Detecting Network Attacks Using High-Resolution Time Series

Mohamed Wasim Lorgat

Supervisor: Professor Alireza Baghai-Wadji

Department of Electrical Engineering

University of Cape Town

This dissertation is submitted for the degree of

Master of Science in Electrical Engineering

February 2018

Declaration

I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own. This dissertation has been submitted to the Turnitin module and I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.

Signed by candidate

Mohamed Wasim Lorgat

February 2018

Acknowledgements

Thanks to my supervisor, Professor Alireza Baghai-Wadji, for his ongoing mentorship and guidance in topics ranging from engineering to life in general. Thanks to Andre McDonald, for his many contributions despite working on his own PhD. I am thankful to my colleagues and friends, Mahmood Akbari, Bruce Mkhaliphi, Brenwen Ntlangu, Oladejo Sunday, and especially Lateef Akinyemi, who has been a close friend and advisor throughout my studies.

Finally, I thank my family for their continued support and encouragement; I am indebted to my wife for her endless patience and love.

Funding Statement

I gratefully acknowledge the interest of Dr Joey Jansen van Vuuren, Manager of the Cybersecurity Centre of Innovation, CSIR, South Africa, in this project and her continuous encouragement. I am indebted to ARMSCOR for being awarded the prestigious Ledger Bursary Grant. I also acknowledge the award of a joint UCT/CSIR Seed Grant, which enabled the initiation of the project.

Abstract

Research in the detection of cyber-attacks has sky-rocketed in the recent past. However, there remains a striking gap between usage of the proposed algorithms in academic research versus industrial applications. Leading researchers have argued that efforts toward the understanding of proposed detectors are lacking. By digging deeper into their inner workings and critically evaluating their underlying assumptions, better detectors may be built. The aim of this thesis is therefore to provide an underlying theory for understanding a single class of detection algorithms, in particular, anomaly-based network intrusion detection algorithms that utilise high-resolution time series data.

A framework is proposed to deconstruct the algorithms into their constituent components (windows, representations, and deviations). The framework is applied to a class of algorithms, allowing to construct a "space" of algorithms spanned by five variables: windowing procedure, information availability, single- or multi-aggregated representation, marginal distribution model, and deviation. The detection of a simple class of Denial-of-Service (DoS) attacks is modelled as a detection theoretic problem. It is shown that the effect of incomplete information is greatest when detecting low-intensity attacks (less than 5%), however, the effect slowly decays as the attack intensity increases. Next, the representation and deviation components are jointly analysed via a proposed experimental procedure using network traffic from two publicly available datasets: the Measurement and Analysis on the WIDE Internet (MAWI) archive, and the Booters dataset. The experimental analysis shows that varying the representation (single- versus multi-aggregated) has little effect on detection accuracy, and that the likelihood deviation is superior to the L_2 distance deviation, although the difference is negligible for large-intensity attacks (approximately 80%).

Table of contents

List of figures

List of tables

List of publications

The following publications have been incorporated into this thesis:

- Chapter 4 extends the preliminary formulations in [1]: M. W. Lorgat, A. R. Baghai-Wadji, and A. McDonald, "Towards a General Framework for Network Traffic Time Series Anomaly Detection," in *Proc. Eur. Conf. Cyber Warfare and Security*, Jun. 2017, pp. 252–260.

- Chapter 5 is substantially based on [2]: M. W. Lorgat, A. R. Baghai-Wadji, and A. McDonald, "Quantifying the Effect of Incomplete Information in Denial of Service Detection," in *Proc. Global Wireless Summit*, Oct. 2017, pp. 67–71.

List of terms and acronyms

Anomaly-based IDS An anomaly-based Intrusion Detection System (IDS) constructs profiles of normal (or expected) behaviour from observed data and identifies deviations from the constructed profiles as potential attacks.

AUC The Area Under the Receiver Operating Characteristic Curve (AUC) summarises the accuracy of a detector.

Bivariate distribution The bivariate distributions of a stochastic process are the two-dimensional probability distributions of the process at two distinct points in time.

DDoS attack A Distributed Denial-of-Service (DoS) attack is a class of DoS attacks that originate from a multitude of attacking devices.

DoS attack A Denial-of-Service (DoS) attack aims to shut down a victim's device by overflowing their device's capabilities with meaningless data.

FN False Negative.

FP False Positive.

Gaussian process A stochastic process is said to be Gaussian if and only if all of its finite-dimensional joint distributions are multi-variate Gaussian.

IDS An Intrusion Detection System (IDS) gathers data about its environment and attempts to extract information from that data in order to identify possibly malicious activities.

Likelihood function The likelihood function of a given data point is the PDF evaluated at that point as a function of a parameter vector.

Local stationarity A stochastic process is said to be locally stationary if and only if it is stationary on intervals of equal length L.

LRT Likelihood Ratio Test.

MAP Maximum *A Posteriori*.

Marginal distribution The marginal distributions of a stochastic process are the one-dimensional probability distributions of the process at given points in time.

Misuse-based IDS A misuse-based Intrusion Detection System (IDS) constructs profiles of a threat and attempts to match observed data with the constructed profiles to identify potential attacks.

ML Maximum Likelihood.

MPE Minimum Probability of Error.

NP Neyman-Pearson.

PCA Principal Component Analysis.

PDF Probability Density Function.

Periodic stationarity A stochastic process is said to be periodically stationary if and only if its finite-dimensional joint probability distributions (or in the weaker sense, its moments) are periodic with period P.

ROC curve The Receiver Operating Characteristic (ROC) curve is the curve of the probability of detection versus the probability of false alarm parametrised by a decision threshold value. The ROC curve completely specifies the accuracy of a detector.

Sliding window Non-stationary time series are often viewed within a sliding window, such that the observed sub-series are stationary.

Strict stationarity A stochastic process is said to be strictly stationary if and only if all of its finite-dimensional joint probability distributions are invariant to translation.

Time series A time series is a sequence of observations made successively in time.

TN True Negative.

TP True Positive.

Weak stationarity A stochastic process is said to be n^{th}-order weakly stationary if and only if the n^{th}-order moments of all of its n-dimensional joint probability distributions are invariant to translation.

List of symbols

A	Attack intensity
$C_{00}, C_{01}, C_{10}, C_{11}$	Decision costs for binary classification
$D \in \{\texttt{l2-distance}, \texttt{likelihood}\}$	Deviation of a detector
d_t	Deviation score sequence
$E[X]$	Expectation of a random variable X
H_0, H_1	Hypotheses for binary classification
$I \in \{\texttt{incomplete}, \texttt{complete}\}$	Information availability of a detector
L	Length of the local stationarity interval
$\Lambda(x)$	Likelihood Ratio Test (LRT) function evaluated at a realization x of a random variable X
M	Number of window positions
μ_t	Mean of a stochastic process
N	Length of the observed time series
N_{ref}	Length of the reference window
N_{shift}	Length of the sliding window shift
N_{test}	Length of the test window
$p_X(x)$	Probability Density Function (PDF) of a random variable X
$p_{X\|H_0}(x\|H_0), p_{X\|H_1}(x\|H_1)$	Conditional PDF of a random variable X given an hypothesis H_0 or H_1
P	Length of the periodic stationarity interval
$\Pr\{E\}$	Probability of an event E

P_0, P_1	Prior probabilities for binary classification
P_D	Probability of detection
P_e	Probability of error
P_FA	Probability of false alarm
$\Pr\{H_0\|X\}, \Pr\{H_1\|X\}$	Posterior probability of an hypothesis (H_0 or H_1) given a random variable X
$R \in \{\texttt{single}, \texttt{multi}\}$	Representation of a detector
r	Ratio of minimum test window lengths required by two detectors to achieve a given accuracy
r_dec	Decision rate
r_err	Error rate
$\mathscr{R}$	Bayes risk criterion
τ	Actual changepoint
T	Estimated changepoint
$W \in \{\texttt{local}, \texttt{periodic}\}$	Windowing procedure of a detector
$\mathbf{x} = (x_1, x_2, \cdots, x_N)$	An observed time series
$\bar{x}$	Arithmetic mean of a vector $\mathbf{x}$
$\mathbf{x}_m^\mathrm{ref}$ $(m = 1, 2, \cdots, M)$	Reference sub-series
$\mathbf{x}_m^\mathrm{test}$ $(m = 1, 2, \cdots, M)$	Test sub-series

Chapter 1

Introduction

In June 2017, chaos was spread globally by the so-called NotPetya cyber-attack. The attack was disguised as ransomware, a tool used by cyber-criminals to hold a victim's information hostage. However, it was later discovered to be a highly sophisticated weapon designed with one goal in mind – the mass destruction of information assets. The unfortunate reality is that NotPetya is only one of many cyber-attacks occurring on a daily basis. The importance of cyber-security cannot be understated. To that end, the National Institute of Standards and Technology (NIST) in the United States put forward the Cyber-Security Framework [3]. The framework classifies cyber-security activities along five different categories: identify, protect, detect, respond, and recover. In this thesis, efforts are geared towards addressing the third function, the detection of cyber-attacks.

Statement of the Problem

Research in the detection of cyber-attacks has sky-rocketed in the recent past, with numerous proposed algorithms stemming from fields as diverse as machine learning, statistics, and signal processing [4]. However, Sommer and Paxson note that 'despite extensive academic research one finds a striking gap in terms of actual deployments of such systems' [5, p. 305]. In particular, they identified an unfortunate lack of deep insight into how the techniques underlying proposed detectors work; instead they are often treated as black boxes. Along

a similar argument, Gates and Taylor caution that 'little work has gone into determining if the underlying assumptions hold' [6, p. 21]. They identified that proposed algorithms are rarely accompanied by explicit underlying assumptions. Thus, they argue that by critically evaluating the assumptions underlying detectors, future work will be based on more complete knowledge, and perhaps better detectors will be built. In summary, the lack of research in *understanding* detection algorithms has led to a striking gap between academic research and real-world application, which can be remedied by digging deeper into the inner workings of the algorithms and critically evaluating their underlying assumptions.

Aim and Scope

The aim of this thesis is to provide an underlying theory for understanding a single class of cyber-attack detection algorithms. Connecting a class of seemingly disparate algorithms to a single consistent theoretical framework would allow to gain a deeper insight into how the algorithms really work. By exploring their strengths and limitations, we will be able to build better detectors. In addition, by identifying a connection between methods in cyber-security and existing theories in other domains, new research avenues will be highlighted. Finally, this thesis will strive to be completely open and reproducible. The developed computer code is made publicly available in its entirety in the appendices and the described numerical experiments are run on publicly available datasets. The author's intention is for this thesis to serve as part of a foundation for new researchers to more easily enter the domain and provide their own original research.

The focus of this study is on the network level (operating on collections of interconnected hosts) rather than the host level (operating on single entities). Two complementary approaches to detection exist: anomaly-based detection, which operates by building profiles of normal behaviour and detecting deviations from those profiles as potential attacks; and misuse-based detection, which constructs profiles of the threat itself and matches observed data with those profiles. The anomaly-based approach is considered here. Furthermore, due to increasing throughputs and end-to-end encryption, techniques using only packet-header

data (and not payload data) are considered here, also known as flow-based detection. Finally, only methods based on statistical and signal processing domains are studied here.

Overview and Contributions

This thesis is split into two parts. The first part (Chapters 2 and 3) serves as the foundation, upon which the contributions in the second part (Chapters 4 to 6) are built.

To begin with, Chapter 2 details the relevant background in three separate areas: intrusion detection, time series analysis, and detection theory. The discussion is narrowed down in Chapter 3, where related literature is reviewed. The concepts detailed in this part are core to the contributions made in the chapters that follow.

The first contribution (Chapter 4) provides a framework for deconstructing detection algorithms into three components: windows, representations, and deviations. The framework is applied to two state-of-the-art detection algorithms ([7] and [8]), allowing to highlight five key differences between them: windowing procedure, information availability, representation, marginal distribution model, and deviation. Each difference is regarded as a separate variable, such that their collection comprises a "space" of previously unexplored detection algorithms. The exploration of this "space" is the focus of the remaining two contributions.

The second contribution (Chapter 5) models a simple detection problem – the detection of a constant-rate Denial-of-Service (DoS) attack – within the framework of detection theory. Two detectors are constructed with varying levels of information availability. Their comparison, in terms of the Area Under the Receiver Operating Characteristic Curve (AUC) allows to investigate the effect of information availability on detection accuracy.

The third and final contribution (Chapter 6) jointly investigates the effect of variables representation and deviation on detection accuracy. By varying the representation and deviation, four detectors are constructed. An experimental procedure is proposed where each detector is evaluated on one of 128 ($= 9 \times 14$) different real network traffic datasets, obtained by superposing each of 9 attacks of varying intensity on 14 non-attack datasets.

Finally, Chapter 7 concludes and suggests directions for future research.

Chapter 2

Background

This chapter introduces basic concepts in intrusion detection, time series analysis, and detection theory. These three areas serve as the foundation for the chapters that follow.

2.1 Intrusion Detection

An Intrusion Detection System (IDS) gathers data about its environment and attempts to extract information from that data in order to identify possibly malicious activities. Since, typically, most data traversing the network are benign, the process has been likened to 'finding a needle in a haystack.'

Based on a user's authorization level and the activity they are trying to perform, it is straightforward to identify unauthorized activity. Intrusion detection, on the other hand, attempts to detect seemingly authorized activity performed by an intruder. For example, an intruder might masquerade as a user with a higher authorization level to gain access to confidential information. While the action would still appear to be authorized, an IDS is expected to detect that it is in fact intrusive.

IDSs were deemed necessary when experts realized that security flaws in a static defence are unavoidable. In an ideal world, one could design an entity that is perfectly secured. In reality, however, as the complexity of designed entities is continuously growing, it is unreasonable to expect perfectly secured designs. The intruder always finds a way in. The

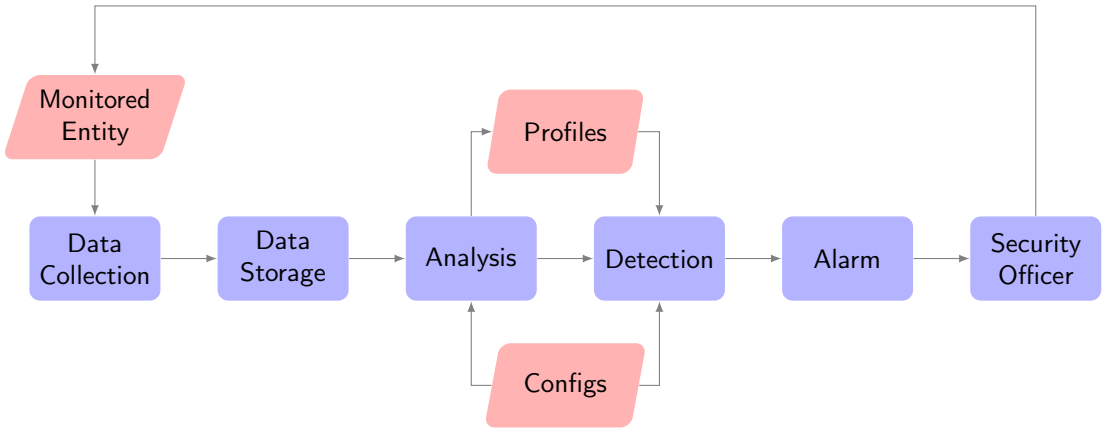


Fig. 2.1. A diagram of a general IDS model, adapted from [9]. Objects (physical or digital) are represented in red, and processes are represented in blue.

intruder cracks the password, phishes for private information, finds a vulnerable back door, or masquerades as someone else and enters through the front door. When an intruder has breached static defence measures, an IDS attempts to dynamically detect their unwelcome presence before significant damage is dealt to the entity.

A general, high-level IDS model was originally proposed by Denning [10], thereafter extended by Axelsson [9], and further adapted here. A diagrammatic description of the model is presented in Fig. 2.1. Note that certain details have been abstracted out of the diagram, e.g., software and hardware required for data collection and storage. Furthermore, not every possible interaction between processes has been drawn in the figure. The purpose of the diagram is to provide a high-level description of the main processes involved in an IDS, namely:

1. *Monitored Entity:* Typically a host (e.g., computer, tablet, or mobile phone) or a network of connected hosts that is to be monitored and secured.

2. *Data Collection:* Relevant data are collected from the monitored entity. In the case of monitored hosts, data might include user login attempts, file manipulations, input and output logs, and CPU usage. In the case of monitored networks, network packets are considered the basic units of data from which different levels of aggregated data can be derived, e.g., NetFlow data.

3. *Data Storage:* A large amount of data can be collected and needs to be stored.

4. *Configuration Settings:* All parameters required by the IDS must be set and stored. Parameters may include threshold values, window lengths, profile construction frequency, update frequency, and detection frequency. Parameters may be adjusted adaptively by the IDS or manually by a security officer.

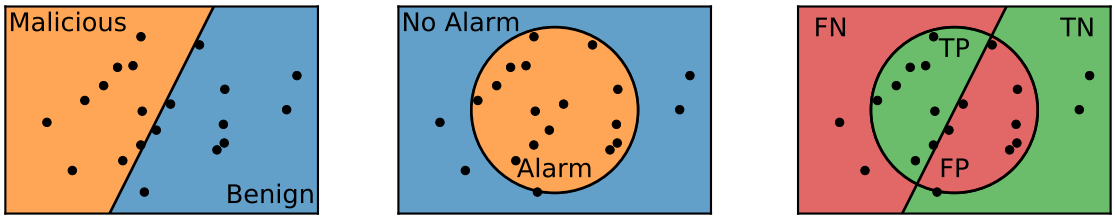
5. *Analysis:* Based on the configuration settings, profiles are constructed from the collected data. Anomaly- and misuse-based IDSs differ in what they construct profiles of. The former constructs profiles of normal (or expected) behaviour, and the latter constructs profiles of malicious behaviour.

6. *Profiles:* Profiles range from simple static signatures (e.g., Snort [11] rule sets) to more complex statistical and signal processing models (e.g., those described in Chapters 3 and 4).

7. *Detection:* Based on the configuration settings, newly observed data are compared against constructed profiles. If some pre-defined decision rule is met, an alarm is raised. In anomaly-based detection, an alarm is raised if observed data sufficiently deviate from pre-constructed profiles of normal behaviour. In misuse-based detection, an alarm is raised if observed data are sufficiently similar to pre-constructed profiles of malicious behaviour.

8. *Alarm:* A report is generated containing all information relevant to a detected incident, allowing the security officer(s) to respond accordingly. Automated responses can also be performed directly on the monitored entity, e.g., drop malicious packets, close malicious connections, close malicious applications, and so on.

9. *Security Officer:* Security officers may manually fine-tune configurations and profiles and are expected to respond to alarms. IDS performance is essentially measured in terms of its usability by security officers. If the IDS overwhelms security officers with excessive false alarms, the system is not usable. If attacks pass by the IDS unnoticed, the system is not trustworthy.



(a) **(b)** **(c)**

Fig. 2.2. A numerical experiment demonstrating the possible outcomes of an IDS for a hypothetical scenario. (a) A set of two-dimensional data points that are either malicious or benign. (b) An arbitrarily chosen circular boundary representing how an IDS might decide which data points cause an alarm or not. (c) The outcomes of the IDS on the dataset.

In the last point of the above description, the term *false alarms* was used. This term will now be described briefly. It is often useful to think of IDSs as binary classifiers. Given a set of data points, an IDS must decide whether each data point is malicious or benign. The outcome of each decision can be correct or incorrect, depending on whether the data point is actually malicious or benign. There are four possible outcomes to any such decision, called true positive (TP), true negative (TN), false positive (FP), and false negative (FN). These outcomes are explained visually in Fig. 2.2 with the aid of a hypothetical scenario. A primary challenge in designing IDSs is to find an optimum between minimizing the false positive rate while maximizing the true positive rate.

Anomaly- and misuse-based detection were briefly introduced in the above discussion. The two complementary approaches will now be compared against each other. Misuse-based detection constructs well-defined profiles of malicious behaviour. If the profiles are matched by observed data, it is highly likely that an intrusion has occurred. However, the misuse-based approach is completely incapable of detecting novel and unknown attacks, since a profile of a novel and unknown attack cannot exist before the attack has been observed. On the other hand, anomaly-based detection operates without a model of the attack scenario, by constructing profiles of normal (or expected) behaviour. Deviations in observed data from the profiles are identified as potential attacks. Anomaly-based IDSs typically have high false alarm rates, since anomalous behaviour is often not malicious, e.g., new software is released, new habits are formed, and so on. However, the anomaly-based

approach allows the detection of novel and unknown attacks, and of attacks that are difficult to characterize by explicit signatures.

Anomaly- and misuse-based detection are complementary approaches. In a comprehensive defence, both approaches must be balanced.

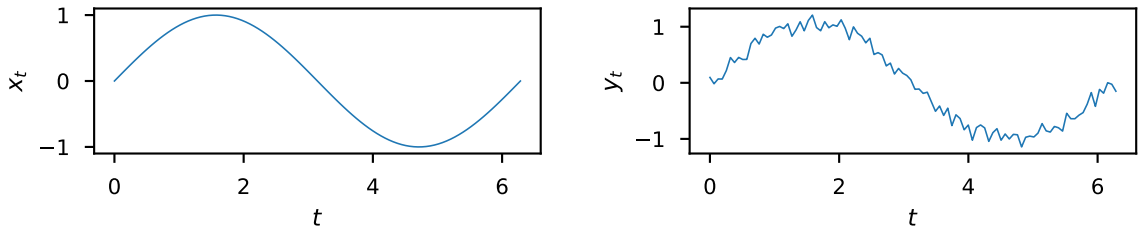
2.2 Time Series Analysis

This section provides a brief discussion of foundational concepts in the analysis of time series that are relevant to this thesis. The discussion does not aim to be comprehensive. The ideas expressed closely follow the textbook by Box, Jenkins, and Reinsel [12] to which the reader is directed for further details.

