymorphic blending attack encodes the attack payload so that the distribution of 2-grams in the transformed attack "looks" like normal, from the point of view of the model of normal traffic constructed by PAYL. As discussed in [34], a polymorphic attack against 2-gram PAYL is also able to evade 1-gram PAYL. This is because the distribution of 1-grams can be derived from the distribution of 2-grams. Thus, if the distribution of 2-grams in the attack payload "looks" like normal, so does the distribution of 1-grams.

In order to construct the attack, first of all the attacker needs to monitor part of the traffic towards the network protected by the IDS [34]. By monitoring this traffic, a polymorphic blending engine constructs an approximate normal profile for the 2-grams and transforms the attack payload accordingly. It has been proved that a "perfect" single byte encoding transformation of the attack payload in order to reflect the estimated normal profile is NP-complete [34]. Therefore, Fogla et al. [34] proposed an approximate solution to the problem. High frequency 2-grams in the attack payload are greedily matched via one-to-one byte substitution with 2-grams that have high frequencies in normal traffic. This approximate substitution does not guarantee to find a transformation that brings the attack payload close to the distribution of normal traffic [34]. The proposed approach could also be generalized to evade an n-gram version of PAYL. However, because of the way the algorithm greedily matches n-grams in the attack payload with n-grams in normal traffic [34], the single byte encoding algorithm proposed is less and less likely to generate a successful attack payload transformation, as n grows. This means that although the polymorphic blending attack may still work well for $n = 2$, it is likely to fail for $n \gg 2$.

Hardness of Evasion. In Section 5.4.2 we showed that an MD classifier constructed using $\nu = 0$ (i.e., using the 2-gram technique) and $k = 10$ achieves very good classification performances (see Table 5.3). However, the use of only one classifier does not help in hardening the anomaly detector against evasion attempts. The attacker may easily modify the polymorphic blending attack against 2-gram PAYL in order to evade this one particular classifier.

We constructed our anomaly IDS using multiple classifiers that work on different descriptions of the payload. In this case the polymorphic blending attack that mimics the normal distribution of 2-grams does not work anymore because it can already be detected for a percentage of false positives as low as 0.5%, as shown by the experimental results reported in Section 5.4.2. In order for the attacker to evade our IDS, she needs to devise a substitution algorithm that evades the majority of the classifiers at the same time. Therefore, the attacker needs to transform the attack payload in order to mimic the distribution of 2_ν-grams for different values of ν. Because of the way the features are extracted using the 2_ν-gram technique, this result may be achieved by a polymorphic transformation that encodes the attack payload to reflect the distribution of the n-grams in normal traffic, with n greater than $\frac{max(\nu)+2}{2}$. Here $max(\nu)$ represents the maximum value of ν used during the feature extraction process. Thus, in order to evade our IDS the attacker needs to encode the attack payload mimicking the distribution in normal traffic of 7-grams. This makes it much harder to evade our IDS, compared to 1-gram and 2-gram PAYL. In theory a hypothetical 7-gram PAYL would be as hard to evade as our IDS. However, this hypothetical 7-gram PAYL would easily suffer from the curse of dimensionality and memory consumption problems due to the huge number of features (equal to 256^7). Our anomaly IDS is much more practical.

Chapter 6

Conclusion

Statistical pattern recognition techniques have been successfully applied in many fields. Relatively recently, researchers have stared to apply pattern recognition to computer and network security, and in particular to network intrusion detection systems. We believe statistical pattern recognition will play a more and more important role in the development of future network IDS. Motivated by this belief, in this thesis we have studied the main challenges and possible solutions related to the application of statistical pattern recognition techniques for designing network IDS. Our objective was to point out strengths and weaknesses of such systems, and to stimulated further research on the problems and solutions discussed throughout the thesis.

6.1 Our Contribution

In this thesis we focused our attention on three main problems:

a) **Learning from unlabeled traffic**. We discussed the state of the art in unlabeled network anomaly detection and the inherent difficulties related to learning from unlabeled data. We also discussed the base-rate fallacy and how it affects anomaly detection systems, showing that it is critical to optimize the accuracy of network anomaly detectors in order to increase the detection rate and, in particular, to decrease the false positive rate as much as possible. To this end, we studied the application of a modular MCS constructed by combining multiple one-class classifiers. Experiments performed on the KDDCup'99 dataset showed that the proposed approach improves the accuracy performance, compared to “monolithic” unlabeled network anomaly detectors proposed by other researchers.