The analysis of time series ought to begin by defining what a time series is. A time series is considered to be a set of observations $x(\tau_1), x(\tau_2), \cdots, x(\tau_t), \cdots, x(\tau_N)$ made at discrete points in time $\tau_1, \tau_2, \cdots, \tau_t, \cdots, \tau_N$. It is assumed here that the observation times $\{\tau_t\}$ are recorded at fixed intervals of length h starting at some initial time τ_0; i.e., the observation times can be rewritten as $\tau_0 + h, \tau_0 + 2h, \cdots, \tau_0 + th, \cdots, \tau_0 + Nh$. Given the initial time τ_0 and the unit of time h, it is notationally convenient to write the time series as $x_1, x_2, \cdots, x_t, \cdots, x_N$. Here, the index t conveys that x_t was observed t intervals of length h after the initial time τ_0.

Time series may be recorded in one of two ways. Either, by *sampling* a continuous-time variable at discrete points in time (e.g., temperature recorded every minute), or by *accumulating* a variable over time (e.g., the total amount of rainfall recorded each day). Time series may also be categorised as being either *deterministic* or *statistical*. A time series is said to be deterministic if its future value is determined by some mathematical function. On the other hand, a time series is said to be statistical if its future value is described only in terms of a probability distribution. In order to visualize the difference between deterministic and statistical time series, simple examples of both are plotted in Fig. 2.3. Time series dealt with in practice typically consist of both deterministic and statistical components.

In order to model a statistical time series, it is considered a realization of a *stochastic process* – a statistical phenomenon that evolves in time according to probabilistic laws. It



(a) **(b)**

Fig. 2.3. Simple examples of a deterministic time series and a statistical time series. (a) A deterministic time series, $x_t = \sin(t)$, completely specified by a sinusoidal function of time. (b) A statistical time series, $y_t = \sin(t) + \epsilon_t$, specified by a sinusoidal function depending on time in addition to a noise term ϵ_t which can only be described in terms of a probability law.

may be imagined that many other time series could have been generated as realizations of the same underlying stochastic process, although only one particular time series was generated and observed (as visualized in Fig. 2.4). Formally, the observation x_t at a given time t, may be regarded as a realization of a random variable X_t with probability density function $p_t(x_t)$. Similarly, the observations at any two times t_1 and t_2 may be regarded as realizations of two random variables X_{t_1} and X_{t_2} with joint probability density function $p_{t_1,t_2}(x_{t_1}, x_{t_2})$. More generally, the observations at N successive points in time $t = 1, 2, \cdots, N$ may be regarded as a realization of N random variables $X_1, X_2, \cdots, X_N$ with joint probability density function $p_{1,2,\cdots,N}(x_1, x_2, \cdots, x_N)$. Marginal and bivariate distributions estimated from an ensemble of realizations of a stationary Gaussian stochastic process are plotted in Fig. 2.5 to aid understanding of the above concepts.

Thus, in modelling a time series of N observations, it may be regarded as a single realization of N jointly-distributed random variables. For small N it is possible to specify the N-dimensional joint probability density function thus providing a complete model of a given time series. However, specifying a high-dimensional probability distribution function becomes increasingly impractical as N grows. In contrast, it is desired to model a time series using a parsimonious model; i.e., using as few parameters as possible. It often suffices in practice to model only the *second-order properties* of a time series – the first- and second-

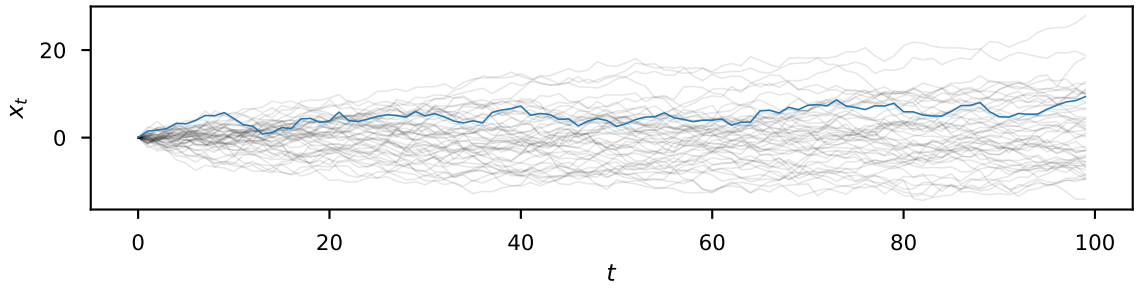


Fig. 2.4. A collection of time series, each a possible realization of the same underlying stochastic process. The iterative generating process in this case is a random walk, $X_t = X_{t-1} + \epsilon_t$ for $t > 0$ and $X_0 = 0$, where ϵ_t is Gaussian-distributed with zero mean and unit variance. A single realization (or time series) is highlighted in blue.

order moments of its joint probability distribution. This includes the mean $\mu_t = E[X_t]$ and the covariance $\gamma_{s,t} = E[(X_s - \mu_s)(X_t - \mu_t)]$ where $s, t = 1, 2, \cdots, N$. [1]

Modelling only the second-order properties of a time series is not necessarily a complete specification of its underlying stochastic process, except in one special case – if the process is Gaussian. A stochastic process is said to be Gaussian if and only if all of its finite-dimensional joint distributions are multi-variate Gaussian; i.e., the joint distributions of $(X_{t_1}, X_{t_2}, \cdots, X_{t_n})$ for all $n \in \{1, 2, \cdots, N\}$ and $t_1, t_2, \cdots, t_n \in \{1, 2, \cdots, N\}$ are multi-variate Gaussian. Gaussian processes are completely specified by their mean μ_t and covariance $\gamma_{s,t}$. However, if a process is non-Gaussian, the second-order properties are not sufficient to completely specify its probability law; information is lost. This information may be partially recovered by specifying the first- and/or second-dimensional joint distributions of the process, usually called the marginal distributions and the bivariate distributions, respectively. The marginal distributions, $p_t(x_t)$, are not necessarily equal for all t. Similarly, the bivariate distributions, $p_{t_1,t_2}(x_{t_1}, x_{t_2})$, are not necessarily equal for all t_1 and t_2. However, if a stochastic process is *strictly stationary*; i.e., all of its finite-dimensional distributions are invariant under translation, then of course the marginal and bivariate distributions are also invariant under translation. If only the moments up to order m are invariant under translation, then the process is said to be *weakly stationary* of order m. [2]

[1] $E[X]$ denotes the application of the expectation operator to a random variable X.
[2] Definitions related to stationarity often differ between texts, the notation of [12] has been adopted here.

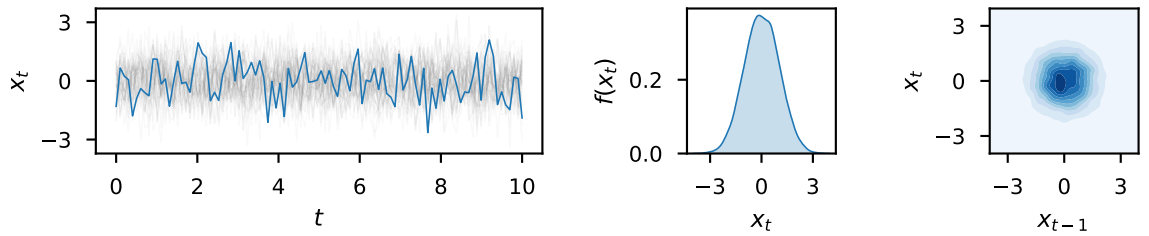


Fig. 2.5. Marginal and bivariate distributions of a stationary Gaussian stochastic process. (a) Realizations of a stationary Gaussian stochastic process X_t, with a single realization highlighted in blue. (b) Estimated marginal distribution, $p(x_t)$, of the process. (c) Estimated bivariate distribution, $p_{t,t-\tau}(x_t, x_{t-\tau})$, of the process. Darker colours indicate higher probability densities. A separate bivariate distribution exists for each lag value τ; only the distribution for $\tau = 1$ has been plotted.

Stationarity is an important requirement for performing almost any statistical inference about a set of data. For example, suppose a given time series $(x_1, x_2, \cdots, x_N)$ is generated by a non-stationary stochastic process. This is a single sample of an N-dimensional joint distribution. Not much can be inferred about a generating probability distribution from only a single sample! However, stationarity implies that the random variables $X_1, X_2, \cdots, X_N$ all share the same probability distribution. Thus, a single realization of a stationary time series of length N provides N samples of the marginal distribution of that time series; much can be inferred about the generating distribution in this case.

The concepts expressed in this section are basic requirements for working with statistical time series. In particular, the notion of stationarity is central to the contributions in Chapter 4.

2.3 Detection Theory

Relevant background material in classical detection theory will be developed in this section. The discussion closely follows the textbook by Van Trees, Bell, and Tian [13]. Detection theory provides a useful framework for dealing with certain problems in intrusion detection (e.g., see [14] and Chapter 5 of this thesis).

The detection problem can be modelled as consisting of four basic components: a source, a probability transition mechanism, an observation space, and a decision rule. An

experiment begins with the source, a device that generates an output out of M choices. The present discussion will be confined to the binary problem, $M = 2$. The choices are referred to as hypotheses, and denoted by H_0 and H_1. For example, in radar target detection, H_0 represents the absence of a target, and H_1 represents the presence of a target. In network intrusion detection, H_0 represents the absence of an intruder, and H_1 represents the presence of an intruder. The challenge here is that it is not known which hypothesis is true.

The probability transition mechanism is a device that knows which hypothesis is true. Depending on which hypothesis is true, the device generates a signal, represented by an N-dimensional vector $\mathbf{x} = (x_1, x_2, \cdots, x_N)$. The signal $\mathbf{x}$ inhabits the observation space, and is generated according to some probability law. The respective probability law under each hypothesis is denoted by $p_{\mathbf{X}|H_0}(\mathbf{x}|H_0)$ and $p_{\mathbf{X}|H_1}(\mathbf{x}|H_1)$. In the radar target detection example, the probability transition mechanism may corrupt the source signal with additive Gaussian noise. In the network intrusion detection example, the probability transition mechanism embeds the attack signal in benign background network traffic.

The problem is then to define a decision rule, which observes $\mathbf{x}$ in the observation space, and must decide which hypothesis was true. This process corresponds to assigning each point in the observation space to either H_0 or H_1. There are four possible outcomes to a given decision attempt:

1. H_0 true; choose H_0.
2. H_0 true; choose H_1.
3. H_1 true; choose H_1.
4. H_1 true; choose H_0.

Outcomes 1 and 3 correspond to correct decisions, and outcomes 2 and 4 correspond to errors. A decision criterion is required to attach relative importance to each of the four possible outcomes. Two criteria of interest are the Bayes Criterion and the Neyman-Pearson Criterion. It turns out that the processing on $\mathbf{x}$ required to make the best decision are the same for both of these criteria [13].

The Bayes Criterion is based on two assumptions. First, that either H_0 is true with probability $P_0 = \Pr\{H_0 \text{ true}\}$, or H_1 is true with probability $P_1 = \Pr\{H_1 \text{ true}\}$. The probabilities P_0 and P_1 are referred to as the *prior* probabilities. Second, that each of the four possible outcomes incurs a cost, denoted by C_{00}, C_{01}, C_{11}, and C_{10}, respectively. Here, the first subscript denotes the hypothesis that is chosen, and the second subscript denotes the hypothesis that was true. Denote the probability that a particular course of action is taken by

$$\Pr\{\text{choose } H_k \cap H_l \text{ true}\} = \Pr\{\text{choose } H_k | H_l \text{ true}\} P_l, \tag{2.1}$$

for $k = 0,1$ and $l = 0,1$. Then, the Bayes Criterion, also called the risk, is defined as the expected value of the cost incurred; i.e., the sum of the costs weighted by their probability of occurrence:

$$\begin{aligned} \mathscr{R} = {} & C_{00} P_0 \Pr\{\text{choose } H_0 | H_0 \text{ true}\} \\ & + C_{10} P_0 \Pr\{\text{choose } H_1 | H_0 \text{ true}\} \\ & + C_{11} P_1 \Pr\{\text{choose } H_1 | H_1 \text{ true}\} \\ & + C_{01} P_1 \Pr\{\text{choose } H_0 | H_1 \text{ true}\}. \end{aligned} \tag{2.2}$$

It can be shown that, provided $C_{10} > C_{00}$ and $C_{01} > C_{11}$, $\mathscr{R}$ is minimized by choosing H_1 if

$$\underbrace{\frac{p_{\mathbf{X}|H_1}(\mathbf{x}|H_1)}{p_{\mathbf{X}|H_0}(\mathbf{x}|H_0)}}_{\Lambda(\mathbf{x})} > \underbrace{\frac{P_0(C_{10} - C_{00})}{P_1(C_{01} - C_{11})}}_{\eta}, \tag{2.3}$$

and choosing H_0 otherwise [13]. It is interesting to note that all processing of the data $\mathbf{x}$ has been disentangled from the criterion-specific assumptions about the prior probabilities and cost assignments, and is entirely handled by the likelihood ratio $\Lambda(\mathbf{x})$. Regardless of the dimensionality of $\mathbf{x}$, the likelihood ratio, upon which decisions are based, is always one-dimensional. The threshold η, brings the assumptions about the prior probabilities and cost assignments into the decision rule.

A special case of interest is when $C_{10} = C_{01} = 1$ and $C_{00} = C_{11} = 0$, and the risk becomes the total probability of making an error

$$\begin{aligned} P_e = P_0 \Pr\{\text{choose } H_1 | H_0 \text{ true}\} \\ + P_1 \Pr\{\text{choose } H_0 | H_1 \text{ true}\}. \end{aligned} \tag{2.4}$$

The optimal decision rule is then to choose H_1 if

$$\frac{p_{\mathbf{X}|H_1}(\mathbf{x}|H_1)}{p_{\mathbf{X}|H_0}(\mathbf{x}|H_0)} > \frac{P_0}{P_1}, \tag{2.5}$$

and H_0 otherwise. Eq. (2.5) is known as the Minimum Probability of Error (MPE) detector. An alternative form can be used, since $P_1 > 0$ and $p_{\mathbf{X}|H_0}(\mathbf{x}|H_0) > 0$,

$$p_{\mathbf{X}|H_1}(\mathbf{x}|H_1) P_1 > p_{\mathbf{X}|H_0}(\mathbf{x}|H_0) P_0. \tag{2.6}$$

From Bayes rule

$$\Pr\{H_k|\mathbf{x}\} = \frac{p_{\mathbf{X}|H_k}(\mathbf{x}|H_k) P_k}{p_{\mathbf{X}}(\mathbf{x})}, \tag{2.7}$$

for $k = 0, 1$, where $p_{\mathbf{X}}(\mathbf{x})$ is a normalizing factor defined by

$$p_{\mathbf{X}}(\mathbf{x}) = p_{\mathbf{X}|H_0}(\mathbf{x}|H_0) P_0 + p_{\mathbf{X}|H_1}(\mathbf{x}|H_1) P_1. \tag{2.8}$$

The *posterior* probability $\Pr\{H_k|\mathbf{x}\}$ is the probability that H_k was true, given an observed $\mathbf{x}$. An equivalent detector to (2.5) is then to choose H_1 if

$$\Pr\{H_1|\mathbf{x}\} > \Pr\{H_0|\mathbf{x}\}, \tag{2.9}$$

and H_0 otherwise. Eq. (2.9) is known as the Maximum *A Posteriori* (MAP) detector. Furthermore, if $P_0 = P_1$, the optimal detector is to choose H_1 if

$$p_{\mathbf{X}|H_0}(\mathbf{x}|H_0) > p_{\mathbf{X}|H_1}(\mathbf{x}|H_1), \tag{2.10}$$

known as the Maximum Likelihood (ML) detector. Interestingly, all of the above-mentioned detectors may be described as a Likelihood Ratio Test (LRT); i.e., comparing the likelihood ratio $\Lambda(\mathbf{x})$ with some threshold η.

In realistic scenarios it is often difficult to assign appropriate costs or prior probabilities. Recall that the general goal in detection problems is to maximise the probability of detection

$$P_{\mathrm{D}} = \Pr\{\text{choose } H_1 | H_1 \text{ true}\}, \tag{2.11}$$

and to minimize the probability of false alarm

$$P_{\mathrm{FA}} = \Pr\{\text{choose } H_0 | H_1 \text{ true}\}. \tag{2.12}$$

However, these two conditions are generally conflicting. The Neyman-Pearson (NP) approach proposes to maximise P_{D} for a given $P_{\mathrm{FA}} = \alpha$. The NP detector; i.e., the optimal decision rule with respect to the NP Criterion may also be written as a LRT [13]:

$$\frac{p_{\mathbf{X}|H_1}(\mathbf{x}|H_1)}{p_{\mathbf{X}|H_0}(\mathbf{x}|H_0)} > \xi, \tag{2.13}$$

where the threshold ξ is chosen such that $P_{\mathrm{FA}} = \alpha$.

Just as the NP criterion is completely described in terms of P_{D} and P_{FA}, the Bayes risk criterion in (2.2) may also be characterized completely by P_{D} and P_{FA} as follows. Since the events {choose H_0} and {choose H_1} are complements of each other, it follows that

$$\begin{aligned} \Pr\{\text{choose } H_0 | H_0 \text{ true}\} &= 1 - \Pr\{\text{choose } H_1 | H_0 \text{ true}\} \\ &= 1 - P_{\mathrm{FA}}, \end{aligned} \tag{2.14a}$$

$$\begin{aligned} \Pr\{\text{choose } H_0 | H_1 \text{ true}\} &= 1 - \Pr\{\text{choose } H_1 | H_1 \text{ true}\} \\ &= 1 - P_{\mathrm{D}}. \end{aligned} \tag{2.14b}$$

Substituting (2.14a) and (2.14b) into (2.2) and simplifying, the risk can be written as

$$\mathscr{R} = P_0 C_{00} + P_1 C_{01} + P_0 (C_{10} - C_{00}) P_{\mathrm{FA}} - P_1 (C_{01} - C_{11}) P_{\mathrm{D}}. \tag{2.15}$$

Therefore, regardless of the chosen evaluation criterion, the accuracy of a detector can be completely specified by the probability of false alarm, P_{FA}, and the probability of detection, P_{D}. The curve obtained by observing P_{FA} versus P_{D}, while varying the threshold η, is known as the Receiver Operating Characteristic (ROC) curve, which provides a complete measure of a detector's accuracy.

Detection theory provides useful tools that are yet to be adopted into mainstream intrusion detection research [14]. To that end, the ideas introduced in this chapter are applied to the intrusion detection domain in Chapter 5.

2.4 Summary

This chapter introduced fundamental ideas and notions required by the chapters to follow. Basic concepts in intrusion detection, time series analysis, and detection theory were briefly touched upon. In particular, the notion of stationarity is central to time series anomaly detection in Chapter 4. Furthermore, detection theory provides a useful model for detecting network attacks in Chapter 5.

Chapter 3

Literature Review

This chapter reviews the related literature. The discussion is focused on Intrusion Detection Systems (IDSs) operating on the network level rather than on individual hosts, and on the anomaly-based approach rather than the misuse-based approach. This subset of anomaly-based network IDSs coincides with research in the field of network traffic anomaly detection, providing an alternative perspective also developed here. A wide variety of detection methods have been proposed in the literature, of which statistical and signal processing methods are emphasised in this chapter.

3.1 Historical Development of Intrusion Detection

The first documented design and development of automated intrusion detection was by Anderson [15] in the 1980s. Prior to this, intrusion detection was performed manually by printing out audit trail data (e.g., logs of user logins and file access) and manually searching for suspicious activity. Anderson proposed to summarise large volumes of audit trail records by statistical measures, in particular the mean and standard deviation of chosen features of the data, and to compare the measures against a set of absolute limits (a predefined number of standard deviations away from the mean).

This novel approach gained impetus through the work of Denning. Of particular import was Denning's general IDS model proposed in 1987 [10]. The model decomposes IDSs into

their interacting elements (similar to that discussed in Section 2.1). Denning also paved the way forward by posing research questions that are still relevant today, quoting from [10]:

- *Soundness of Approach:* Does the approach actually detect intrusions? Is it possible to distinguish anomalies related to intrusions from those related to other factors?
- *Completeness of Approach:* Does the approach detect most, if not all, intrusions, or is a significant proportion of intrusions undetectable by this method?
- *Timeliness of Approach:* Can we detect most intrusions before significant damage is done?
- *Choice of Metrics, Statistical Models, and Profiles:* What metrics, models, and profiles provide the best discriminating power? Which are cost-effective? What are the relationships between certain types of anomalies and different methods of intrusions? [10, p. 231]

Although the modern landscape presents new challenges, the above questions and limitations raised by Denning are still at the heart of anomaly-based IDSs today. Several key reviews have acknowledged that treatments of the above issues are still lacking [4–6, 16, 17]. In addition to these issues, two contemporary challenges have been recognized by researchers in the field [4, 16–18]. First, the enormous and growing throughput of network traffic places computational constraints on the development of IDSs. And, second, the increasing use of end-to-end encryption in user applications renders obsolete classical detection methods that rely on analyzing packet payload information.

Sperotto *et al.* [18] argued that detection based only on packet-header data, also called flow data, provides a means to overcome the above two challenges: high throughputs are managed since less data requires processing, and encryption is no longer inhibiting since the encrypted payload data are not required for detection. Although certain intrusions simply cannot be detected using only packet-header data, these flow-based intrusion detection

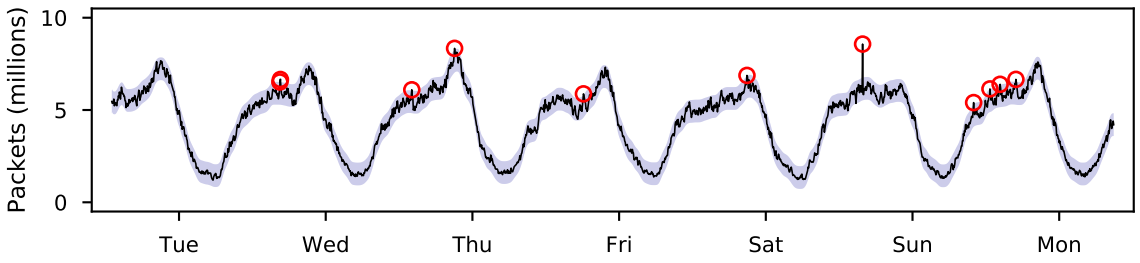


Fig. 3.1. A plot of Maxion-and-Feathers's method [19] applied to the Waikato ISPDSL2 dataset [20]. Where the original time series (black) exits the tolerance envelope (light blue) an anomaly is identified (red).

methods have proven capable of detecting a variety of malicious network-wide activities, including Distributed Denial-of-Service (DDoS) attacks, port scans, worms, and botnets [18].