b) **Learning in adversarial environment**. We studied the consequences of learning from unlabeled traffic in presence of an adversary who may try to mislead the learning process to make the obtained IDS ineffective. We briefly discussed some theoretical work on learning in presence of an adversary that introduces malicious errors in the training dataset. Then, we presented a case study and showed that this kind of attacks are possible in practice. We showed how an attacker can inject noise into the training dataset used by automatic signature generators during the signature learning process, and how this attack may negatively affect the accuracy of IDS which use the generated signatures to stop the propagation of worms. We also discussed possible ad-hoc solutions to the noise injection attack, although a generic solution to the problem of learning in presence of this kind of misleading attack is still to be found.

c) **Operating in adversarial environment**. We also studied the problem of launching and detecting mimicry attacks. Mimicry attacks are evasive attacks against anomaly detection systems. We presented an attack called Polymorphic Blending Attack (PBA), which is able to evade recently proposed network anomaly detectors based on statistical pattern recognition techniques. We analyze the reasons why the attack works and propose a new and robust network anomaly IDS intended to make the PBA unlikely to succeed. Our IDS is constructed by combining multiple one-class SVM classifiers. Experiments were performed on several days of normal HTTP traffic of an academic network and on 18 attacks, including “standard” polymorphic attacks and the PBA. The results show that the proposed IDS is more robust to the PBA, compared to other recently proposed network anomaly IDS.

6.2 Future Work

Future work is needed on all of the three problems we addressed in this theses. Further improvements are needed regarding the accuracy performance of unlabeled anomaly IDS. We plan to study different modular MCS schemes in order to further reduce the false positive rate while keeping a high detection rate at the same time, thus alleviating the base-rate fallacy problem presented in Chapter 3. However, in order to achieve this result more work is needed on estimating *a posteriori* class probabilities using one-class classifiers and on effectively combining them to construct a more accurate multiple one-class classifier system.

Regarding the problem of learning in adversarial environment, the effort the attacker has to do in case of real applications is mostly unknown. For example, how difficult is it for the attacker to “move” the decision surface constructed by complex classifiers (e.g., SVM, Artificial Neural Networks, etc.)? Can the attacker approximate the state of the IDS, i.e., its decision surface, without having access to the entire training dataset used by the IDS? How much “misleading” traffic does the adversary has to inject? An answer to these important questions is needed. Moreover, work is needed on the practical application of *disinformation* and *randomization* techniques to make it difficult for the attacker to successfully interfere with the learning process implemented by IDS.

Further work is also needed for making anomaly-based network IDS more robust against mimicry attacks. The combination of multiple models, as proposed in Chapter 5, is definitely promising. However, we need to make mimicry attacks as unlikely to succeed as possible. Because one of the fundamental assumptions for the success of mimicry attacks, and in particular for the polymophic bleding attack, is the adversary’s knowledge about the detection algorithm implemented by the IDS, *disinformation* and *randomization* techniques similar to those proposed for the problem of learning in adversarial environment may be implemented in combination to ensemble methods. This would make much harder for the adversary to approximate the model of normal traffic used by the IDS, thus making mimicry attacks very difficult.

Bibliography

[1] Mini-SQL w3-msql buffer overflow vulnerabilities, 1999. http://www.securityfocus.com/bid/898.

[2] Cern HTTPd Proxy, 2000. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-79.

[3] InetServ 3.0 WebMail Long GET request vulnerability, 2000. http://www.securityfocus.com/bid/949.

[4] Microsoft IIS 4(NT4) and IIS 5(Windows 2k) .asp buffer overflow exploit, 2002. http://downloads.securityfocus.com/vulnerabilities/exploits/DDK-IIS.c.

[5] OmniHTTPD Long Request buffer overflow vulnerability, 2002. http://www.securityfocus.com/bid/5136.

[6] AnalogX Proxy 4.13 buffer overflow vulnerability, 2003. http://www.securityfocus.com/archive/1/322861.

[7] ActivePerl perlIIS.dll buffer overflow vulnerability, 2006. http://www.securityfocus.com/bid/3526/.

[8] Microsoft IIS 5.0 .printer ISAPI Extension buffer overflow vulnerability, 2006. http://www.securityfocus.com/bid/2674/.

[9] Netscape FastTrack Server GET buffer overflow vulnerability, 2006. http://www.securityfocus.com/bid/908/.

[10] D. Anderson, T. Lunt, H. Javitz, and A. Tamaru. Nides: Detecting unusual program behavior using the statistical component of the next generation intrusion detection expert system. Technical Report SRI-CSL-95-06, Computer Science Laboratory, SRI International, Menlo Park, CA, USA, May 1995.

[11] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co, Fort Washington, 1980. `http://seclab.cs.ucdavis.edu/projects/history/CD/ande80.pdf`.

[12] S. Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security (TISSEC)*, 3(3):186–205, 2000.

[13] N. E. Ayat, M. Cheriet, and C. Y. Suen. Automatic model selection for the optimization of SVM kernels. *Pattern Recognition*, 38(10):1733–1745, 2005.

[14] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar. Can machine learning be secure? In *ACM Symposium on Information, Computer and Communications Security*, 2006.

[15] G. Batista, R. C. Prati, and M. C. Monard. A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD Explorations Newsletter*, 6(1):20–29, 2004.

[16] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[17] A. P. Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[18] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4), 2003.

[19] R. Brunelli and D. Falavigna. Person identification using multiple cues. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(10):955–966, 1995.

[20] C. Chang and C. Lin. LIBSVM: a library for support vector machines, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[21] R. Chinchani and E.V.D. Berg. A fast static analysis approach to detect exploit code inside network flows. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[22] I. Cohen, F. G. Cozman, N. Sebe, M. C. Cirelo, and T. Huang. Semi-supervised learning of classifiers: theory, algorithms and their applications to human-computer interaction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(12):1553–1567, 2004.

[23] W. W. Cohen. Fast effective rule induction. In *International Conference on Machine Learning*, 1995.

[24] M. Damashek. Gauging similarity with n-grams: Language-independent categorization of text. *Science*, 267(5199):843–848, 1995.

[25] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[26] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack Issue 0x3d*, 2003.

[27] I. S. Dhillon, S. Mallela, and R. Kumar. A divisive information-theoretic feature clustering algorithm for text classification. *Journal of Machine Learning Research*, 3:1265–1287, 2003.

[28] T. G. Dietterich. Ensemble methods in machine learning. In *International Workshop on Multiple Classifier Systems (MCS)*, 2000.

[29] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley, 2000.

[30] C. Elkan. Results of the KDD'99 classifier learning contest. *ACM SIGKDD Explorations Newsletter*, 1(2):63–64, 2000.

[31] E. Eskin, A. Arnold, M. Prerau, L. Portnoy, and S. Stolfo. A geometric framework for unsupervised anomaly detection: Detecting intrusions in unlabeled data. In D. Barbara and S. Jajodia, editors, *Applications of Data Mining in Computer Security*. Kluwer, 2002.

[32] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred. Statistical approaches to DDoS attack detection and response. In *DARPA Information Survivability Conference and Exposition (DISCEX)*, 2003.

[33] Firew0rker. Windows Media Services remote command execution exploit, 2003. `http://www.milw0rm.com/exploits/48`.

[34] P. Fogla, M. Sharif, R. Perdisci, O. M. Kolesnikov, and W. Lee. Polymorphic blending attack. In *USENIX Security Symposium*, 2006.

[35] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless inter-domain routing (cidr): an address assignment and aggregation strategy,. `http://www.ietf.org/rfc/rfc1519.txt`, 1993.

[36] G. Giacinto, F. Roli, and L. Didaci. Fusion of multiple classifiers for intrusion detection in computer networks. *Pattern Recognition Letters*, 24(12):1795–1803, 2003.

[37] G. Giacinto, F. Roli, and L. Didaci. A modular multiple classifier system for the detection of intrusions in computer networks. In *Multiple Classifier Systems*, volume LNCS 2709, pages 346–355, 2003.

[38] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson. CSI/FBI computer crime and security survey, 2006. Computer Security Institute (`http://www.gocsi.com`).

[39] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *USENIX Security Symposium*, pages 115–131, 2001.

[40] K. Haugsness. Intrusion Detection FAQ: What is polymorphic shell code and what can it do? SANS Institute. (`http://www.sans.org/resources/idfaq/polymorphic_shell.php`).

[41] A. Jain, L. Hong, and Y. Kulkarni. A multimodal biometric system using fingerprint, face and speech. In *Proceeding of the Conference on Audio and Video based Person Authentication (AVBPA)*, 1999.

[42] A. Jain, A. Ross, and S. Prabhakar. Fingerprint matching using minutiae and texture features. In *International Conference on Image Processing (ICIP)*, 2001.