This subset of flow-based network IDSs marks the point of intersection with another area of research: network traffic anomaly detection. Whereas flow-based IDSs began with security in mind and adapted to the network domain, network traffic anomaly detection began attempting to characterize networks and network traffic and later adapted to the security domain. The latter perspective will be described next.

3.2 Network Traffic Anomaly Detection

The pioneering work of Maxion and Feather [19] was among the first and most comprehensive treatments of network traffic anomaly detection. The emphasis was not to identify malicious activity, since cybercrime was not yet a serious concern, but rather to identify network faults and outages. Maxion and Feather proposed a method that constructed time series models of daily activity by smoothing time series data of previous days. Tolerance envelopes based on standard deviations allowed to categorize levels of normality. This method and variations thereof are considered standard approaches to time series anomaly detection. The approach was validated in a series of case studies where several distinct network faults and attacks were successfully detected. In order to demonstrate the method in action, it has been applied in Fig. 3.1 to the Waikato ISPDSL2 dataset [20]. The time series is the number of packets received within five minute intervals for approximately one week. The centre of

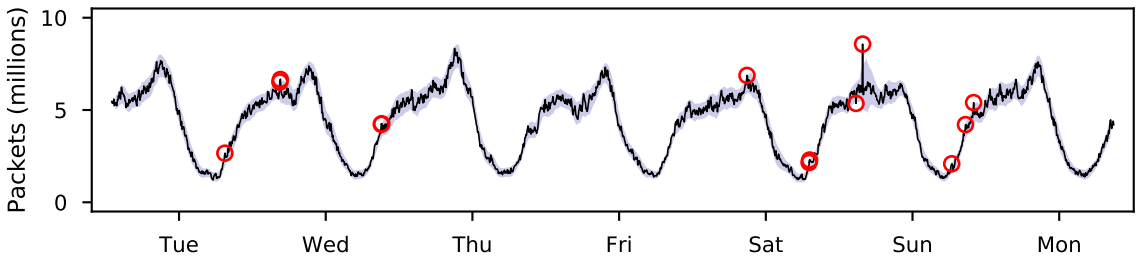


Fig. 3.2. A plot of Brutlag's method [21] applied to the Waikato ISPDSL2 dataset [20]. Where the original time series (black) exits the tolerance envelope (light blue) an anomaly is identified (red).

the tolerance envelope (light blue) is a smoothed version of the original time series (black). The upper and lower limits of the tolerance envelopes are plus and minus three and a half standard deviations of the residual (the residual being the smoothed time series subtracted from the original). Where the original time series leaves the tolerance envelope, anomalies have been identified (red). The method is an improvement over absolute limits (as in [15]) since it takes deterministic cycles into consideration.

Brutlag [21] extended methods based on simple exponential smoothing to employ Holt-Winters forecasting models (also known as triple exponential smoothing). Holt-Winters models take into account deterministic trends and cycles. The method operates by predicting one sample point ahead and computing the residual (prediction error). An adaptive envelope of acceptable residual values is constructed by applying a seasonal forecasting model to the residual time series. This is the primary improvement to Maxion-and-Feather's method. The implementation of the model was made public and parameter selection was guided in detail – two further reasons for the work's acclaim. The method has been applied in Fig. 3.2 to the Waikato ISPDSL2 dataset [20]. The difference from Maxion-and-Feather's approach, due to the adaptive tolerance envelope used here, is particularly noticeable where the time series steadily rises and decays. Since there is typically low variability in the time series at these parts, the tolerance envelope is narrower here than in Fig. 3.1. The adaptive tolerance envelope allows to better detect anomalies that occur at these low variable parts of the time series, as is demonstrated in the figure. Brutlag also attempted to reduce false positives by only addressing so-called *failures*, where k anomalies occur within a window of fixed length.

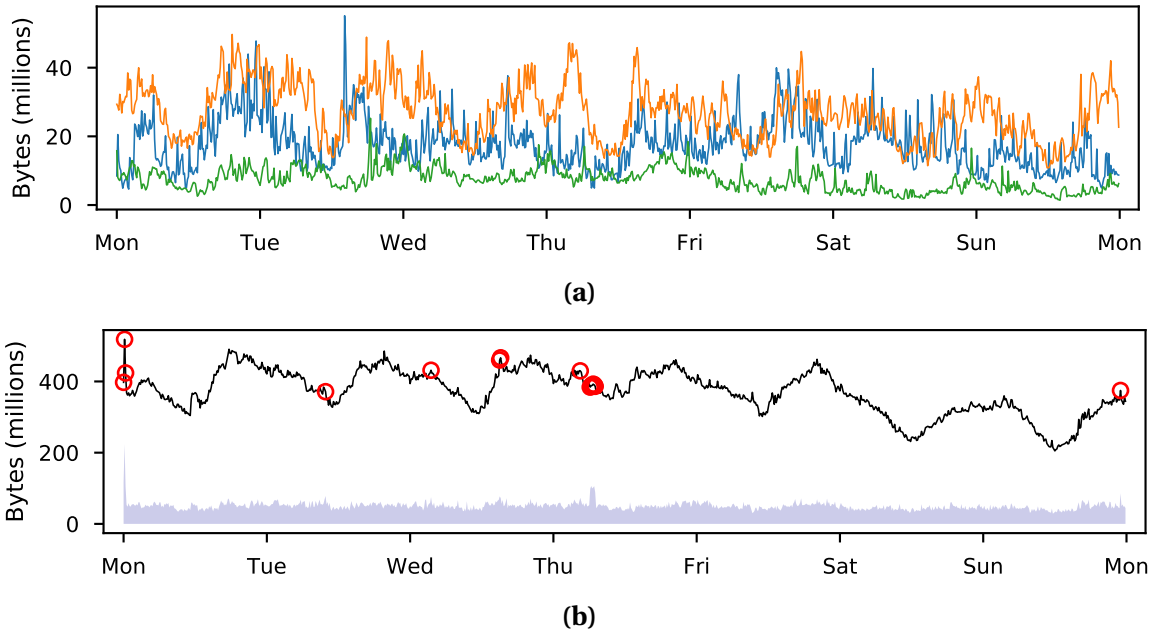


Fig. 3.3. A plot of Lakhina's method [24–26] applied to the Abilene dataset [27]. (a) 3 out of a collection of 121 dependent time series. (b) The L_2 norm across all 121 time series at given points in time (black) and the PCA reconstruction error (light blue) with the 15 highest reconstruction errors labelled as anomalies (red).

Barford *et al.* [22, 23] proposed an alternative method based on wavelet analysis. The time series was decomposed via wavelet analysis into three frequency bands: low, medium, and high. The low-frequency part captures regular trends in the traffic; i.e., daily and weekly cycles. The medium-frequency part captures daily variations away from low-frequency trends. Lastly, the high-frequency part captures noise and erratic bursts. Since anomalies are typically transient, they tend not to affect the low-frequency part. Therefore, a deviation score was defined in terms of the medium- and high- frequency parts. Barford *et al.* compared their method with Brutlag's method on a manually labelled data set, and found that they produced similar results.

Lakhina *et al.* [24–26], in a sequence of three influential papers, extended network traffic anomaly detection to a *network-wide* view. Whereas, previously, multiple time series each corresponding to distinct network links were treated independently, this work constructed a normal model from the inter-dependence between time series (corresponding to distinct network links). This form of dependence has been considered the *spatial* equivalent to

temporal dependence as in the previously described methods. A principal component analysis (PCA) was performed on the collection of time series. The principal components were ordered in terms of their proportion of total energy captured. Projecting the time series onto the first few principle components reconstructs the regular deterministic trends (daily and weekly cycles). Projecting the time series onto the next few principle components reconstructs daily variations and spikes. Projecting the time series onto the last few principle components reconstructs noise. The key finding was that out of hundreds of time series, the subspace spanned by the first 5 to 10 principal components allowed to reconstruct each time series with high accuracy. Furthermore, the reconstruction error was demonstrated as a useful metric for detecting anomalies.

The method has been demonstrated in Fig. 3.3 on the Abliene dataset [27], a collection of one hundred and twenty one (121) dependent time series were sampled at ten minute intervals for a total of one week. First, the time series are reconstructed using the first seven principal components, which yields a reconstruction error of 4% error. The anomalies with the 15 highest reconstruction errors have been plotted, showing that the method detects anomalies that would not be detected by, e.g., thresholding the L_2 norm across all time series at each point in time.

The method of [25] was further extended in [26] to consider packet-header features (IP addresses and ports) in addition to simple byte counts. It was recognized that the *dispersion* of feature distributions is useful for characterizing well-known anomalies. For example, port scans send packets to the same destination IP from many different source IPs over different ports. This dispersion of observed source IPs and ports together with the concentration of many packets sent to the same destination IP can therefore be used to detect port scans. Sample entropy was proposed as one possible measure of dispersion, though the authors suggested that other measures could also have been used. Interestingly, it was found that the entropy method and the prior volume method detected barely overlapping sets of anomalies; the two methods complemented each other.

The methods described in this section have since been extended in many different directions. This thesis focuses on one particular direction: detection with high-resolution

time series data, sampled at millisecond intervals. All of the above methods operate on time series sampled at intervals no less than five minutes long. Since decisions can only be made as frequently as new samples are observed, anomalies (correspondingly attacks) can only be detected at five minute intervals. In reality, it is desirable to operate at much finer resolutions. The faster an attack is detected the less damage it can deal. Time series sampled at millisecond intervals present new challenges that require approaches distinct from those presented in this section. These challenges and approaches will be discussed next.

3.3 Detection With High-Resolution Time Series

It is useful to think of network traffic time series as being composed of two parts: deterministic and stochastic. The deterministic part closely follows the so-called *diurnal cycle*; i.e., high activity during the day peaking around noon, followed by low activity during the night. On the other hand, the characteristics of the stochastic part depend on the length of the sample interval.

The methods discussed in the previous section operate on time series sampled at intervals of no less than five minutes long. It is desirable to operate with much shorter sample intervals. However, as the sample interval is shortened to meet this requirement, the properties of the time series change. With five minute intervals, the deterministic component of the time series is the main component required to model the time series. Models that decompose time series into smooth deterministic functions of time superposed with Gaussian noise have been comprehensively treated in the literature, e.g., those described in the previous section [19, 21, 23–26]. With one millisecond intervals, the deterministic component is approximately constant, and superposed with *non-Gaussian* noise. To model these time series requires specifying a time-varying non-Gaussian distribution. Two key approaches to this problem will be discussed next.

Scherrer *et al.* [7] proposed a model of the time series of packet counts consisting of two parts. The marginal distribution of the time series was modelled by a gamma distribution, and the autocovariance function was modelled by that of an Autoregressive Fractionally

Integrated Moving Average (ARFIMA) process. The model was demonstrated to provide a reasonable approximation over a wide range of sample interval lengths and for a variety of different networks, including wide area networks, local area networks, and backbone networks. Models were validated graphically by comparing an empirical histogram to the theoretical probability density function, and numerically by comparing the chi-squared test statistic of the gamma distribution to that of the exponential, log-normal, and chi-squared distributions. The gamma distribution provided the best fit over the majority of sample interval lengths. The gamma distribution also fit the data under the effects of a DoS attack, though the parameters shifted. This observation served as the basis for detecting DoS attacks. Attacks were detected by thresholding the mean quadratic distance between the parameters of the model fitted to a reference window and a test window. The assumption is that the occurrence of the attack in the test window would be detected as a shift in parameters of the model. The method was evaluated on real recorded network traffic of a simulated DoS user datagram protocol (UDP) flood attack with varying intensity, duration, packet size, and send rate. Attacks were generated using the software iPerf [28] and Trinoo [29] on a low activity network. The method was evaluated via a Receiver Operating Characteristic (ROC) analysis, though the method was not compared to any others. This is perhaps due to this work being among the first to focus on detection with high-resolution time series.

The above work was extended by Simmross-Wattenberg *et al.* [8]. First, an alpha-stable distribution was proposed to replace the gamma distribution model. It was demonstrated that the alpha-stable distribution better models the heavy tails present in network traffic time series, thus leading to more accurate detection. Second, the generalized likelihood ratio (GLRT) [13] was computed to measure the deviation between reference and test data rather than the mean quadratic distance. Third, assuming periodicity (with a weekly period) in the marginal distributions, the reference models were constructed from previous week data. For example, to perform detection at 1pm on Monday today, make reference to 1pm on Monday last week, the week before that, and so on. Fourth, a model of the attack was assumed by simulating the attack and superposing the results on the reference data. Goodness-of-fit of the alpha-stable and gamma distribution were compared via the corrected Kolmogorov-

Smirnoff test statistic. The correction is required due to the bias introduced into the statistic when the parameters of the probability density function are estimated from data (the true parameters of the distribution are unknown). The method was evaluated via a ROC analysis and demonstrated superior detection performance to [7]. Three critical points stand out of this work. First, their work did not investigate the effects of each of the above four extensions independently. Second, the method of [7] was re-implemented with sample interval lengths far greater than those originally designed for. The authors acknowledged this point as being due to technical limitations; they did not have data available with appropriate sample intervals. Third, an attack model was assumed although the validity of this assumption was not investigated. It is reasonable to expect that an assumed attack model would often be mismatched to real attacks. The effects of assuming a mismatched attack model are yet to be investigated.

3.4 Summary

In this chapter a brief historical account of the literature was developed from two alternative perspectives. Several seminal network traffic anomaly detection (also known as flow-based intrusion detection) methods were discussed, setting the stage for the focus of this thesis – detection with high-resolution time series. Whereas the previously discussed methods operate with sample intervals no less than five minutes long, it is desirable to detect attacks much more frequently. However, performing detection with millisecond sample intervals presents new challenges. Two key approaches were detailed ([7] and [8]) that attempt to address these challenges, thus forming the basis of this research.

Chapter 4

Analysing Network Traffic Anomaly Detection Algorithms

This chapter proposes a framework for analysing time series anomaly detection algorithms. Each algorithm is decomposed into three constituent components, namely: windows, representations, and deviations. Two state-of-the-art algorithms ([7] and [8]) that operate on high-resolution time series are analysed in terms of the framework, allowing to highlight their differences and suggest previously unexplored configurations of their components.

4.1 Deconstructing Detection Algorithms

In this section, it is proposed to decompose anomaly-based network intrusion detection algorithms into three constituent components: windows, representations, and deviations. This deconstruction allows to compare different detection algorithms within a consistent framework. The central object processed by each of the three components is a time series; i.e., a set of observations made sequentially in time (see Section 2.2 for an introduction to the analysis of time series). A given time series is processed by each of the components as follows:

1. *Windows:* Real-world time series are typically non-stationary; i.e., their statistical properties evolve over time. However, stationarity is an important requirement for

many statistical and signal processing techniques. A classical approach for dealing with a non-stationary time series is to construct a sliding window such that each sub-series observed within successive positions of the window is stationary.

2. *Representations:* Time series may be represented in different ways. A representation attempts to capture the underlying characteristics of a time series in a small number of parameters. Operations may then be performed on the representation rather than the original time series. Two notable examples are the *time-domain* and *frequency-domain* representations. Time series adopt a time-domain representation by default; i.e., observations made at distinct points in time are stored in separate components of a vector. The Fourier transform of a time series gives its frequency-domain representation; i.e., each component of a vector represents a distinct frequency present in the time series. Since the time series of interest here are noisy and lack deterministic components, statistical representations are deemed appropriate. That is, the probability distribution of the time series is modelled by a parametrized family of distributions. Different distributions emphasise different characteristics of a time series, therefore it must be ensured that the choice of distribution is appropriate for a given time series.

3. *Deviations:* The deviation (or dissimilarity) between two time series is measured in order to detect anomalies. Measuring deviations in the time-domain is not always appropriate. Since different representations emphasise different underlying characteristics of a time series, pairing appropriate deviation measures with appropriate representations allows to pinpoint how exactly the dissimilarity between two time series is to be measured for a given application. For example, in engineering applications it is often useful to compare time series in terms of their frequency-domain representations.

A detector (or detection algorithm) is a combination of the above components into a single consistent system. A typical detector operates as follows: two or more windows are constructed on the original time series, usually a *test* window and one or more *reference* windows; the sub-series observed within the windows are transformed into appropriately

chosen representation(s); deviations are calculated between the represented reference and test sub-series; finally, decisions are made according to the magnitude of the deviations. Two different classes of detectors will be contrasted, local anomaly detectors and periodic anomaly detectors, that detect distinct classes of anomalies.

Having described each component of a detection algorithm at a high-level, specific components appropriate for operating with the time series of interest here will be detailed next. Specifically, sliding windows to deal with non-stationarity, marginal distribution models to deal with non-Gaussianity, and multi-aggregated representations to accommodate self-similarity. Thereafter, local and periodic anomaly detection algorithms will be contrasted.

4.1.1 Sliding Windows

The majority of statistical and signal processing techniques operate under the assumption that a given time series is stationary. However, time series found in the real-world are often non-stationary. Provided a time series is at least locally stationary, i.e., its statistical properties vary slowly with time, a standard approach is to construct a sliding window of fixed length such that the sub-series observed within successive positions of the window are stationary. More formally, consider a time series of N observations, $\mathbf{x} = (x_1, x_2, \cdots, x_N)$. Supposing that $\mathbf{x}$ is locally stationary on intervals of fixed length L, construct a sliding window of length L. For each position of the window $m = 0, 1, \cdots, M-1$ (where M is the total number of window positions), the observed sub-series $(x_{m+1}, x_{m+2}, \cdots, x_{m+L})$ can be assumed to be stationary. For an in-depth statistical treatment of locally stationary processes see [30].

To further demonstrate the concept of local stationarity, the results of a brief numerical experiment are plotted in Fig. 4.1. A zero-mean uncorrelated Gaussian time series with changing variance was generated, shown in Fig. 4.1a. In Fig. 4.1b, its true variance, following a sinusoidal function of time, is plotted (black dashed line), together with estimated variance functions. The variance functions were estimated using the standard variance estimator within sliding windows of lengths $L \in \{50, 10\,000, 30\,000\}$ sample points. The experiment also

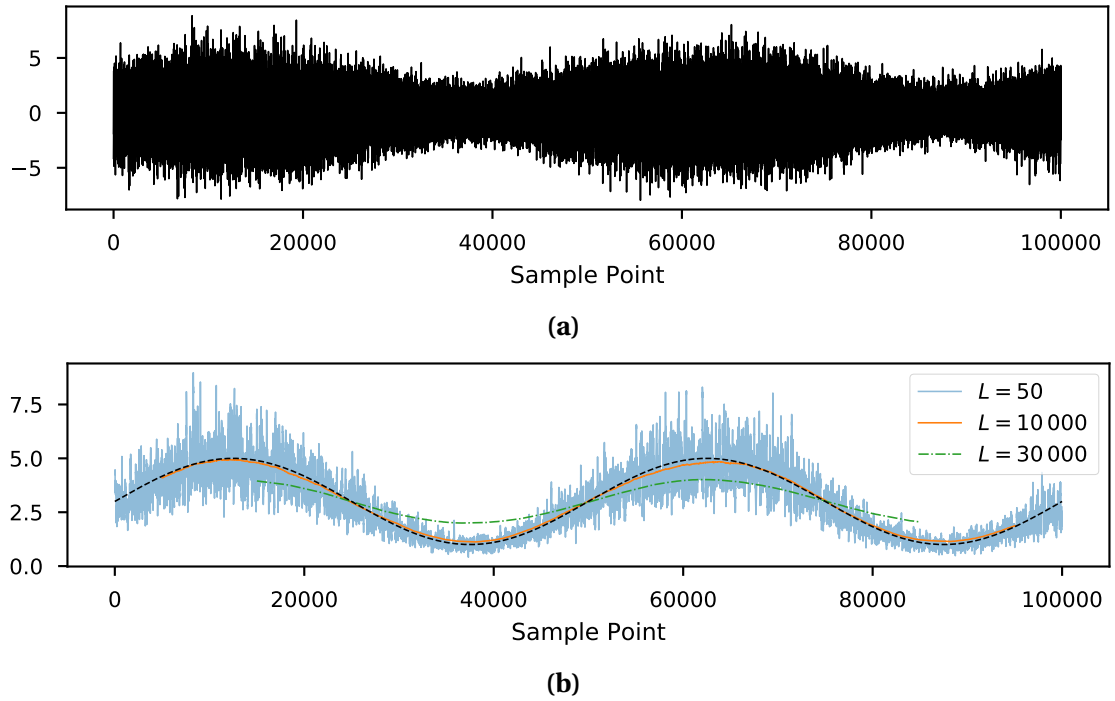


Fig. 4.1. Locally estimating the changing variance in a time series. (a) A zero-mean uncorrelated Gaussian time series with variance changing as a sinusoidal function of time. (b) The true variance (black dashed line) and the variance estimated locally within windows of lengths $L \in \{50, 10\,000, 30\,000\}$ sample points.

demonstrates the difficulty in choosing L (if the true L is unknown) so that the sub-series are both sufficiently long yet also stationary. With $L = 50$, the sub-series are stationary, however, they are too short leading to a noisy result. With $L = 30\,000$, the sub-series are sufficiently long, however, they are not stationary leading to a smooth but biased result. A good balance is achieved at $L = 10\,000$ for this experiment.