[43] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[44] A. K. Jain, R. P. W. Duin, and J. Mao. Statistical pattern recognition: A review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):4–37, 2000.

[45] N. Japkowicz and S. Stephen. The class imbalance problem: A systematic study. *Intelligent Data Analysis*, 6(5):429–449, 2002.

[46] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, October 2003.

[47] M. Kearns and M. Li. Learning in the presence of malicious errors. *SIAM Journal on Computing*, 22:807–837, 1993.

[48] H. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[49] S. T. King, P. M. Chen, Y. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy*, pages 314–327, 2006.

[50] J. Kittler, M. Hatef, R. P. W. Duin, and J. Matas. On combining classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(3):226–239, 1998.

[51] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[52] O. Kolesnikov, D. Dagon, and W. Lee. Advanced polymorphic worms: Evading ids by blending in with normal traffic. Technical report, Georgia Institute of Technology, 2004. `http://www.cc.gatech.edu/~ok/w/ok_pw.pdf`.

[53] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. In *Proceedings of the 2nd ACM Workshop on Hot Topics in Networks*, November 2003.

[54] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Security Symposium*, 2005.

[55] C. Kruegel, T. Toth, and E. Kirda. Service specific anomaly detection for network intrusion detection. In *ACM Symposium on Applied Computing (SAC)*, 2002.

[56] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *ACM Conference on Computer and Communication Security (ACM CCS)*, 2003.

[57] L. Kuncheva. *Combining Pattern Classifiers*. Wiley, 2004.

[58] C. E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(13):247–278, 1981.

[59] W. Lee. *A Data Mining Framework for Constructing Features and Models for Intrusion Detection Systems*. PhD thesis, Computer Science Department, Columbia University, New York, USA, 1999.

[60] W. Lee and S. Stolfo. A framework for constructing features and models for intrusion detection systems. *ACM Transactions on Information and System Security*, 3(4):227–261, 2000.

[61] W. Lee and D. Xiang. Information-theoretic measures for anomaly detection. In *IEEE Symposium on Security and Privacy*, 2001.

[62] E. Leopold and J. Kindermann. Text categorization with support vector machines. How to represent texts in input space? *Machine Learning*, 46:423–444, 2002.

[63] K. Leung and C. Leckie. Unsupervised anomaly detection in network intrusion detection using clusters. In *In 28th Australasian Computer Science Conference*, 2005.

[64] Z. Liang and R. Sekar. Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In *Proceedings of the 21st Annual Computer Security Applications Conference*, December 2005.

[65] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 darpa off-line intrusion detection evaluation. *Computer Networks*, 34(4):579–595, 2000.

[66] M. V. Mahoney. Network traffic anomaly detection based on packet bytes. In *ACM Symposium on Applied Computing (SAC)*, 2003.

[67] M. V. Mahoney and P. K. Chan. An analysis of the 1999 DARPA/Lincoln Laboratory evaluation data for network anomaly detection. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2003.

[68] Matthew V. Mahoney and Philip K. Chan. Learning nonstationary models of normal network traffic for detecting novel attacks. In *ACM SIGKDD international conference on Knowledge discovery and data mining*, 2002.

[69] J. McHugh. Intrusion and intrusion detection. *International Journal of Information Security*, 1(1):14–35, 2001.

[70] J. McHugh. Testing intrusion detection systems: A critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Transactions on Information and System Security*, 3(4):262–294, November 2006.

[71] J. McHugh, A. Christie, and J. Allen. Defending yourself: The role of intrusion detection systems. *IEEE Software*, pages 42–51, Sept./Oct. 2000.

[72] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security & Privacy*, pages 33–39, July-August 2003.

[73] D. Moore, C. Shannon, and J. Brown. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the Internet Measurement Workshop (IMW)*, November 2002.

[74] S. Mukkamala, G. Janoski, and A. Sung. Intrusion detection using neural networks and support vector machines. In *International Joint Conference on Neural Networks (IJCNN)*, 2002.

[75] D. Mutz, G. Vigna, and R.A. Kemmerer. An experience developing an ids stimulator for the black-box testing of network intrusion detection systems. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2003.

[76] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[77] J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting signature learning by training maliciously. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[78] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the 12th Network and Distributed System Security Symposium*, February 2005.

[79] A. One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[80] S. Peddabachigari, A. Abraham, and J. Thomas. Intrusion detection systems using decision trees and support vector machines. *International Journal of Applied Science and Computations*, 11(3):118–134, 2004.

[81] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. In *IEEE Symposium on Security and Privacy*, 2006.

[82] R. Perdisci, G. Giacinto, and F. Roli. Alarm clustering for intrusion detection systems in computer networks. *Engineering Applications of Artificial Intelligence*, 19(4):429–438, 2006.

[83] B. Pfahringer. Winning the KDD'99 classication cup: Bagged boosting. *ACM SIGKDD Explorations Newsletter*, 1(2):65–66, 2000.

[84] L. Portnoy, E. Eskin, and S. Stolfo. Intrusion detection with unlabeled data using clustering. In *ACM CSS Workshop on Data Mining Applied to Security*, 2001.

[85] T. Ptacek and T. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection, 1998. Secure Networks, Inc. (`http://www.nai.com/services/support/whitepapers/security/IDSpaper.pdf`).

[86] Symantec Security Response. W32.spybot.worm. `http://securityresponse.symantec.com/avcenter/venc/data/w32.spybot.worm.html`.

[87] Symantec Security Response. W32.welchia.c.worm. `http://securityresponse.symantec.com/avcenter/venc/data/w32.welchia.c.worm.html`.

[88] Symantec Security Response. W32.sasser.worm. `http://securityresponse.symantec.com/avcenter/venc/data/w32.sasser.worm.html`, 2004.

[89] Symantec Security Response. W32.zotob.a. `http://securityresponse.symantec.com/avcenter/venc/data/w32.zotob.a.html`, 2005.

[90] M. Roesch. Snort - Lightweight intrusion detection for networks. In *USENIX Systems Administration Conference*, pages 229–238, 1999.

[91] S. Rubin, Somesh Jha, and Barton P. Miller. Automatic generation and analysis of nids attacks. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, 2004.

[92] B. Schölkopf, J. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Computation*, 13:1443–1471, 2001.

[93] F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47, March 2002.

[94] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation*, December 2004.

[95] Sniphs. Snot. `http://www.stolenshoes.net/sniph/index.html`, 2003.

[96] D. Song. Fragroute. `http://www.monkey.org/~dugsong/fragroute/`, 2005.

[97] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time. In *USENIX Security Symposium*, 2002.

[98] K.M.C. Tan, K.S. Killourhy, and R.A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.

[99] Y. Tang and S. Chen. Defending against internet worms: A signature-based approach. In *Proceedings of the 24th Annual Conference IEEE INFOCOM 2005*, March 2005.

[100] D. M. J. Tax. *One-Class Classification, Concept Learning in the Absence of Counter Examples*. PhD thesis, Delft University of Technology, Delft, Netherland, 2001.

[101] D. M. J. Tax and R. P. W. Duin. Combining one-class classifiers. In *International Wokshop on Multiple Classifier Systems (MCS)*, 2001.

[102] D. M. J. Tax and R. P. W. Duin. Support vector data description. *Mchine Learning*, 54(1):45–66, 2004.

[103] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.

[104] A. Valdes and K. Skinner. Probabilistic alert correlation. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2001.

[105] F. Valeur, G. Vigna, C. Kruegel, and R. A. Kemmerer. Comprehensive approach to intrusion detection alert correlation. *IEEE Transactions on Dependable and Secure Computing*, 1(3):146–169, 2004.

[106] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[107] V. Vapnik. *Statistical Learning Theory*. Wiley, 1998.

[108] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *ACM Conference on Computer and Communication Security (ACM CCS)*, pages 21–30, 2004.

[109] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM Conference on Computer and Communication Security (ACM CCS)*, 2002.

[110] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *ACM SIGCOMM Conference*, 2004.

[111] K. Wang, J. J. Parekh, and S. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[112] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.

[113] K. Wang and S. Stolfo. Anomalous payload-based worm detection and signature generation. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[114] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

[115] C. Wong, C. Wang, D. Song, S. Bielski, and G. R. Ganger. Dynamic quarantine of internet worms. In *International Conference on Dependable Systems and Networks*, July 2004.

[116] Y. Yang and F. Ma. An unsupervised anomaly detection patterns learning algorithm. In *International Conference on Communication Technology (ICCT)*, 2003.

[117] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[118] S. Zanero and S. M. Savaresi. Unsupervised learning techniques for an intrusion detection system. In *ACM Symposium on Applied Computing (SAC)*, 2004.

[119] C. C. Zou, W. Gong, and D. Towsley. Worm propagation modeling and analysis under dynamic quarantine defense. In *ACM CCS Workshop on Rapid Malcode*, 2003.