4.1.2 Marginal Distribution Models

Network traffic time series sampled at intervals in the order of minutes appear to be Gaussian-distributed with deterministic mean. However, as the sample interval shortens, the sampled time series becomes increasingly non-Gaussian. This notion is empirically demonstrated via the plots in Fig. 4.2. The same network traffic data is sampled at widening intervals, starting at one millisecond and increasing to one second. Histograms of the marginal distributions of the sampled time series are plotted together with fitted Gaussian

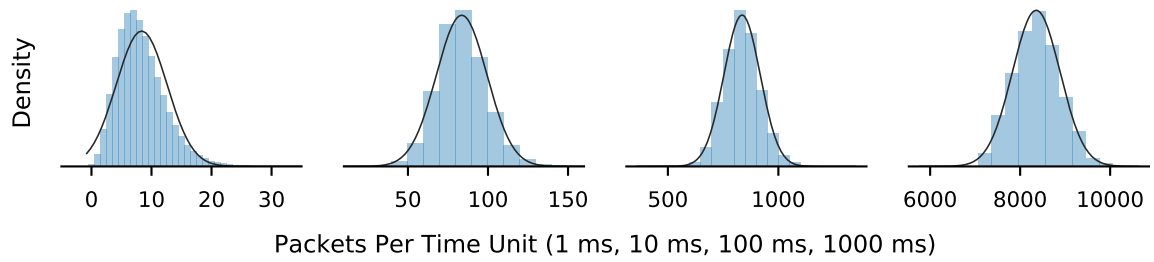


Fig. 4.2. Demonstrating the intrinsic non-Gaussianity in network traffic time series. Histograms of the marginal distribution of time series from the Auckland7 dataset [20] sampled with four different interval lengths, along with fitted Gaussian probability density functions. The figure demonstrates how the marginal distribution tends from non-Gaussian with shorter sample intervals (1 ms) to Gaussian with wider sample intervals (1000 ms).

probability density functions. It is demonstrated that for shorter sample intervals, the Gaussian fit is poorer since the histogram is skewed to the left. On the other hand, the Gaussian curve provides a reasonable fit to the histogram for wider sample intervals.

To account for this intrinsic non-Gaussianity, it has been proposed in the literature to model the marginal distribution of network traffic time series in addition to its second-order properties. It has been empirically demonstrated that a Gamma distribution provides a reasonable approximation to the true marginal distribution of the packet counts time series for a variety of network configurations. The gamma marginal distribution model has been implemented and demonstrated to effectively detect network attacks in [7, 31, 32]. Formally, it is assumed that each observation x_t of the time series is a realization of a random variable X_t with gamma probability density function

$$p(x;\alpha,\beta) = \frac{1}{\beta\Gamma(\alpha)}\left(\frac{x}{\beta}\right)^{\alpha-1}\exp\left(-\frac{x}{\beta}\right), \tag{4.1}$$

for $x,\alpha,\beta > 0$. Here, $\Gamma(\cdot)$ denotes the gamma function, and α and β denote the shape and scale parameters of the gamma distribution. The scale parameter β, as its name suggests, simply scales the value of the random variable X_t. By varying the shape parameter α, the gamma distribution provides a smooth evolution from an exponential distribution ($\alpha < 1$) to a Gaussian distribution ($\alpha \to \infty$). Since network traffic time series tend towards a Gaussian distribution as the sample interval widens, the gamma distribution provides a reasonably

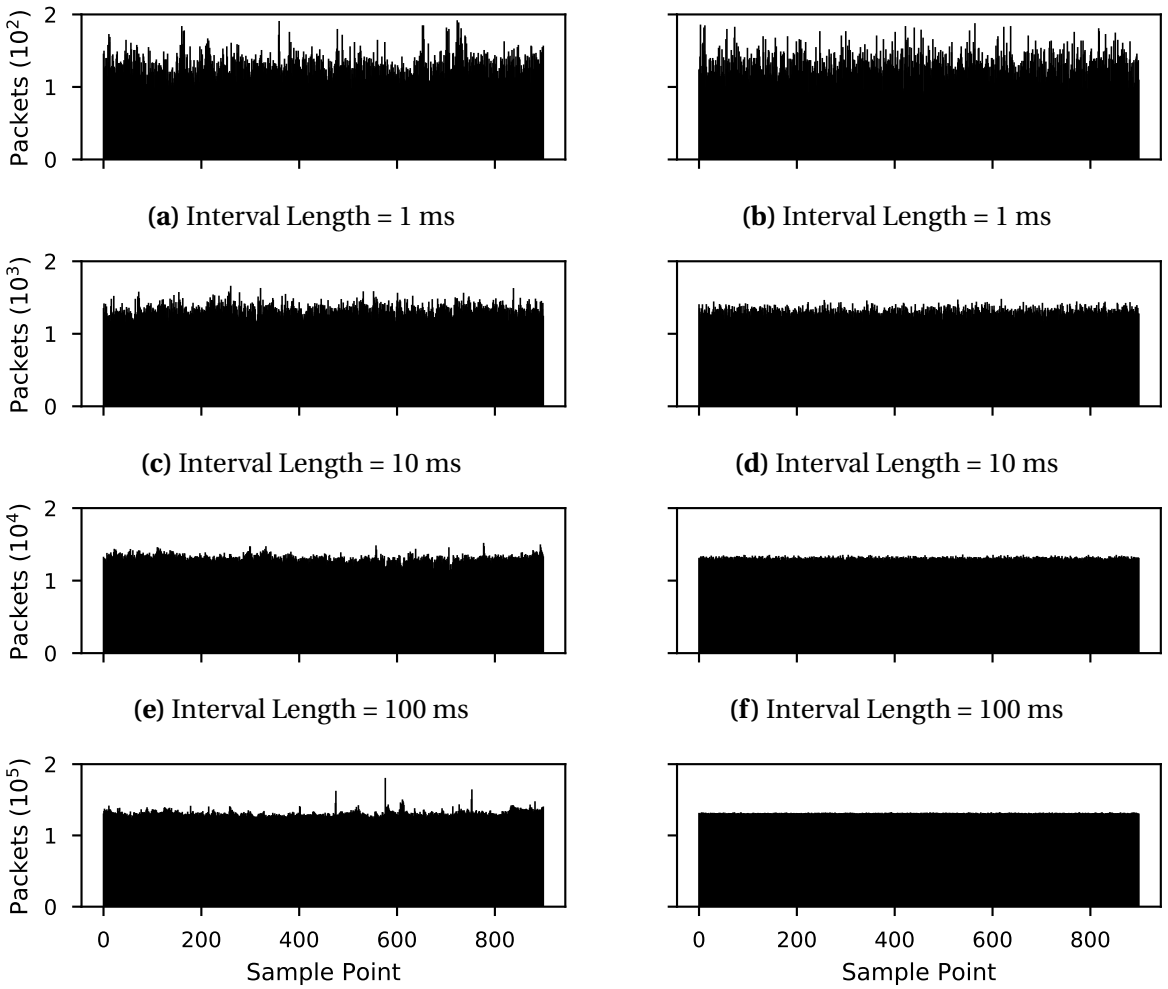
accurate model over a wide range of sample interval lengths. This property will be further utilized in the next section.

4.1.3 Multi-Aggregated Representations

Typical approaches to modelling time series require the selection of an appropriately sized sample interval such that a phenomenon of interest best exhibits itself. However, in network traffic time series, it is not a single sample interval length, but rather a range, that is of interest. This is due to the property known as *self-similarity*. The intuitive idea is that self-similar time series appear similar when viewed over intervals of differing lengths. Time series that do not exhibit this property, when viewed over increasing lengths, tend very quickly towards a constant deterministic function of time. Fig. 4.3 (adapted from the seminal work of Leland *et al.* [33]) demonstrates this concept by comparing a self-similar and non-self-similar time series sampled at widening intervals.

A more formal definition is as follows. Given the time series $\mathbf{x}$, define the k-aggregated time series $\mathbf{x}^{(k)}$ to be the addition of all observations x_t within non-overlapping intervals of length k $(= 1, 2, \cdots)$. In other words, the n^{th} observation of the k-aggregated time series is defined as $x_n^{(k)} = x_{k(n-1)+1} + x_{k(n-1)+2} + \cdots + x_{k(n-1)+k}$. A process is said to be self-similar if for all positive integer k, the original time series $\mathbf{x}$ is equivalent to the k-aggregated time series $\mathbf{x}^{(k)}$ in distribution; i.e., all of their finite-dimensional joint distributions are equivalent. The rows of Fig. 4.3 are in fact plots of the original time series $\mathbf{x}$ sampled at one millisecond intervals followed by three k-aggregated time series $\mathbf{x}^{(k)}$ for $k \in \{10, 100, 1000\}$. The process of aggregating the time series is the same as sampling the underlying data within wider intervals. Since $\mathbf{x}$ was sampled within one millisecond intervals, $\mathbf{x}^{(k)}$ corresponds to the same data sampled within k millisecond intervals. The concept illustrated in the figure is due to the property that the variance of a self-similar process decays as a power law of k, whereas that of a non-self-similar process decays exponentially fast in k [33].

Network traffic time series are self-similar, thus their covariance does not decay exponentially fast in k [33]. It has therefore been proposed in the literature to model network traffic time series at multiple aggregation levels k (correspondingly, at multiple sample inter-



(a) Interval Length = 1 ms

(b) Interval Length = 1 ms

(c) Interval Length = 10 ms

(d) Interval Length = 10 ms

(e) Interval Length = 100 ms

(f) Interval Length = 100 ms

(g) Interval Length = 1000 ms

(h) Interval Length = 1000 ms

Fig. 4.3. A brief numerical experiment adapted from [33] visualising the effect of self-similarity in a time series. (a), (c), (e), (g) Number of packets received per interval, sampled from the Auckland 7 dataset [20] with four different interval lengths. (b), (d), (f), (h) Time series synthetically generated by fitting a gamma distribution to the original time series in (a) and aggregating. Both time series appear similar at 1 millisecond intervals. However, as the sample interval widens, the synthetic time series converges to a constant value much faster than the real time series. This effect is due to the self-similarity present in the original time series.

val lengths). Given a time series $\mathbf{x}$, the k-aggregated time series are constructed for some pre-defined values of k. Thereafter, the collection of aggregated time series $\mathbf{x}^{(k)}$ is modelled together. This approach has been combined with the gamma marginal distribution model in [7] to improve Denial-of-Service (DoS) detection accuracy.

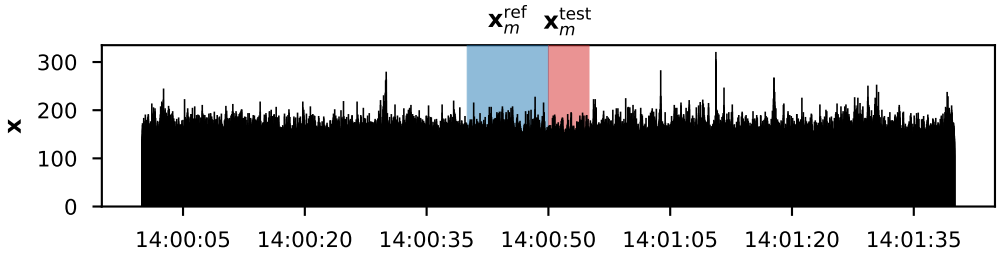


Fig. 4.4. An example demonstrating the windowing process of a local anomaly detector on the MAWI dataset [34]. The reference sub-series $\mathbf{x}_m^{\text{ref}}$ of length N_{ref} (= 10 seconds), is observed within the blue window starting at 14:00:40. The test sub-series $\mathbf{x}_m^{\text{test}}$ of length N_{test} (= 5 seconds) is observed within the red window starting at 14:00:50.

4.1.4 Local Versus Periodic Anomaly Detectors

Anomalies can be thought of as observations where some model of normality – what is regarded as normal – breaks down. In this section, it is proposed that Scherrer *et al.* [7] and Simmross-Wattenberg *et al.* [8] detect two distinct classes of anomalies, each being an instance where a different assumption about stationarity breaks down, more specifically, local stationarity and periodic stationarity assumptions. For an introductory discussion to the concepts of local stationartiy and periodic stationarity, the reader is referred to Section 2.2.

Local anomaly detectors, as the name suggests, attempt to identify where an assumption of local stationarity breaks down. This class of algorithms operates as follows. It is assumed that a time series $\mathbf{x}$ is locally stationary; i.e., stationary on intervals of length L. Therefore, two contiguous (touching each other at the borders) and synchronous (moving together) sliding windows are constructed: the reference window of length N_{ref} and the test window of length N_{test} (where $N_{\text{ref}} + N_{\text{test}} = L$). At each position of the window ($m = 0, 1, \cdots, M-1$), the reference sub-series is given by $\mathbf{x}_m^{\text{ref}} = (x_{m+1}, x_{m+2}, \cdots, x_{m+N_{\text{ref}}})$ and the test sub-series is given by $\mathbf{x}_m^{\text{test}} = (x_{m+N_{\text{ref}}+1}, x_{m+N_{\text{ref}}+2}, \cdots, x_{m+N_{\text{ref}}+N_{\text{test}}})$. This formulation has been visualized in Fig. 4.4 to further clarify the idea.

Due to the local stationarity assumption, both sub-series are stationary; since they are contiguous, they share the same statistical properties. Anomalies are detected by identifying cases where the local stationarity assumption is broken; i.e., where $\mathbf{x}_m^{\text{ref}}$ and $\mathbf{x}_m^{\text{test}}$ do not

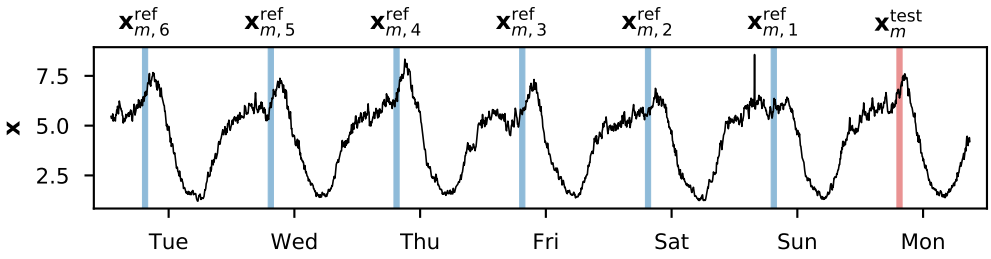


Fig. 4.5. An example demonstrating the windowing process of a periodic anomaly detector on the Waikato ISPDSL2 dataset [20]. For a given test sub-series $\mathbf{x}^{\text{test}}_m$ of length N_{test} (= 1 hour) starting on Sunday at 19:00, J(= 6) reference sub-series are observed, $\{\mathbf{x}^{\text{ref}}_{m,1}, \cdots, \mathbf{x}^{\text{ref}}_{m,J}\}$ each of length N_{ref} (= 1 hour) shifted integer multiples of the period P (= 1 day) before the test sub-series.

share the same statistical properties. Therefore, detection is performed by computing the deviation between appropriately chosen representations of the sub-series $\mathbf{x}^{\text{ref}}_m$ and $\mathbf{x}^{\text{test}}_m$, and comparing with a pre-defined threshold value.

On the other hand, periodic detectors attempt to identify where an assumption of periodic stationarity breaks down. In this case, it is assumed that a given time series $\mathbf{x}$ is both locally stationary on intervals of length L and periodically stationary with period P. A collection of synchronous sliding windows are constructed, each of length L: a single test window, and multiple reference windows shifted from the test window by integer multiples of P. For the m^{th} position of the windows, the test sub-series is given by $\mathbf{x}^{\text{test}}_m = (x_{m+1}, x_{m+2}, \cdots, x_{m+L})^{\text{T}}$, and the corresponding collection of reference sub-series are given by $\mathbf{x}^{\text{ref}}_{m,j} = (x_{m-jP+1}, x_{m-jP+2}, \cdots, x_{m-jP+L})^{\text{T}}$ $(j = 1, 2, \cdots, J)$. The number of reference sub-series J depends on the number of periods of data available. An example demonstrating the construction of reference and test sub-series for a periodic anomaly detector has been visualized in Fig. 4.5.

For each successive m, due to the assumed local stationarity, each sub-series $\mathbf{x}^{\text{test}}_m$ and $\mathbf{x}^{\text{ref}}_{m,j}$ is stationary, and due to the assumed periodic stationarity, the sub-series $\mathbf{x}^{\text{test}}_m$ and $\mathbf{x}^{\text{ref}}_{m,j}$ share the same statistical properties. The algorithm identifies anomalies where $\mathbf{x}^{\text{test}}_m$ and $\mathbf{x}^{\text{ref}}_{m,j}$ do not share the same statistical properties. Thus, a deviation is calculated between appropriately chosen representations of the test sub-series and each reference sub-series. The collection of J deviation values (one for each reference sub-series) must be combined in order to make a final decision as to whether periodic stationarity has broken down.

4.2 Constructing Detection Algorithms

In the previous sections, certain elements of detection algorithms were described independently. Two state-of-the-art algorithms (those of Scherrer *et al.* [7] and Simmross-Wattenberg *et al.* [8]) will now be analysed in terms of the described elements, thus placed within a consistent framework. The analysis allows to identify exactly how and where the algorithms differ and to propose previously unexplored configurations of their components.

4.2.1 Local Detection Algorithm (Scherrer *et al.*)

As mentioned in Section 4.1.4, Scherrer *et al.*'s algorithm [7] can be considered a local anomaly detection algorithm. Recall that local anomaly detectors attempt to identify where an assumption of local stationarity breaks down. Therefore, two contiguous and synchronous sliding windows are constructed: the reference window and the test window. At each position of the windows ($m = 0, 1, \cdots, M-1$), the reference sub-series is denoted by $\mathbf{x}_m^{\text{ref}}$ and the test sub-series is denoted by $\mathbf{x}_m^{\text{test}}$. Detection is performed as follows for each successive position, m, of the windows. Due to the self-similarity exhibited by network traffic time series, the sub-series are represented at multiple aggregation levels; i.e., $\mathbf{x}_m^{\text{ref}}$ and $\mathbf{x}_m^{\text{test}}$ are each aggregated K times with dyadically increasing interval lengths, giving $\mathbf{x}_{m,k}^{\text{ref}}$ and $\mathbf{x}_{m,k}^{\text{test}}$ for $k = 0, 1, \cdots, K$.

Next, the marginal distribution of each $\mathbf{x}_{m,k}^{\text{ref}}$ and $\mathbf{x}_{m,k}^{\text{test}}$ is approximated by a Gamma distribution. The model is fitted via maximum likelihood estimation, giving the estimated gamma shape and scale parameters for each sub-series, $\left(\alpha_{m,k}^{\text{ref}}, \beta_{m,k}^{\text{ref}}\right)$ and $\left(\alpha_{m,k}^{\text{test}}, \beta_{m,k}^{\text{test}}\right)$. The parameters corresponding to each sub-series can be considered as a vector, where each component corresponds to a different aggregation level k; i.e.,

$$\boldsymbol{\alpha}_m^{\text{ref}} = \left(\alpha_{m,0}^{\text{ref}}, \alpha_{m,1}^{\text{ref}}, \cdots, \alpha_{m,K}^{\text{ref}}\right), \tag{4.2a}$$

$$\boldsymbol{\alpha}_m^{\text{test}} = \left(\alpha_{m,0}^{\text{test}}, \alpha_{m,1}^{\text{test}}, \cdots, \alpha_{m,K}^{\text{test}}\right), \tag{4.2b}$$

$$\boldsymbol{\beta}_m^{\text{ref}} = \left(\beta_{m,0}^{\text{ref}}, \beta_{m,1}^{\text{ref}}, \cdots, \beta_{m,K}^{\text{ref}}\right), \tag{4.2c}$$

$$\boldsymbol{\beta}_m^{\text{test}} = \left(\beta_{m,0}^{\text{test}}, \beta_{m,1}^{\text{test}}, \cdots, \beta_{m,K}^{\text{test}}\right). \tag{4.2d}$$

The deviation between the reference and test sub-series is then calculated as the L_2 distance between corresponding parameter vectors. A separate deviation is calculated for the shape and scale parameters[1]

$$d_m^{(\alpha)} = \left\| \boldsymbol{\alpha}_m^{\text{ref}} - \boldsymbol{\alpha}_m^{\text{test}} \right\|_2, \tag{4.3a}$$

$$d_m^{(\beta)} = \left\| \boldsymbol{\beta}_m^{\text{ref}} - \boldsymbol{\beta}_m^{\text{test}} \right\|_2. \tag{4.3b}$$

The algorithm takes into consideration the local stationarity using sliding windows, non-Gaussianity using the gamma distribution model, and self-similarity using the multi-aggregated representation.

4.2.2 Periodic Detection Algorithm (Simmross-Wattenberg *et al.*)

In contrast to the previously described algorithm, Simmross-Wattenberg et al.'s [8] can be considered a periodic anomaly detection algorithm (as argued in Section 4.1.4). Therefore, a collection of synchronous sliding windows are constructed, each of length L: a single test window, and multiple reference windows shifted from the test window by integer multiples of P. For a given window position m, we have a test sub-series $\mathbf{x}_m^{\text{test}}$ and J corresponding reference sub-series $\mathbf{x}_{m,j}^{\text{ref},H_0}$. The superscript H_0 has been included here to indicate that the reference sub-series is assumed to be benign. Each reference benign sub-series $\mathbf{x}_{m,j}^{\text{ref},H_0}$ is superposed with an attack time series to give a reference attack sub-series $\mathbf{x}_{m,j}^{\text{ref},H_1}$. Each reference sub-series (benign and attack) is modelled by an alpha-stable distribution giving the alpha-stable parameter vectors $\boldsymbol{\theta}_{m,j}^{\text{ref},H_0}$ for the benign reference sub-series and $\boldsymbol{\theta}_{m,j}^{\text{ref},H_1}$ for the attack reference sub-series.

[1] $\|\mathbf{x}\|_2$ denotes the L_2 norm of a vector $\mathbf{x}$, thus $\|\mathbf{x}-\mathbf{y}\|_2$ denotes the L_2 distance between vectors $\mathbf{x}$ and $\mathbf{y}$ of the same dimension.

The likelihood ratio test statistic is computed for the test data using above parameters:[2]

$$d_{m,j} = \frac{p\left(\mathbf{x}^{\text{test}} | \boldsymbol{\theta}_{m,j}^{\text{ref},H_1}\right)}{p\left(\mathbf{x}^{\text{test}} | \boldsymbol{\theta}_{m,j}^{\text{ref},H_0}\right)}. \tag{4.4}$$

The deviations $d_{m,j}$ $(j = 1,2,\cdots,J)$, each corresponding to a different reference sub-series, are then combined, in this case by simply considering only their maximum

$$d_m = \max_j d_{m,j}, \tag{4.5}$$

giving the final anomaly score for the m^{th} window position.

4.2.3 Unexplored Algorithms

The analyses carried out in the previous sections allow to identify that the algorithm of Simmross-Wattenberg *et al.* [8] differs from Scherrer *et al.*'s algorithm [7] in five independent ways:

1. Periodic anomaly detection is performed, rather than local anomaly detection.
2. Attack information is assumed, rather than only non-attack information.
3. The reference and/or test sub-series are modelled at a single aggregation level, rather than multiple aggregation levels.
4. The alpha-stable marginal distribution model is assumed, rather than the gamma marginal distribution model.
5. A likelihood-based deviation is computed for the test sub-series given the reference parameters, rather than the L_2 distance between reference and test parameter vectors.

We now propose to construct a "space"[3] of algorithms that the above two inhabit. Denote each of the above five independent differences by the following variables: W, denoting

[2] $p(x|\boldsymbol{\theta})$ denotes the conditional probability density function of the random variable X as a function of the parameter vector $\boldsymbol{\theta}$, also known as the *likelihood* function.

[3] This usage of the word "space" does not refer to the rigorous definition of a vector space, although that definition provides a useful way of thinking about the ideas presented here.

Table 4.1. Summary of the variables constituting the "space" of unexplored detection algorithms.

Description	Symbol	Values	Chapter
Windowing Procedure	W	{local, periodic}	–
Information Availability	I	{incomplete, complete}	Chapter 5
Multi-Aggregation Level Representation	R_1	{single, multi}	Chapter 6
Marginal Distribution Model	R_2	{gamma, alpha-stable}	–
Deviation	D	{L_2 distance, likelihood}	Chapter 6

the windowing procedure; I, denoting the available information; R_1, denoting the multi-aggregation level representation; R_2, denoting the marginal distribution model; and D, denoting the deviation. Each 5-tuple of the form (W, I, R_1, R_2, D) specifies a unique detection algorithm, for a total of 2^5 $(= 32)$ unique algorithms, 30 of which are previously unexplored.

The following chapters will attempt to investigate this "space" of unexplored detection algorithms as summarized in Table 4.1. The scope of this study is narrowed down to only local rather than periodic anomaly detection algorithms. The effects of incomplete information (I) will be independently explored in Chapter 5. Initial numerical experiments with the α-stable distribution found that the required experiments would take a total of 399 days to complete.[4] Thus, only the gamma distribution model of R_2 will be considered. The remaining two variables, R_1 and D, will be jointly explored in Chapter 6.

4.3 Summary

This chapter detailed the first contribution of this thesis. A framework was proposed to deconstruct a class of anomaly detection algorithms into windows, representations, and deviations. The framework was applied to two state-of-the-art algorithms ([7] and [8]), allowing to identify five independent differences between them. By regarding each difference as a variable, a "space" of previously unexplored detection algorithms was constructed. The exploration of this "space" will be the focus of Chapters 5 and 6.

[4] The probability density function of the α-stable distribution is not available in closed-form. Therefore, fitting the α-stable parameters to data, and evaluating its probability density function is much more computationally intensive than for other probability distributions, e.g., the Gaussian or gamma distributions.

Chapter 5

Exploring Information Availability

Chapter 4 identified information availability, denoted by $I \in \{\texttt{incomplete}, \texttt{complete}\}$, as one of five key differences between two state-of-the-art detection algorithms ([7] and [8]). Detection theory provides the necessary tools to quantify the effects of information availability on detection accuracy. Therefore, this chapter investigates I by formulating volumetric Denial-of-Service (DoS) detection as a signal detection problem – the detection of a signal (the attack traffic) embedded in noise (the background traffic) – and comparing two detectors. The first detector operates without knowledge of the attack model ($I = \texttt{incomplete}$) and the second detector operates as if the attack model were known ($I = \texttt{complete}$).

5.1 Network Traffic Data

Numerical experiments are performed on real network traffic superposed with synthetically generated attacks. The benign dataset used here is from network traffic observed as part of the Measurement and Analysis of the WIDE Internet (MAWI) project [34]. The MAWI sample point F connects several Japanese research institutes and universities to the Internet. The archive offers 15 minutes of traffic from 14:00 to 14:15 (Japanese Standard Time) every day of the year since the year 2000. Packet payloads are removed and IP addresses are anonymized. The MAWI archive has proven tremendously useful for network traffic analysis research; an incomplete list of recent work using the archive includes [31, 32, 35–39]. The present

numerical experiments were performed on the particular traffic trace from Tuesday 25$^{\text{th}}$ July 2017, containing roughly 121 million packets transferred at an average rate of 974 Mbps.

Synthetic attacks were used, similar to [7] and [8]. In particular, the authors of [8] simulated flooding attacks using iPerf [28] (a network bandwidth measurement tool) on a quiet network, and superposed the resulting pure attack traffic on real recorded background traffic. Those authors acknowledge that this approximation does not take into account the changing behaviour of the underlying network protocols under the effect of the attack. However, they demonstrated that the marginal distribution of the superposed time series reasonably approximates that of a real attack time series [8, Fig. 4]. In this study, attacks have been further approximated as a step function in time - zero before the attack starts and some non-negative amplitude thereafter - rather than simulated via tools such as iPerf. The reason for this approximation is as follows. By definition, a *volumetric* denial of service attack floods a service with meaningless packets, thus the time series of packet counts rapidly rises to a maximum - the peak capacity allowed by the network - and remains there for the duration of the attack. In other words, the time series of a volumetric denial of service attack is well-approximated by a step function in time. Note, however, that this approximation does not necessarily generalise to other variations of the attack.

5.2 Detection Theoretic Formulation

Detection theory provides the tools required to quantify the effects of information availability on detection accuracy. Formulated within the framework of detection theory, the problem is to detect whether a signal (DoS attack traffic) is present in noise (background traffic).

5.2.1 Denial-of-Service Detection Problem

The time series of interest is the number of packets arriving at the network within non-overlapping one millisecond intervals, denoted by $\mathbf{w} = (w_1, w_2, \cdots, w_{N_{\text{total}})}$. As in Section 4.1.2, the gamma marginal distribution model is assumed. Formally, it is assumed that each w_n ($n = 1, 2, \cdots, N_{\text{total}}$) is sampled from a gamma distribution defined by the probability density

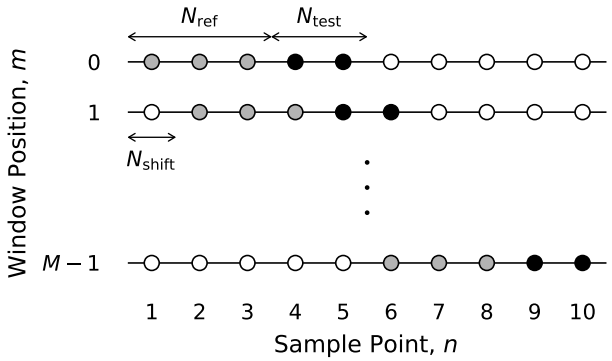


Fig. 5.1. A schematic description of the windowing procedure for a time series of length N_{total} (= 10) sample points. A sliding window is started at the initial position $m = 0$ (not shifted). The first N_{ref} (= 3) sample points of the window comprise the reference sub-series, represented by the grey circles; and the next N_{test} (= 2) sample points comprise the test sub-series, represented by the black circles. White circles represent sample points that are not processed for a given window position. The window is shifted by N_{shift} (= 1) sample points to give the next position $m = 1$ (shifted once). The process is repeated for a total of M window positions.

function

$$f(w_n;\alpha_n,\beta_n) = \frac{1}{\beta_n \Gamma(\alpha_n)} \left(\frac{w_n}{\beta_n}\right)^{\alpha_n - 1} \exp\left(-\frac{w_n}{\beta_n}\right), \tag{5.1}$$

for $w_n, \alpha_n, \beta_n > 0$. Here, $\Gamma(\cdot)$ denotes the gamma function, and α_n and β_n denote the shape and scale parameters of the gamma distribution.

Furthermore, w_n is assumed to be locally stationary (as discussed in Section 4.1.1); i.e., the parameters α_n and β_n are assumed to vary slowly with respect to n. Therefore, A reference window of length N_{ref} and a test window of length N_{test} are constructed, and shifted across the time series by N_{shift} sample points in each iteration. N_{ref} and N_{test} are chosen such that w_n observed within the window are stationary (α_n and β_n are approximately constant for n within the window). For each position of the window $m = 0, \cdots, M-1$, the sample points ($n = mN_{\text{shift}} + 1, \cdots, mN_{\text{shift}} + N_{\text{ref}}$) comprise the reference sub-series and the sample points ($n = mN_{\text{shift}} + N_{\text{ref}} + 1, \cdots, mN_{\text{shift}} + N_{\text{ref}} + N_{\text{test}}$) comprise the test sub-series. Fig. 5.1 provides a schematic description of the windowing procedure, demonstrated for an example time series of 10 sample points. Selection of appropriate values for N_{ref}, N_{test}, and N_{shift} will be guided in Section 5.3.1.

The detection problem is the same for each successive window position m. Without loss of generality, the case $m = 0$ will be considered here for notational simplicity. The detection

problem is to distinguish whether an attack appears in the background traffic time series w_n. The attack is modelled as an additive step function in time - zero before the attack starts and with amplitude $A\bar{w}$[1] thereafter (the last paragraph of Section 5.1 discusses the argument supporting this modelling assumption), where $A > 0$ denotes the attack intensity; i.e., the ratio of the attack pack rate to the benign packet rate. The problem is therefore to distinguish between the two hypotheses:

$$H_0 : x_n = w_n, \tag{5.2a}$$

$$H_1 : x_n = A\bar{w} + w_n, \tag{5.2b}$$

for $n = N_{\text{ref}} + 1, \cdots, N_{\text{ref}} + N_{\text{test}}$. Due to the assumed local stationarity, and provided N_{test} is chosen appropriately, w_n are identically gamma-distributed with parameters α and β; i.e., the parameters are no longer n-dependent within the window. The parameters α and β are unknown. However, it is assumed that the reference sub-series $(x_1, \cdots, x_{N_{\text{ref}}})$, consists only of noise, from which α and β can be estimated provided N_{ref} is sufficiently large. The next section defines two detectors to solve this problem with differing levels of available information.

5.2.2 Defining the Detectors

Detection problems are typically solved by optimizing a specified criterion. Two fundamental criteria include Bayes risk and the Neyman-Pearson (NP) criterion. Bayes risk assigns a prior probability to each hypothesis and a cost to each possible decision outcome. When it is difficult to assume prior probabilities or error costs, the NP criterion is appropriate, which attempts to maximise the probability of detection, P_{D}, for a given probability of false alarm, P_{FA}. It is well known that optimizing either criterion leads to detectors of the same general form; i.e., the likelihood ratio test (LRT) [13].

[1] $\bar{x}$ denotes the the arithmetic mean of the N-dimensional vector $\mathbf{x} = (x_1, x_2, \cdots, x_N)$; i.e., $\bar{x} = (\sum_{n=1}^{N} x_n)/N$.

Denoting an observed test sub-series by $\mathbf{x} = (x_{N_{\text{ref}}+1}, x_{N_{\text{ref}}+2}, \cdots, x_{N_{\text{ref}}+N_{\text{test}}})$, the LRT decides H_1 if

$$\Lambda(\mathbf{x}) = \frac{p_{\mathbf{X}|H_1}(\mathbf{x}|H_1)}{p_{\mathbf{X}|H_0}(\mathbf{x}|H_0)} > \eta. \tag{5.3}$$

In (5.3), $\Lambda(\mathbf{x})$ denotes the likelihood ratio, η denotes the threshold value, and $p_{\mathbf{X}|H_0}(\mathbf{x}|H_0)$ and $p_{\mathbf{X}|H_1}(\mathbf{x}|H_1)$ denote the conditional PDFs of $\mathbf{x}$ under each hypothesis. In the Bayesian approach, η is defined in terms of the assumptions about error costs and prior probabilities, whereas in the NP approach, η is defined such that a given P_{FA} is achieved. Here, the choice of criterion will be left arbitrary by referring in general to η.

In practice, it is rare that the attack model (and thus $p_{\mathbf{X}|H_1}(\mathbf{x}|H_1)$) will be known before the occurrence of the attack. This challenge is a primary reason for the growing interest in anomaly-based intrusion detection, which operates without an attack model. One such detector that operates with incomplete information decides H_1 if

$$\frac{1}{p_{\mathbf{X}|H_0}(\mathbf{x}|H_0)} > \xi, \tag{5.4}$$

where ξ is a threshold value. In contrast with (5.3), there is no optimality associated with (5.4). However, Bishop [40] has shown that under certain assumptions about $p_{\mathbf{X}|H_1}(\mathbf{x}|H_1)$, (5.3) and (5.4) are equivalent detectors. The first, simpler assumption is that $p_{\mathbf{X}|H_1}(\mathbf{x}|H_1)$ is equal to a constant on the support of $p_{\mathbf{X}|H_0}(\mathbf{x}|H_0)$. The second assumption is that $p_{\mathbf{X}|H_1}(\mathbf{x}|H_1)$ is of the form $F(p_{\mathbf{X}|H_0}(\mathbf{x}|H_0))$, where $F(\cdot)$ is a strictly decreasing function. Here, assumptions about $p_{\mathbf{X}|H_1}(\mathbf{x}|H_1)$ are implicitly contained within the threshold value ξ in (5.4).

These assumptions are not often appropriate in reality. Thus, the main contribution of this chapter is to quantify the difference in detection accuracy between (5.3) and (5.4) for a simple class of volumetric DoS attacks, thus investigating the effect of information availability I on detection accuracy.

5.2.3 Measuring Detection Accuracy

A complete measure of detection accuracy is given by the Receiver Operating Characteristic (ROC) curve; i.e., the curve obtained by observing the probability of false alarm P_{FA} versus

Table 5.1. Parameter settings used in the numerical experiments for exploring the effects of information availability on detection accuracy.

Symbol	Description	Value(s)
N_{ref}	Reference window length	60 sec
N_{test}	Test window length	2^k $(k = 0, 1, \cdots, 11)$ sample points
N_{shift}	Window shift length	0.1 sec
A	Attack Intensity	$0.01l$ $(l = 1, 2, \cdots, 50)$

the probability of detection P_{D}, while varying the threshold λ. The ROC curve allows to disentangle the assessment of a detector's accuracy from problem-specific assumptions, such as prior probabilities and error costs [13]. It is often useful to summarize accuracy by computing the Area Under the ROC Curve (AUC), ranging from 0.5, meaning that the detector is no better than flipping a coin, to 1, meaning that the detector has perfect accuracy. In the results to follow, detection accuracy is measured in terms of the AUC.

5.3 Results and Discussion

The purpose of these numerical experiments is to evaluate the detection accuracy of two detectors for a constant-rate volumetric DoS attack as detailed in the previous section. This section guides the selection of appropriate parameters, and thereafter discusses the experimental results

5.3.1 Parameter Settings

The parameters used in this study are detailed in Table 5.1. These particular values were chosen according to the following criteria:

1. The reference window length N_{ref} was selected to attain a balance between avoiding non-stationarity within each window position and providing a sufficient number of samples so that model parameter estimation accuracy did not affect the results.

2. The test window length N_{test} was dyadically increased, until further increases resulted in erratic detection accuracy (which the author suspects is due to introducing non-stationarity into the test sub-series).

3. The window shift length N_{shift} ensured $M \approx 9000$ window positions and therefore detection opportunities per experiment, from which to reliably compute P_{FA} and P_{D}.

4. The attack intensity A was uniformly increased, such that any smaller lead to barely detectable attacks, and any larger resulted in too easily detectable attacks.

5.3.2 AUC Versus Test Window Length

Each sub-figure in Fig. 5.2 fixes A and plots the AUC as a function of N_{test} on a semi-logarithmic axis. The blue curve corresponds to the detector with incomplete information ($I = \texttt{incomplete}$), which for the present discussion will be referred to as Detector $\mathscr{A}$. The orange curve corresponds to the detector with complete information ($I = \texttt{complete}$), which will be referred to as Detector $\mathscr{B}$. For small intensities $A < 5\%$ (Fig. 5.2a), Detector $\mathscr{A}$'s accuracy is roughly constant with respect to N_{test}, whereas Detector $\mathscr{B}$'s accuracy is increasing and piecewise linear with a breakpoint at $N_{\text{test}} \in (200, 300)$. Note that the piecewise linearity is observed on a semi-logarithmic axis. The figure demonstrates that the accuracy of Detector $\mathscr{A}$ remains poor with negligible improvement, despite a substantial increase in the number of samples available for detection.

At $A = 5\%$ (Fig. 5.2b), Detector $\mathscr{A}$'s accuracy begins to increase with N_{test}, however, the difference between the two curves still increases with N_{test}. The curve corresponding to Detector $\mathscr{A}$ also shows approximate piecewise linearity (on the semi-logarithmic axis) with the breakpoint at $N_{\text{test}} \in (200, 300)$. When the intensity reaches $A = 10\%$ (Fig. 5.2c), Detector $\mathscr{B}$ begins to saturate for $N_{\text{test}} \in (500, 600)$; i.e., further increasing N_{test} yields minimal increase in accuracy. This observation is to be expected as detection accuracy draws nearer to perfect. The saturation has the effect that at this intensity, the accuracy difference between the detectors decreases for large N_{test}.

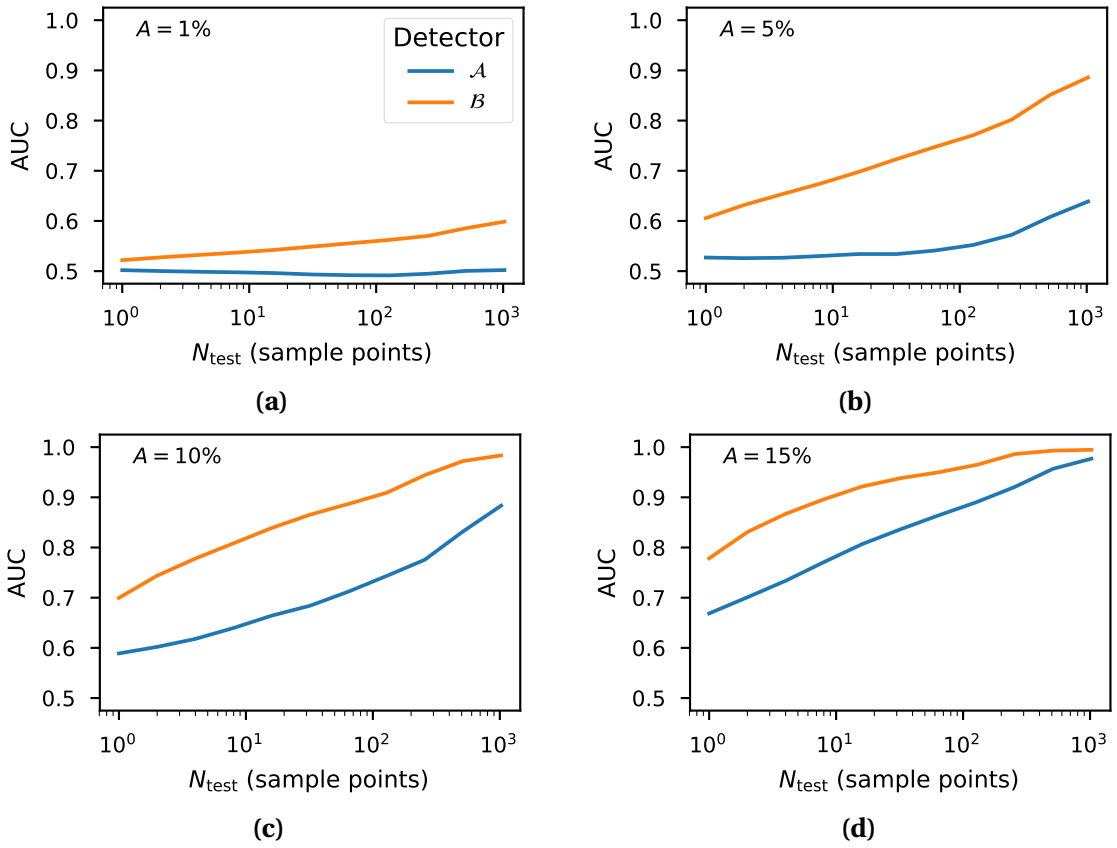


Fig. 5.2. Accuracy, in terms of AUC, versus logarithmically-scaled test window length N_{test} of detectors with varying information availability $I \in \{\texttt{incomplete}, \texttt{complete}\}$. (a) $A = 1\%$. (b) $A = 5\%$. (c) $A = 10\%$. (d) $A = 15\%$.

Whereas for $A < 15\%$, both curves have positive curvature, for $A \geq 15\%$ (Fig. 5.2d), both curves have negative curvature. At this intensity, Detector $\mathscr{A}$ begins to saturate for $N_{\text{test}} \in (500, 600)$. For greater intensities than have been plotted here, no further notable changes are observed; the curves draw increasingly nearer to each other as they asymptotically approach perfect accuracy (AUC ~ 1).

5.3.3 AUC Versus Attack Intensity

Each sub-figure in Fig. 5.3 fixes N_{test} and plots the AUC as a function of A. As before, the blue curve corresponds to Detector $\mathscr{A}$ ($I = \texttt{incomplete}$), and the orange curve corresponds to Detector $\mathscr{B}$ ($I = \texttt{complete}$). While Detector $\mathscr{B}$'s curve has positive curvature throughout, Detector $\mathscr{A}$'s curve has negative curvature for small intensities (roughly $A < 5\%$) and positive

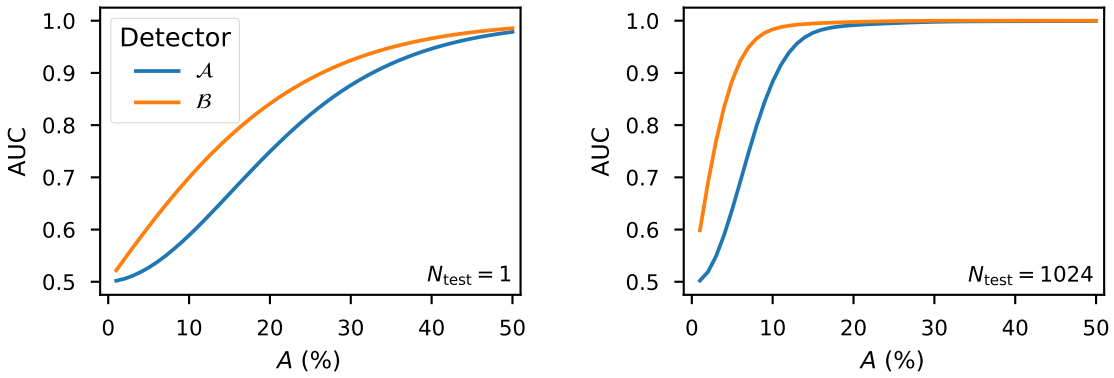


Fig. 5.3. Accuracy, in terms of AUC, versus attack intensity A of detectors with varying information availability $I \in \{\texttt{incomplete}, \texttt{complete}\}$. (a) $N_{\text{test}} = 1$ sample point. (b) $N_{\text{test}} = 1024$ sample points.

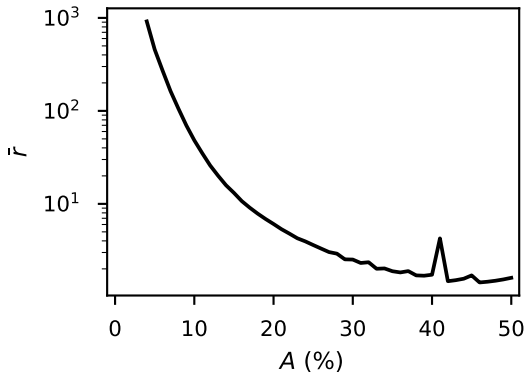


Fig. 5.4. A measure, $\bar{r}$, of the difference in accuracy between Detector $\mathscr{A}$ and Detector $\mathscr{B}$ versus attack intensity A. The spike at $A = 40\%$ is due to the fact $\bar{r}$ is the arithmetic mean across random experiments with gamma-distributed values. The gamma distribution provides higher likelihood to extreme values than the classical Gaussian distribution, and the mean is not a robust statistic, i.e., its value is biased by the presence of extreme values.

curvature thereafter. The negative curvature demonstrates a structural deficiency in Detector $\mathscr{A}$ for small intensity attacks. For each fixed N_{test}, both detectors' accuracies saturate with increasing A and the performance difference diminishes. As N_{test} increases, the curves appear to be increasingly compressed along the A-axis, giving the effect that the performance difference diminishes at lower intensities.

5.3.4 Performance Loss Measure

The performance difference is characterized in another way in Fig. 5.4. In order to describe the function $\bar{r}(A)$ plotted in the figure, the reader is first referred to Fig. 5.2c. For $A = 10\%$, Detector $\mathscr{B}$ achieves an AUC of approximately 0.7 at $N_{\text{test}} = 1$. Since the curve is monotonically increasing, it can be said that $N_{\text{test}} = 1$ is the minimum required value for Detector $\mathscr{B}$ to achieve an AUC of 0.7. Similarly, $N_{\text{test}} \approx 40$ is the minimum required value for Detector $\mathscr{A}$ to achieve the same AUC of 0.7. Denote by $r(\text{AUC}, A)$ the ratio of minimum required N_{test} values to achieve a desired AUC. Continuing with the example, it follows that $r(0.7, 10\%) \approx 40/1 = 40$; i.e., for attacks of intensity $A = 10\%$, Detector $\mathscr{A}$ requires a test window length 40 times that of Detector $\mathscr{B}$ to achieve the same AUC of 0.7. Finally, denote by $\bar{r}(A)$ the average of $r(\text{AUC}, A)$ over all AUC values, which has been plotted in the figure. The curve in Fig. 5.4 demonstrates two key points. First, for small intensities, Detector $\mathscr{A}$ requires a substantially larger N_{test} than Detector $\mathscr{B}$. Second, as the intensity increases, $\bar{r}(A)$ decays slowly; even at $A = 50\%$, Detector $\mathscr{A}$ requires on average 1.5 times the N_{test} of Detector $\mathscr{B}$.

5.4 Summary

This chapter quantified the effects of information availability, $I \in \{\texttt{incomplete}, \texttt{complete}\}$, on detection accuracy. Two detectors were constructed: one without an attack model ($I = \texttt{incomplete}$) and one with an attack model ($I = \texttt{complete}$). The detectors were evaluated and compared in order to investigate the effects of varying information availability I on detection accuracy. Three key findings were as follows. First, the inferior accuracy in the case $I = \texttt{incomplete}$ is most notable for attacks of small intensities ($A < 5\%$). Second, the accuracy of both detectors appears to undergo a structural change around $A = 15\%$. Third, the performance difference decays as A increases, however at a slow rate; at a high attack intensity ($A = 50\%$), Detector $\mathscr{A}$ still requires 1.5 times as many samples for detection than Detector $\mathscr{B}$ in order to match its performance on average.

Chapter 6

Exploring Representations and Deviations

Chapter 4 identified five key differences between two state-of-the-art detection algorithms ([7] and [8]). This chapter investigates two of these differences by constructing four detectors with varying representation, denoted by $R \in \{\texttt{single}, \texttt{multi}\}$, and varying deviation, denoted by $D \in \{\texttt{l2-distance}, \texttt{likelihood}\}$. Each detector is evaluated in a series of numerical experiments, allowing to assess the effects of the variables R and D on detection accuracy.

6.1 Network Traffic Data

Two separate sets of data are required for the present numerical experiments: the benign data, assumed to be mostly free of malicious activity; and the attack data, assumed to consist mostly of malicious activity. A hypothetical scenario is created by superposing the attack data on the benign data. Since the starting and ending times of the attack are known, the data are considered labelled. The benign and attack data used in this chapter are from two separate network traffic archives.

The benign data are from the MAWI network traffic archive (also utilised and described in Section 5.1). For the present numerical experiments, data from 14 consecutive Tuesdays were used – starting from 25$^{\text{th}}$ July 2017 and ending on 24$^{\text{th}}$ October 2017. The attack data are

Table 6.1. Summary of the benign datasets from the MAWI network traffic archive [34].

Index	Date	Packet Rate (pkt/s)	Bit Rate (Mb/s)
1	25/07/2017	135	974
2	01/08/2017	124	825
3	08/08/2017	101	633
4	15/08/2017	92	507
5	22/08/2017	117	670
6	29/08/2017	119	757
7	05/09/2017	101	590
8	12/09/2017	105	637
9	19/09/2017	110	692
10	26/09/2017	139	973
11	03/10/2017	137	965
12	10/10/2017	133	973
13	17/10/2017	139	968
14	24/10/2017	132	970

from the Booters dataset: network traffic measured as part of a collaboration between the Dutch National Research and Education Network (SURFnet) and the University of Twente. A series of volumetric UDP-based Distributed Denial-of-Service (DDoS) attacks were launched at a null-routed IP address at the university. The attacks originated from DDoS-for-hire services, colloquially termed *booters*. Initial investigations identified twenty one potential booters, of which seven had faulty payment systems. Of the fourteen remaining booters: three did not send any traffic, and two sent a handful of TCP packets. The final nine booters performed attacks as requested, generating more than 250 GB of traffic in total. Seven of the attacks were DNS-based reflected and amplified DDoS attacks; and two of the attacks were CharGen reflected and amplified DDoS attacks. The benign and attack datasets are summarised in Tables 6.1 and 6.2, respectively.

Each raw network traffic packet dataset (in .pcap file format) was processed to obtain the time series of packet counts Thereafter, each attack time series was superposed on each benign time series for a total of 126 ($= 14 \times 9$) benign-attack pairs. The next section will detail the approach taken to evaluate each detector on each superposed time series.

Table 6.2. Summary of the attack datasets from the Booters network traffic archive [41].

Index	Packet Rate (pkt/s)	Bit Rate (Mb/s)	Attack Sources
1	46	700	4486
2	32	250	78
3	48	330	54
4	64	1190	2970
5	8	6	8281
6	83	150	7379
7	36	320	6075
8	54	990	281
9	86	5480	3779

6.2 Experimental Approach

Four detectors are considered here, obtained by varying representation, $R \in \{\texttt{single}, \texttt{multi}\}$, and deviation, $D \in \{\texttt{l2-distance}, \texttt{likelihood}\}$ (see Chapter 4 for further details). This section describes the approach taken to evaluate a single detector run on a single labelled time series (i.e., with known attack start and end times).

6.2.1 Evaluating Detectors

The reader is encouraged to supplement the following description with the visual aid presented in Fig. 6.1. By stating that a given time series $\mathbf{x}$ is "labelled" we are indicating that the attack start time τ_1 and attack end time τ_2 (also known as the *changepoints*) are known. The goal of the detection algorithm is to correctly estimate τ_1 and τ_2 given only $\mathbf{x}$. Applying the algorithms constructed in Chapter 4 to a time series $\mathbf{x}$ results in a sequence of deviation scores d_t, where t denotes the discrete point in time. Since the algorithms considered here are *local* anomaly detection algorithms, d_t can be interpreted as a measure of how likely it is that the local stationarity assumption was broken. It is expected that d_t should peak nearby an actual changepoint; i.e., $t = \tau_i$ $(i = 1, 2)$. Following previous work [42], each peak in the deviations sequence is considered an estimate of the changepoints $\{\tau_i\}$.

In order to reduce the number of false positives, only peaks with values greater than a pre-defined threshold λ may be considered. In other words, denoting the time of the

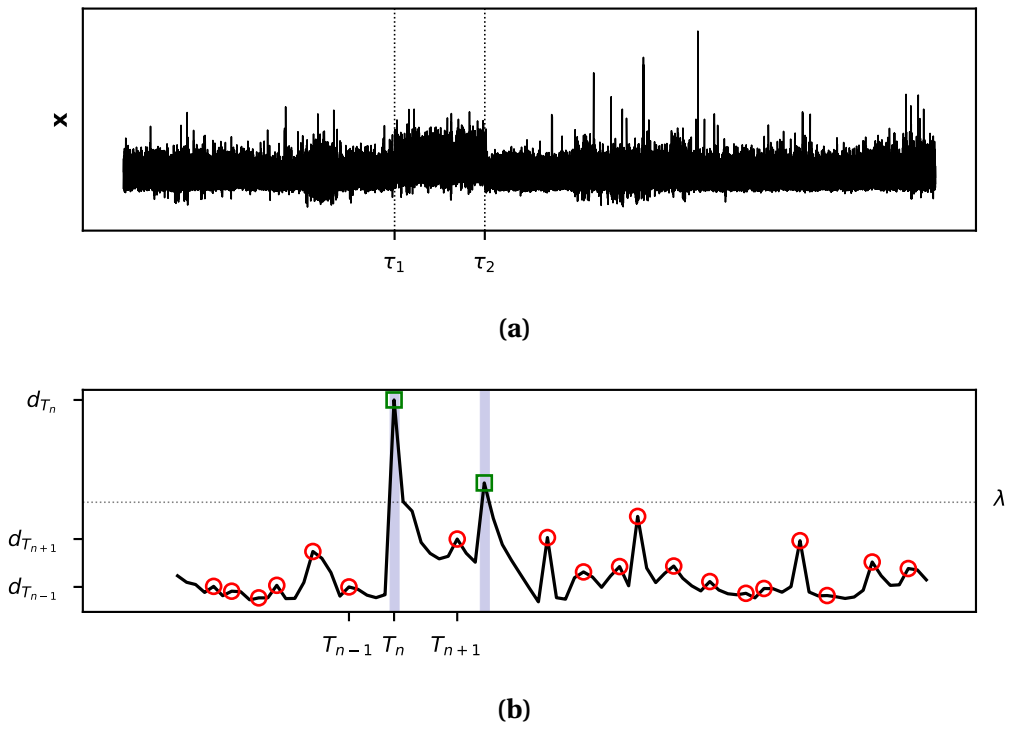


Fig. 6.1. Visualising the evaluation of a detector on a labelled time series. (a) A labelled time series **x** with an attack starting at τ_1 and ending at τ_2. (b) The deviation scores d_t output by a detector. Peaks in d_t occur at times T_n. Those peaks within the allowed bands (shaded in blue) are considered true positives (green squares), otherwise they are considered false positives (red circles). If a threshold is applied, then only peaks with d_{T_n} greater than a predefined threshold value λ are considered as potential alarms. The labels y_n for the three peaks identified in the figure are $(0,1,0)$.

n^{th} peak in d_t by T_n, T_n is only considered an alarm if $d_{T_n} > \lambda$. Since the time series data used here were sampled at millisecond intervals, it is perhaps too much to expect T_n to correspond exactly with τ_i. Instead, a band of, say ± 5 seconds, is allowed; detecting an attack 5 seconds after it occurs is considered a valid detection. In summary, an estimated changepoint T_n with value larger than the threshold ($d_{T_n} > \lambda$) is considered a true positive if $T_n \in [\tau_i - 5\,\text{sec}, \tau_i + 5\,\text{sec}]$ for $i = 1,2$, otherwise, T_n is considered a false positive. It will be useful to construct a label variable, y_n, which is equal to 1 if the n^{th} peak is a true positive, and 0 if it is a false positive. The result of each numerical experiment; i.e., the application of a given detector to a given labelled time series can therefore be summarized by a collection of estimated changepoints with times $\{T_n\}$, values $\{d_{T_n}\}$, and labels $\{y_n\}$. From these quantities, the evaluation metrics discussed in the next section may be computed.

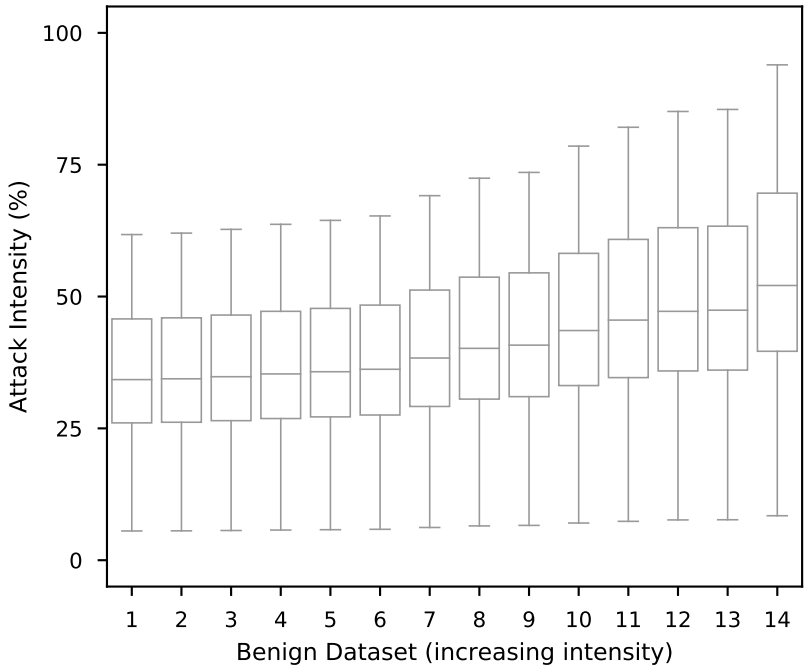
6.2.2 Measuring Detection Accuracy

Detection accuracy is measured here by the Area Under the Receiver Operating Characteristic Curve (AUC), ranging from 0.5, meaning that the detector is no better than flipping a coin, to 1, meaning that the detector has perfect accuracy. While the AUC summarizes the entire Receiver Operating Characteristic (ROC) curve, in certain cases only the region of the curve for very small P_{FA} is of interest. Consider the following example. Each detection algorithm has a *decision rate*, r_{dec}, the rate at which it makes decisions as to whether an attack has occurred or not. For the parameters used in this chapter, the decision rate $r_{\text{dec}} = 8640$ decisions per day. The *error rate*, r_{err}, the rate at which incorrect decisions are made, is determined by the probability of false alarm and decision rate as $r_{\text{err}} = P_{\text{FA}} r_{\text{dec}}$. Fixing some probability of false alarm, say $P_{\text{FA}} = 0.1$, meaning that 10% of all decisions will be incorrect, yields an error rate of $r_{\text{err}} = (0.1)(8640) = 864$, errors per day! False alarms are considerably costly since they require time and effort to be diagnosed by a team of experts. A more reasonable error rate would be in the order of one error per day. To achieve this error rate would require $P_{\text{FA}} = r_{\text{err}} / r_{\text{dec}} = 1/8640 \approx 10^{-4}$. Therefore, in addition to the AUC, which summarizes the ROC curve over all possible P_{FA}, we will assess the P_{D} for a given $P_{\text{FA}} = 10^{-4}$, which yields an error rate of approximately one error per day; a practical yet considerably more stringent assessment than the AUC. Thus far, only the evaluation for a single dataset has been covered. The next section will detail the evaluation across many different benign and attack datasets.

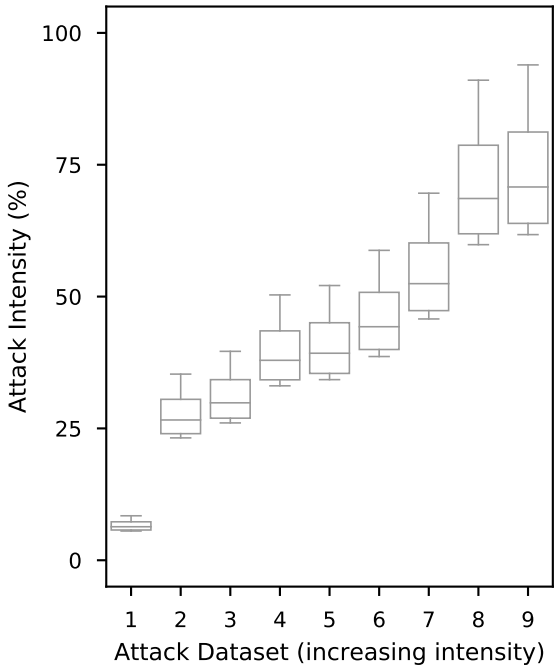
6.2.3 Experiments Across Multiple Datasets

Thus far the discussion has focused on a single numerical experiment; i.e., a single detector applied to a labelled time series obtained by superposing an attack time series on a benign time series. However, the analysis is performed across 14 different benign datasets, 9 different attack time series, and 4 different detectors.

A useful quantity for parametrizing DoS attacks is the attack intensity; i.e., the ratio of the attack packet rate to the benign packet rate. The attack intensity is expected to be



(a)



(b)

Fig. 6.2. Box plots of the attack intensity aggregated across datasets. (a) Attack intensity of benign datasets, sorted in increasing order of intensity, aggregated across attack datasets. (b) Attack intensity of attack datasets, sorted in increasing order of intensity, aggregated across benign datasets.

correlated with the detection accuracy; attacks of lower intensity are harder to detect, while attacks of higher intensity are easier to detect. The attack intensity has been calculated for each possible benign-attack pair. Fig. 6.2 shows box plots of the attack intensities aggregated (separately) across all benign and attack datasets and ordered by increasing intensity. Fig. 6.2b illustrates that the attack intensity varies over a similar range (0% to at least 60%) for each benign dataset. Fig. 6.2a shows that for a given attack dataset the attack intensity does not vary much across benign datasets. Given these two observations, it would be reasonable to separate the analyses by attack dataset (indirectly separating by attack intensity). That is, for each attack time series (correspondingly, attack intensity) experimental results will be aggregated across all benign time series. This will be repeated for each detector, and their results will be compared. Aggregating across multiple benign datasets also potentially reduces biases that may be present in a single dataset, e.g., a certain detector may perform better in one benign dataset but not others.

6.3 Results and Discussion

The results obtained by performing the numerical experiments described in the previous section are now detailed. Each of the four detectors is evaluated on each combination of benign and attack time series. The detectors are evaluated in terms of their accuracy, measured by the AUC in addition to the probability of detection, P_{D}, for a given probability of false alarm, $P_{\mathrm{FA}} = 10^{-4}$, chosen such that the resultant error rate is in the order of one error per day, a manageable quantity. First, parameter settings are guided Thereafter, results are presented for detectors with varying representation, $R \in \{\texttt{single}, \texttt{multi}\}$, and deviation, $D \in \{\texttt{l2-distance}, \texttt{likelihood}\}$.

6.3.1 Parameter Settings

For each benign-attack time series pair, the attack time series is superposed at 1000 different starting times, in order to avoid any bias caused by only detecting attacks that occur at the same time. A further advantage is that the number of numerical experiments is increased,

Table 6.3. Parameter settings used in the numerical experiments for exploring the effects of representations and deviations on detection accuracy.

Symbol	Description	Value
N_{ref}	Reference window length	60 sec
N_{test}	Test window length	10 sec
N_{shift}	Window shift	10 sec
–	Aggregation factor	2
K	Aggregation levels	7
–	Changepoint interval	5 sec

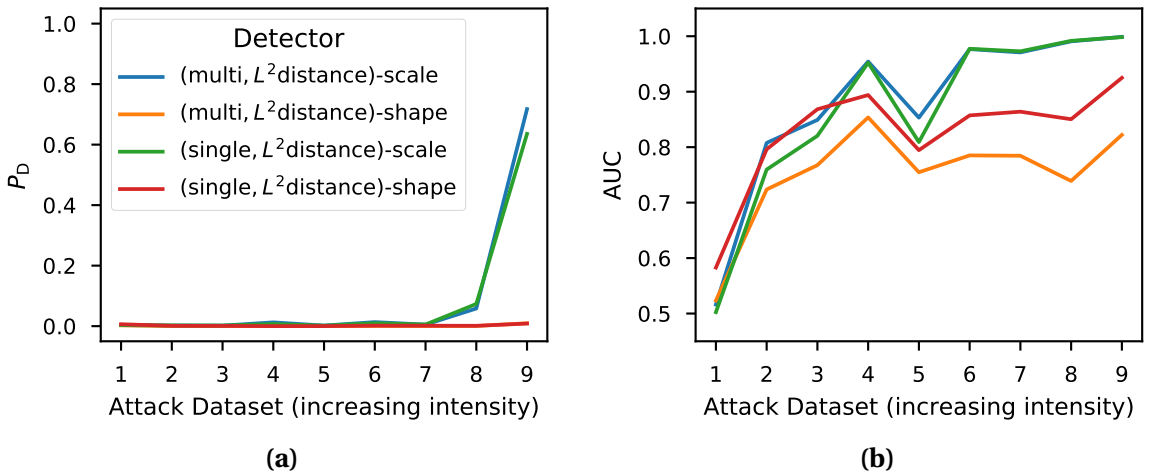
thus increasing the number of detection attempts made by each detector, which provides a higher resolution ROC curve. This is necessary in order to obtain the P_{D} for a P_{FA} as low as is required (10^{-4}).

The detector parameters used for the present numerical experiments are presented in Table 6.3. Each parameter is chosen according to the following criteria:

1. The reference window length, N_{ref}, and test window length, N_{test}, should be chosen such that sub-series observed within intervals of length $N_{\text{ref}} + N_{\text{test}}$ are stationary. Additionally, each window should be long enough to provide a minimum number of required samples by any statistical techniques utilised.

2. The window shift length N_{shift} determines the decision rate – the rate at which detection attempts occur and attacks may be detected. A higher decision rate is of course desirable, however, also requires increasingly smaller P_{FA} to avoid impractically high error rates. Furthermore, the computational cost is proportional to the decision rate.

3. For detectors with $R = \texttt{multi}$, the aggregation factor – the length of the intervals within which the time series is successively aggregated – should be chosen as per [7, 31]. The maximum aggregation level K – the maximum number of times the time series is successively aggregated – should be chosen as large as possible while still yielding aggregated sub-series with a sufficient number of samples required to fit a gamma probability distribution.



(a) **(b)**

Fig. 6.3. Accuracy of detectors with fixed deviation, $D = \texttt{l2-distance}$, and varied representation R. (a) Probability of detection P_D for fixed probability of false alarm $P_{FA} = 10^{-4}$. (b) AUC.

4. The length of the interval surrounding changepoints within which an alarm is considered a true positive should be set between 5 and 10 seconds, following previous work [42].

6.3.2 Non-monotonicity of Accuracy Versus Attack Dataset

It is interesting to note that the accuracy (measured in terms of both AUC and P_D) versus attack dataset does not monotonically increase in all of the presented figures: Fig. 6.3, Fig. 6.4, Fig. 6.5, and Fig. 6.6. This non-intuitive behaviour is due to the fact that the ordering of attack datasets by increasing *intensity* - i.e., ratio of attack packet rate to benign packet rate - does not necessarily reflect the true difficulty of detecting an attack in all cases. In particular, Attack Dataset 5, appears to be at least as difficult to detect as Attack Dataset 3 and perhaps also Attack Dataset 2. While this suggests that the dynamics of Attack Dataset 5 might be worth further investigating, the ordering by attack intensity is retained, as no technically consistent alternative presented itself.

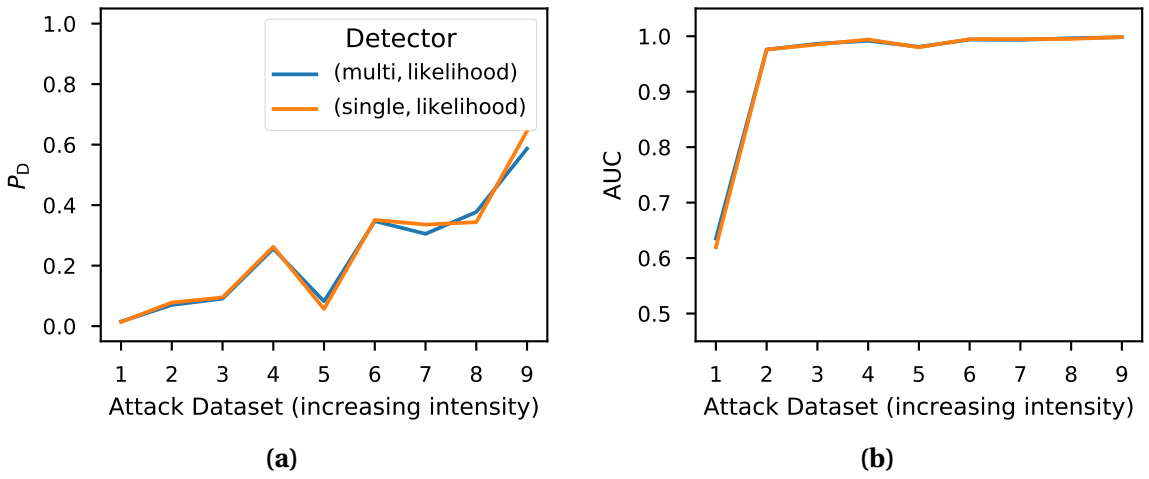
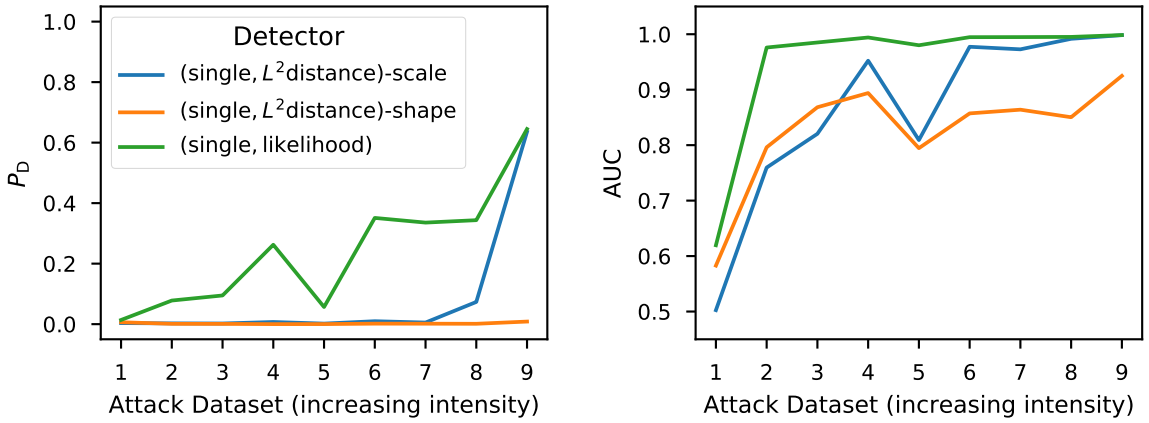


Fig. 6.4. Accuracy of detectors with fixed deviation, $D = \texttt{likelihood}$, and varied representation R. (a) Probability of detection P_D for fixed probability of false alarm $P_{FA} = 10^{-4}$. (b) AUC.

6.3.3 Single Versus Multiple Aggregation Levels

The first variable to be investigated is the representation. Detectors with fixed deviation, $D = \texttt{l2-distance}$, and varied representation $R \in \{\texttt{single}, \texttt{multi}\}$ are compared in terms of their P_D and AUC in Fig. 6.3. Each detector here returns two outputs, corresponding to the shape and scale parameters of the gamma distribution, treated as two separate detectors (see Chapter 4 for further details). All detectors are incapable of detecting attacks given the required P_{FA}. However, scale parameter detectors are capable of detecting the largest intensity attack. Varying R has negligible effect. In terms of the AUC, scale parameter detectors perform better for larger intensity attacks, whereas shape parameter detectors with $R = \texttt{single}$ perform slightly better for lower intensity attacks. Setting $R = \texttt{multi}$ improves the accuracy for scale parameter detectors (green to blue) – particularly for low intensity attacks – while worsening the accuracy for shape parameter detectors (red to orange). The curves in Fig. 6.4 correspond to detectors obtained by fixing $D = \texttt{likelihood}$ and varying R. Here, the effect of varying R is negligible in terms of both the P_D and the AUC.

In summary, it appears that varying R does not have much effect on detection accuracy as measured by P_D. However, setting $R = \texttt{multi}$ improves accuracy for scale parameter detectors and worsens accuracy for shape parameter detectors in terms of AUC. Interestingly,



(a) **(b)**

Fig. 6.5. Accuracy of detectors with fixed representation, $R = \texttt{single}$, and varied deviation D. (a) Probability of detection P_D for fixed probability of false alarm $P_{FA} = 10^{-4}$. (b) AUC.

the shape parameter has higher AUC for larger intensity attacks, whereas the scale parameter is more sensitive for lower intensity attacks.

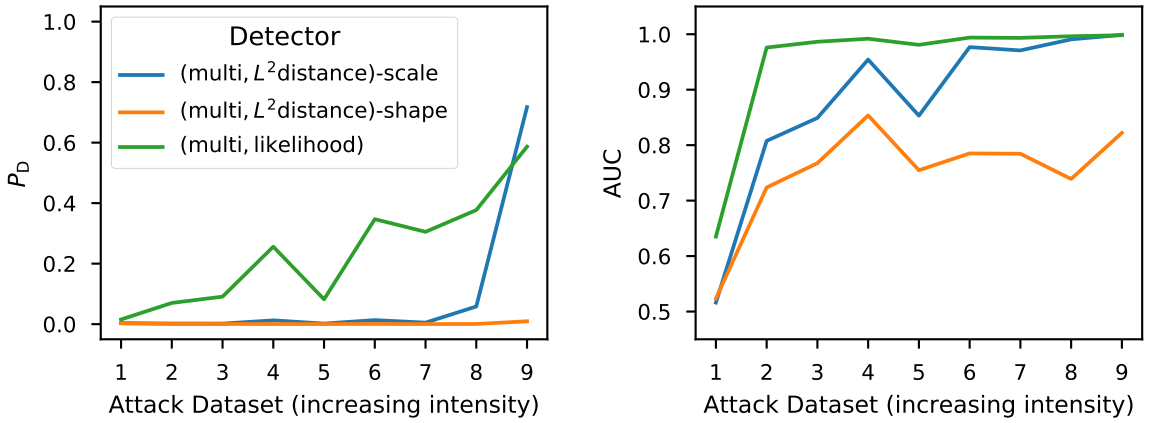
6.3.4 L_2 Distance Versus Likelihood

Fig. 6.5 plots the accuracy for detectors with fixed $R = \texttt{single}$ and varied D. While for $D = \texttt{l2-distance}$, no detectors are capable of detecting attacks given the required P_{FA}, for $D = \texttt{likelihood}$, a near perfect AUC of 1 is achieved for all but the lowest intensity attack.

Finally, Fig. 6.6 plots the accuracy for detectors with fixed $R = \texttt{multi}$ and varied D. A similar pattern is observed to the previous figure, except in this case the gap between $D = \texttt{likelihood}$ and $D = \texttt{l2-distance}$ detectors is narrower, further evidencing that setting $R = \texttt{multi}$ pairs well with $D = \texttt{l2-distance}$, though does not seem to affect $D = \texttt{likelihood}$.

6.4 Summary

This chapter explored the space of detection algorithms identified in Chapter 4. Each detector was evaluated in terms of the area under the ROC curve (AUC) and probability of detection P_D for a fixed P_{FA}. The detectors were evaluated on 9 attack datasets, each superposed on 14 benign datasets, giving a total of 126 labelled datasets. Two variables



(a) **(b)**

Fig. 6.6. Accuracy of detectors with fixed representation, $R = \texttt{multi}$, and varied deviation D. (a) Probability of detection P_D for fixed probability of false alarm $P_{FA} = 10^{-4}$. (b) AUC.

were assesed through the evaluations: representation, denoted by $R \in \{\texttt{single}, \texttt{multi}\}$, and deviation, denoted by $D \in \{\texttt{l2-distance}, \texttt{likelihood}\}$. Three key findings were as follows. First, varying R has little effect for $D = \texttt{likelihood}$ detectors. Second, setting $R = \texttt{multi}$ negatively affects $D = \texttt{l2-distance}$ detectors based on the shape parameter, while positively affecting those based on the scale parameter. Third, overall, $D = \texttt{likelihood}$ is superior to $D = \texttt{l2-distance}$, although the difference becomes negligible for large intensity attacks.

Chapter 7

Conclusions

The aim of this thesis is to provide an underlying theory for understanding a single class of cyber-attack detection algorithms. This section will recall how each contribution presented in this thesis aligns with the original aim.

Chapter 4 proposed a framework for deconstructing detection algorithms into windows, representations, and deviations. The framework was then applied to two state-of-the-art algorithms, allowing to identify five key differences between them. By regarding each difference as a variable, a class of detection algorithms was constructed. In particular, the framework allowed to link the type of anomalies detected by each detector to a distinct assumption about the observed time series; i.e., local or periodic stationarity.

Chapter 5 investigated the effect of information availability on detection accuracy. In order to do so, a simple problem – detection of a constant-rate Denial-of-Service (DoS) attack – was modelled within the framework of detection theory. By explicitly modelling the probability distribution of the observed data under two scenarios (no attack present versus attack present), and constructing detectors that operate with and without attack information, we were able to quantify the effect of information availability on detection accuracy. It was found that the attack information particularly helps the detector discriminate when the data cannot be easily identified as an attack or not; i.e., when the attack intensity is low. Furthermore, while the effect decays as the attack intensity grows, it does so at a slow rate, i.e., at a high attack intensity (50%) detection without attack information still requires 1.5 times

as many samples as detection with attack information in order to match its performance on average.

Finally, Chapter 6 proposed an experimental procedure to jointly investigate the effects of the representation and deviation of the detector on its accuracy. Interestingly, it was found that moving from single- to multiple- aggregation levels provided negligible improvement in detection accuracy, in some cases notably degrading the accuracy. Further, it was found that the likelihood deviation greatly outperformed the L_2 distance between parameters of a fitted probability distribution (supported by results in detection theory [13]).

In summary, a framework was proposed to deconstruct detection algorithms into their constituent components. The framework was applied to a class of detection algorithms, allowing to construct a "space" – each point in the space corresponding to a distinct configuration of detector components. By exploring this space, and leveraging notions from detection theory, we gained deeper insights into a class of detection algorithms.

Future Work

Five directions for further research have been identified:

1. Of the five key differences identified between the studied algorithms, two were excluded: periodic anomaly detection, and the alpha-stable marginal distribution model. Expanding the class of algorithms studied here to include these aspects is the first proposed direction of further research.

2. The present investigation considered algorithms that each operate on a single time series $\mathbf{x} = (x_1, x_2, \cdots, x_N)$. These algorithms are capable of detecting points in time n $(= 1, 2, \cdots, N)$ where an attack might have occurred. However, extending the class of algorithms to operate on multiple time series $\{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_M\}$ each corresponding to a distinct IP address, would allow to identify users m $(= 1, 2, \cdots, M)$ responsible for an attack. Considering the IP address as a *spatial* component, joint spatio-temporal analysis techniques could be used to identify both the point in time n and the user m responsible for malicious activity.

3. Windows with adaptive shifts, widths, and shapes could be explored. Denote the shift and width of the m^{th} window by S_m and N_m, respectively. Let the special case where $S_m = S$ and $N_m = N$ correspond to fixed shifts and widths. The values of S_m and N_m could be adaptively computed based on statistical properties of the observed traffic. For example, during periods of high activity, S_m and N_m could be decreased for a finer analysis, and during extended periods of low activity, S_m and N_m could be increased. Furthermore, increasingly more complex window shapes can be designed in regard to their tapering properties.

4. The simplistic attack model assumed in Chapter 5 can be extended to better reflect the effects of the attack on the underlying network protocols. For instance, the proposed detection theoretic formulation provides a platform to utilize previous work on the spectral characterization of DoS attacks [43] for enhanced detection. The framework can also easily be extended to attacks other than the DoS attack.

5. Furthermore, the attack model was assumed here to be either completely known or unknown. It is perhaps unrealistic to know the attack model completely before the attack occurs. Therefore, the effects of an attack model mismatch could be studied. This would allow to identify under which conditions it might be more effective not to assume an attack model at all, but rather to take the anomaly-based approach.

References

[1] M. W. Lorgat, A. Baghai-Wadji, and A. McDonald, "Towards a general framework for network traffic time series anomaly detection," in *Proc. Eur. Conf. Cyber Warfare and Security*, Jun. 2017, pp. 252–260.

[2] ——, "Quantifying the effect of incomplete information in denial of service detection," in *Proc. Global Wireless Summit*, Oct. 2017, pp. 67–71.

[3] "Framework for improving critical infrastructure cybersecurity," National Institute of Standards and Technology, Gaithersburg, Maryland, United States, Tech. Rep. Version 1.1, Draft 2, 2017.

[4] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Network anomaly detection: Methods, systems and tools," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 1, pp. 303–336, Jan.–Mar. 2014.

[5] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Proc. 2010 IEEE Symp. Security and Privacy*, May 2010, pp. 305–316.

[6] C. Gates and C. Taylor, "Challenging the anomaly detection paradigm: A provocative discussion," in *Proc. 2006 New Security Paradigms Workshop*, Sep. 2006, pp. 21–29.

[7] A. Scherrer, N. Larrieu, P. Owezarski, P. Borgnat, and P. Abry, "Non-gaussian and long memory statistical characterizations for internet traffic with anomalies," *IEEE Trans. Depend. Sec. Comput.*, vol. 4, no. 1, pp. 56–70, Jan. 2007.

[8] F. Simmross-Wattenberg, J. I. Asensio-Perez, P. C. de-la Higuera, M. Martin-Fernandez, I. A. Dimitriadis, and C. Alberola-Lopez, "Anomaly detection in network traffic based on statistical inference and α-stable modeling," *IEEE Trans. Depend. Sec. Comput.*, vol. 8, no. 4, pp. 494–509, Jul. 2011.

[9] S. Axelsson, "Research in intrusion detection systems: A survey," Dept. Comput. Eng., Chalmers Univ. Technol., Göteborg, Sweden, Tech. Rep. 98-17, 1999.

[10] D. E. Denning, "An intrusion-detection model," *IEEE Trans. Softw. Eng.*, vol. 13, no. 2, pp. 222–232, Feb. 1987.

[11] Cisco, "Snort." [Online]. Available: https://www.snort.org/ Accessed on: 2017-07-06.

[12] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel, *Time Series Analysis*. Wiley-Blackwell, jun 2008.

[13] H. L. Van Trees, K. L. Bell, and Z. Tian, *Detection Estimation and Modulation Theory, Part I: Detection, Estimation, and Filtering Theory, 2nd Edition.* John Wiley & Sons, Apr. 2013.

[14] S. Axelsson, "A preliminary attempt to apply detection and estimation theory to intrusion detection," Dept. Comput. Eng., Chalmers Univ. Technol., Göteborg, Sweden, Tech. Rep. 00-4, 2000.

[15] J. P. Anderson, "Computer security threat monitoring and surveillance," James P. Anderson Co., Fort, Washington, PA, USA, Tech. Rep. 98–17, 1980.

[16] A. Patcha and J.-M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Comput. Networks*, vol. 51, no. 12, pp. 3448–3470, Aug. 2007.

[17] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *Comput. Secur.*, vol. 28, no. 1-2, pp. 18–28, Feb. 2009.

[18] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An overview of IP flow-based intrusion detection," *IEEE Commun. Surveys Tuts.*, vol. 12, no. 3, pp. 343–356, Apr. 2010.

[19] R. A. Maxion and F. E. Feather, "A case study of ethernet anomalies in a distributed computing environment," *IEEE Trans. Rel.*, vol. 39, no. 4, pp. 433–443, Oct. 1990.

[20] "Waikato internet traffic storage." [Online]. Available: https://wand.net.nz/wits/ Accessed on: 2017-07-06.

[21] J. D. Brutlag, "Aberrant behavior detection in time series for network monitoring," in *Proc. 14th USENIX Conf. Syst. Admin.*, 2000, pp. 139–146.

[22] P. Barford and D. Plonka, "Characteristics of network traffic flow anomalies," in *Proc. 1st ACM SIGCOMM Workshop Internet Measurement*, Nov. 2001, pp. 69–73.

[23] P. Barford, J. Kline, D. Plonka, and A. Ron, "A signal analysis of network traffic anomalies," in *Proc. 2nd ACM SIGCOMM Workshop Internet Measurment*, Nov. 2002, pp. 71–82.

[24] A. Lakhina, K. Papagiannaki, M. Crovella, C. Diot, E. D. Kolaczyk, and N. Taft, "Structural analysis of network traffic flows," in *Proc. Joint Int. Conf. Measurement and Modeling Comput. Syst.*, Jun. 2004, pp. 61–72.

[25] A. Lakhina, M. Crovella, and C. Diot, "Diagnosing network-wide traffic anomalies," in *Proc. 2004 Conf. Appl., Technol., Architectures, and Protocols Comput. Commun.*, Aug. 2004, pp. 219–230.

[26] ——, "Mining anomalies using traffic feature distributions," in *Proc. 2005 Conf. Appl., Technol., Architectures, and Protocols Comput. Commun.*, Aug. 2005, pp. 217–228.

[27] M. Crovella, "Mining low-dimensional network data," 2015. [Online]. Available: https://github.com/mcrovella/mining-low-dim-network-data Accessed on: 2017-12-16.

[28] ESnet, "iPerf – the ultimate speed test tool for TCP, UDP and SCTP." [Online]. Available: https://iperf.fr/ Accessed on: 2017-07-06.

[29] "Trinoo." [Online]. Available: https://staff.washington.edu/dittrich/misc/trinoo.analysis.txt Accessed on: 2017-07-06.

[30] R. Dahlhaus, "Locally stationary processes," in *Time Series Analysis: Methods and Applications*. Elsevier BV, 2012, pp. 351–413.

[31] G. Dewaele, K. Fukuda, P. Borgnat, P. Abry, and K. Cho, "Extracting hidden anomalies using sketch and non gaussian multiresolution statistical detection procedures," in *Proc. 2007 Workshop Large Scale Attack Defense*, Aug. 2007, pp. 145–152.

[32] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda, "MAWILab: Combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking," in *Proc. 6th Int. Conf. Emerging Networking Experiments and Technol.*, Nov. 2010.

[33] W. Leland, M. Taqqu, W. Willinger, and D. Wilson, "On the self-similar nature of ethernet traffic (extended version)," *IEEE/ACM Trans. Networking*, vol. 2, no. 1, pp. 1–15, 1994.

[34] K. Cho, K. Mitsuya, and A. Kato, "Traffic data repository at the WIDE project," in *Proc. USENIX Annu. Technical Conf.*, Jun. 2000. [Online]. Available: http://mawi.wide.ad.jp/mawi/

[35] Y. Kanda, R. Fontugne, K. Fukuda, and T. Sugawara, "ADMIRE: Anomaly detection method using entropy-based PCA with three-step sketches," *Comput. Commun.*, vol. 36, no. 5, pp. 575–588, 2013.

[36] R. Fontugne, P. Abry, K. Fukuda, D. Veitch, K. Cho, P. Borgnat, and H. Wendt, "Scaling in internet traffic: A 14 year and 3 day longitudinal study, with multiscale analyses and random projections," *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 2152–2165, Aug. 2017.

[37] J. Dromard, G. Roudière, and P. Owezarski, "Online and scalable unsupervised network anomaly detection method," *IEEE Trans. Netw. Service Manag.*, vol. 14, no. 1, pp. 34–47, Mar. 2017.

[38] J. Mazel, R. Fontugne, and K. Fukuda, "Profiling internet scanners: Spatiotemporal structures and measurement ethics," in *Network Traffic Measurement and Anal. Conf.*, Jun. 2017, pp. 1–9.

[39] Y. Himura, K. Fukuda, P. Abry, K. Cho, and H. Esaki, "Characterization of host-level application traffic with multi-scale gamma model," *IEICE Trans. Commun.*, vol. E93–B, no. 11, pp. 3048–3057, Nov. 2010.

[40] C. M. Bishop, "Novelty detection and neural network validation," in *Proc. IEEE Vision, Image and Signal Process.*, Aug. 1994, pp. 217–222.

[41] J. J. Santanna, R. van Rijswijk-Deij, R. Hofstede, A. Sperotto, M. Wierbosch, L. Z. Granville, and A. Pras, "Booters – an analysis of DDoS-as-a-service attacks," in *2015 IFIP/IEEE Int. Symp. Integrated Network Management*, May 2015, pp. 243–251.

[42] S. Liu, M. Yamada, N. Collier, and M. Sugiyama, "Change-point detection in time-series data by relative density-ratio estimation," in *Structural, Syntactic, and Statistical Pattern Recognition*, 2012, pp. 363–372.

[43] A. Hussain, J. Heidemann, and C. Papadopoulos, "A framework for classifying denial of service attacks," in *Proc. 2003 Conf. Appl., Technol., Architectures, and Protocols for Comput. Commun.*, Aug. 2003, pp. 99–110.

[44] M. Duarte, "BMC: Notes on scientific computing for biomechanics and motor control," 2017. [Online]. Available: https://github.com/demotu/BMC Accessed on: 2017-12-28.

Appendix A

Description of the Source Code

As part of the contributions of this thesis, a comprehensive Python library was developed for constructing and evaluating anomaly detection (or change detection) algorithms, called the `changedetect` library. This appendix guides the reader through the practical usage of the library. The complete source code is also included in Appendix B.

A.1 Constructing Detection Algorithms

The library is centred around the components discussed in Chapter 4. Each component is constructed as a Python object. The objects are described below, and demonstrated by way of example in the construction of Scherrer *et al.*'s [7] detection algorithm.

Sequences. A sequence (or series) is simply implemented as a numpy array. In the examples below, a given sequence is denoted by `seq`.

Windows. Sliding windows are implemented as *iterators*[1] that take a sequence as input and yield the sub-sequences observed within successive positions of the window. Window objects are adjustable in terms of their starting position, window length, and shift length (and could easily be extended to include window weights). A sliding window iterator is constructed by writing `window_iter = SlidingWindowIterator(win_start, win_len, win_shift)`

[1] An iterator is an object that performs operations on a list and returns results one at a time (called *yielding* the results), rather than processing the entire list in one go. This iterative method of evaluation is also referred to as *lazy* evaluation.

Thereafter, successive sub-sequences observed within a windowed sequence are processed using a simple loop:

```
for window in window_iter(seq):
    do_something(window.seq)
```

Representations. Sub-sequences returned from window iterators are then processed by representation objects to return new sequences. The following representations have been implemented:

- Temporal, which simply returns the original sequence.
- Gamma and alpha-stable probability distributions, which fit the respective probability distribution to the original sequence (using maximum likelihood estimation) and return the fitted parameters.
- Multi-Aggregated Temporal and Multi-Aggregated Gamma, which employ temporal and gamma representations at multiple aggregation levels.

For example, a multi-aggregated gamma representation of a given sequence may be initialized by `multi_gamma = MultiScaleGamma(agg_factor, agg_times)`, where the parameters, aggregation factor and aggregation times, are described in Section 6.3.1. Thereafter, the newly represented sequence is computed by `new_seq = multi_gamma(seq)`.

Deviations. Deviation objects take two sequences in appropriately chosen representations and return a measure of dissimilarity between them. The follow deviations have been implemented:

- L_2 Distance, which requires that the two inputs have the same number of rows and columns, treats each column of an array as a vector, and computes the distance between corresponding vectors.
- Log Likelihood, which requires a probability distribution to be specified, and takes fitted parameters of that distribution as input and computes the log likelihood function of a given sequence.

For example, the L_2 distance object is first initialized by `l2_dev = L2Distance()`. Thereafter, the deviation is computed between two sequences by `dev = l2_dev(seq1, seq2)`.

Detectors. All of the above components are interfaced via detector objects. One class of detectors is implemented, called local anomaly detectors. Localy anomaly detectors construct two contiguous and synchronous windows (called the reference and test windows) and compute a deviation between appropriately chosen representations of corresponding reference and test sub-sequences. Using the above building blocks, anomaly detection algorithms can easily be constructed. For example, Scherrer *et al.*'s algorithm [7] is constructed by the following code:

```
detector = LocalDetector(
    ref_iter = SlidingWindowIterator(ref_start, ref_len, win_shift),
    test_iter = SlidingWindowIterator(test_start, test_len, win_shift),
    ref_rep = MultiScaleGamma(agg_factor, agg_times),
    test_rep = MultiScaleGamma(agg_factor, agg_times),
    dev = L2Distance()
)
```

Once constructed, the detector object can process a given sequence, returning a sequence of deviation score values associated with distinct points in time: `scores = detect(seq, detector)`. The next section describes how the library evaluates the resulting deviation score sequence.

A.2 Evaluating Detection Algorithms

The output of a local anomaly detection algorithm is a sequence of deviation scores, indicating the likelihood of the breakdown of local stationarity at distinct points in time. Peaks in the deviation scores are therefore identified as potential *changepoints*; i.e., points in the original time series that may correspond to the start or end of an attack. A list of peaks in the score sequence is obtained by `predictions = detect_peaks(scores)`. Each peak in the returned list is then regarded as a prediction of the actual changepoints.

In order to evaluate a detection algorithm, it is required to compare the predicted changepoints to the actual changepoints. Thus, it is required to have a labelled time series; i.e., a time series with known attack start and end times. This is achieved by the function `changepoints, superposed_ts = generate_superposed(benign_ts, attack_ts, attack_start)`.

The function returns two variables: the known changepoints (the attack start time and the attack end time), and the superposed time series. The detection algorithm may then be evaluated by running the algorithm on the superposed time series, and comparing the predicted changepoints with the actual changepoints. To that end, all predicted changepoints falling within an interval of predefined length surrounding the actual changepoints are labelled as true positives, otherwise, they are labelled as false positives. The predicted changepoints are labelled by the function `labels = label_peaks(predictions, changepoints, interval_length)`. Finally, the collection of predictions and their labels is stored for later analysis.

The evaluation of each detector on each superposed time series is considered a single experiment, thus stored in a single row of a Pandas dataframe.[2] The dataframe allows for easier analysis of the numerical results as described in the next section.

A.3 Analysing the Results

From each collection of labelled predictions corresponding to distinct experiments, the Receiver Operating Characteristic (ROC) curve, and the Area Under the ROC Curve (AUC) are computed and stored in the rows of a new dataframe, denoted by `df`. The following code demonstrates the power of storing the results in a dataframe:

```
detectors = ['single_gamma_l20', 'single_gamma_l22']
seaborn.pointplot(data=df.loc[df['detector_name'].isin(detectors)],
                  x='attack_name',
                  y='tpr_pfa_0_0001',
                  hue='detector_name')
```

First, two detectors are selected (named `single_gamma_l20` and `single_gamma_l22`) then the true positive rate for a fixed probability of false alarm of 10^{-4} (named `tpr_pfa_0_0001`) is plotted on the y-axis versus the attack time series on the x-axis. Different detectors are coloured in different hues, thus producing the plot in Fig. A.1.

The powerful combination of the Seaborn library together with Pandas dataframes is further demonstrated by a final example. ROC curves for all experiments are plotted using only three lines of code:

[2] A dataframe is a data structure for working with tabular data that require named, two-dimensional indexing.

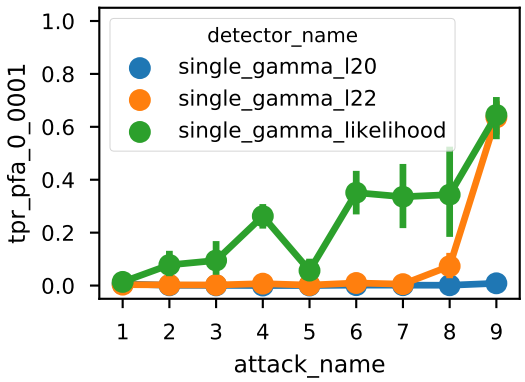


Fig. A.1. An example plot generated by source code in the text. The plot serves to demonstrate the output of the code (without any further refinements as in the rest of the thesis) and is not intended to be interpreted by the reader.

```
(seaborn.FacetGrid(df, row='benign_name', col='attack_name',
                   hue='detector_name')
        .map(plot_rec, 'fpr', 'tpr'))
```

The Seaborn package combined with the Pandas dataframe allows to create a grid of plots with rows corresponding to benign time series, columns corresponding to attack time series, and hues (or the colours of the line plots) corresponding to detectors. The false positive rate (or probability of false alarm) and true positive rate (or probability of detection) are 'mapped' to the `plot_roc` function, which plots the ROC curves in each cell of the grid. Since there were a total of nine attack time series and fourteen benign time series, the resulting number of ROC curves is difficult to interpret. Nevertheless, the ROC curves for the above code block are included in Fig. A.2 on the next page.

Fig. A.2. ROC curves generated by source code in the text. The plot (downscaled by 20%) serves to demonstrate the output of the code and is not intended to be interpreted by the reader.

Appendix B

Complete Source Code

This thesis strives to be completely open and reproducible. The author's intention is for this thesis to serve as part of a foundation for new researchers to more easily enter the domain and provide their own original research. To that end, the developed computer code is included in its entirety in this appendix, in addition to the guided description through the code in Appendix A.

B.1 Dependencies

The code used in this thesis was written in Python version 3.6.3 within the Anaconda environment version 4.4.10 (64 bit). In addition to the standard Python libraries, the following external libraries were used:

- Numpy version 1.13.3: A library for standard scientific computation tools, e.g., data structures implementing vectors and matrices, vector computations, and so on.
- Scipy version 1.0.0: Statistical methods, including probability density functions and parameter fitting mechanisms for standard probability distributions.
- Pylevy version 0.6: A library for evaluating and fitting the alpha-stable distribution.

- Pandas version 0.22.0: A library for manipulating, cleaning, exploring, and analysing medium-sized tabular datasets, which was used to analyse the results of the numerical experiments.
- Scikit-learn version 0.19.1: A library for implementing machine learning algorithms, though its use here has been limited to implementing classification accuracy metrics.
- Jupyter version 1.0.0: Most of the descriptive and graphical analyses here, including the generated figures, were developed in Jupyter notebooks.
- Matplotlib version 2.1.1: A low-level plotting library for visualising data.
- Seaborn version 0.8.1: A wrapper around Matplotlib for enhanced visualisation.
- Dpkt version 1.9.1: A library for reading and parsing network traffic data in the form of pcap files, used to extract time series data.

B.2 Source Code Contents

Each separate file of the code is included as a distinct Source Code block hereafter:

Source Code B.1. changedetect/sequence.py

```
"""
A sequence is just a 1D array or a vector. A multivariate sequence is just a 2D
array or a matrix. Each column of the matrix corresponds to a distinct
variable. Each row of the matrix corresponds to a point in time (or space).

seq.shape[0] : number of rows == number of sample points.
seq.shape[1] : number of cols == number of variables.

len(seq) == seq.shape[0] is _possibly_ a more interpretable form for the number
of sample points.
"""

from numpy import ndarray as Sequence
```

Source Code B.2. changedetect/window.py

```
"""
We process sequences by sequentially processing subsequences i.e. by windowing
in time.
"""
```

```
from typing import Iterator, NamedTuple

from changedetect.sequence import Sequence

Window = NamedTuple("Window", [("start", int), ("end", int), ("seq", Sequence)])

class WindowIterator:
    def __init__(self, start: int, length: int, shift: int) -> None:
        self.start = start
        self.length = length
        self.shift = shift

    def __call__(self, seq: Sequence) -> Iterator[Window]:
        raise NotImplementedError


class SlidingWindowIterator(WindowIterator):
    def __call__(self, seq: Sequence) -> Iterator[Window]:
        first_end = self.start + self.length
        last_end = len(seq) + 1
        ends = range(first_end, last_end, self.shift)

        for end in ends:
            start = end - self.length
            subseq = seq[start:end]

            yield Window(start, end, subseq)
```

Source Code B.3. changedetect/representation.py

```
"""
We try to represent sequences by compressed alternatives, e.g., parameters of
some model.
"""
import numpy as np
from scipy.stats import gamma
from pylevy.levy import fit_levy

from changedetect.sequence import Sequence
from changedetect.tools import multiagg

class Representation:
    def __call__(self, seq: Sequence) -> Sequence:
        raise NotImplementedError

class Temporal(Representation):
    def __call__(self, seq: Sequence) -> Sequence:
        return seq

class Gamma(Representation):
    def __call__(self, seq: Sequence) -> Sequence:
        return np.array(gamma.fit(seq, floc=0))

class Levy(Representation):
    def __call__(self, seq: Sequence) -> Sequence:
        return np.array(fit_levy(seq))
```

```
class MultiScaleTemporal(Representation):
    def __init__(self, factor: int, times: int) -> None:
        self.factor = factor
        self.times = times

    def __call__(self, seq: Sequence) -> Sequence:
        return np.array([x for x in multiagg(seq, self.factor, self.times)])

class MultiScaleGamma(Representation):
    def __init__(self, factor: int, times: int) -> None:
        self.factor = factor
        self.times = times

    def __call__(self, seq: Sequence) -> Sequence:
        return np.array([gamma.fit(x, floc=0)
                         for x in multiagg(seq, self.factor, self.times)])
```

Source Code B.4. changedetect/deviation.py

```
"""
A deviation can be calculated between two sequences or representations of them.
The same type of representation must be used on each sequence.
"""
import numpy as np
from scipy.spatial.distance import euclidean
from scipy.stats import rv_continuous as Distribution

from changedetect.sequence import Sequence

from pylevy.levy import neglog_levy

class Deviation:
    def __call__(self, seq1: Sequence, seq2: Sequence) -> Sequence:
        raise NotImplementedError

class L2Distance(Deviation):
    def __call__(self, seq1: Sequence, seq2: Sequence) -> Sequence:
        if seq1.shape != seq2.shape:
            raise ValueError("seq1 and seq2 must have the same shape.")

        return np.array([euclidean(col1, col2)
                         for col1, col2 in zip(seq1.T, seq2.T)])

class LogLikelihood(Deviation):
    def __init__(self, dist: Distribution) -> None:
        self.dist = dist

    def __call__(self, params: Sequence, seq: Sequence) -> Sequence:
        shape, loc, scale = params
        result = self.dist.logpdf(seq, shape, loc=loc, scale=scale)
        result = np.sum(result)
        return np.abs(result)

class LogLikelihoodLevy(Deviation):
    def __call__(self, params: Sequence, seq: Sequence) -> Sequence:
        print(params.shape, params)
        alpha, beta, mu, sigma, _ = params
```

```
        logpdf = neglog_levy(seq, alpha, beta, mu, sigma)
        deviation = np.sum(logpdf)
        return np.abs(deviation)

class MultiScaleLogLikelihood(Deviation):
    """Combiner is a function that takes a 1D array of deviations (likelihoods)
    and somehow combines them, e.g., maximum, product, sum, average."""
    def __init__(self, dist: Distribution, combiner) -> None:
        self.dist = dist
        self.combiner = combiner

    def __call__(self, multi_params: Sequence, multi_seq: Sequence) -> Sequence:
        deviations = []
        for params, seq in zip(multi_params, multi_seq):  # for each time scale
            shape, loc, scale = params
            logpdf = self.dist.logpdf(seq, shape, loc=loc, scale=scale)
            deviations.append(np.abs(logpdf.sum()))
        return self.combiner(deviations)
```

Source Code B.5. changedetect/detector.py

```
"""
Detectors are combinations of Window, Representation, Deviation, and Threshold.

A detector really is just a collection of these sub-objects.
The detector object says how we link together the sub-objects to perform
levels of processing of an original sequence.
E.g.,

             WindowIterator -> Representation
            /    (REF)                       \
/Sequence/ --                                 -> Deviation -> /Score/
            \                                 /
             WindowIterator -> Representation
                 (TEST)
"""
import numpy as np

from typing import Iterator, NamedTuple

from changedetect.sequence import Sequence
from changedetect.window import WindowIterator
from changedetect.representation import Representation
from changedetect.deviation import Deviation

Score = NamedTuple("Score", [("time", int), ("value", float)])
ScoreSequence = NamedTuple("ScoreSeq", [("times", Sequence), ("vals", Sequence)])

class Detector:
    def __call__(self, seq: Sequence) -> Iterator[Score]:
        raise NotImplementedError

class LocalDetector(Detector):
    def __init__(self,
                 ref_iter: WindowIterator,
                 test_iter: WindowIterator,
                 ref_rep: Representation,
```

```
                test_rep: Representation,
                dev: Deviation) -> None:
        self.ref_iter = ref_iter
        self.test_iter = test_iter
        self.ref_rep = ref_rep
        self.test_rep = test_rep
        self.dev = dev

    def __call__(self, seq: Sequence) -> Iterator[Score]:
        for ref, test in zip(self.ref_iter(seq), self.test_iter(seq)):
            ref_rep = self.ref_rep(ref.seq)
            test_rep = self.test_rep(test.seq)

            win_dev = self.dev(ref_rep, test_rep)

            yield Score(test.start, win_dev)

class LocalDetectorAttackInfo(Detector):
    def __init__(self,
                ref_iter: WindowIterator,
                test_iter: WindowIterator,
                ref_rep: Representation,
                test_rep: Representation,
                dev: Deviation,
                attack_ts: Sequence,
                ref_len: int,
                combiner) -> None:
        self.ref_iter = ref_iter
        self.test_iter = test_iter
        self.ref_rep = ref_rep
        self.test_rep = test_rep
        self.dev = dev
        self.attack_ts = attack_ts
        self.attack_ts_len = len(attack_ts)
        self.ref_len = ref_len
        self.combiner = combiner

    def __call__(self, seq: Sequence) -> Iterator[Score]:
        for ref_H0, test in zip(self.ref_iter(seq), self.test_iter(seq)):
            ref_H1_seq = ref_H0.seq + self.sample_attack_ts()

            ref_H0_rep = self.ref_rep(ref_H0.seq)
            ref_H1_rep = self.ref_rep(ref_H1_seq)

            test_rep = self.test_rep(test.seq)

            win_dev_H0 = self.dev(ref_H0_rep, test_rep)
            win_dev_H1 = self.dev(ref_H1_rep, test_rep)

            win_dev = self.combiner(win_dev_H0, win_dev_H1)

            yield Score(test.start, win_dev)

    def sample_attack_ts(self) -> Sequence:
        until = self.attack_ts_len - self.ref_len
        subset_start = np.random.randint(until)
        subset_end = subset_start + self.ref_len
        return self.attack_ts[subset_start:subset_end]
```

Source Code B.6. changedetect/evaluator.py

```
"""
First we need a labelled sequence generator.

Example 1. A basic Gaussian with mean-change simulator might take in the
parameters: mean, variance, changepoint, mean-change magnitude, and output
a tuple (sequence, changepoint).

Example 2. A superposition approach will take as input: benign sequence,
attack sequence, attack shift, and output a tuple (superposed sequence, attack
shift).

Then we need an evaluator, which will take as input: labelled sequence,
detector, and output the performance analysis/results.
Performance results, will include: the ROC curve and AUC.
"""
from typing import Dict, List, NamedTuple, Tuple

import numpy as np

from changedetect.sequence import Sequence
from changedetect.detector import Detector
from changedetect.detect_peaks import detect_peaks

# we must have len(benign_seq) == len(attack_seq)
LabelledSeq = NamedTuple("LabelledSeq",
                         [("changepoints", np.ndarray),
                          ("seq", Sequence)])

ScoreSequence = NamedTuple("ScoreSeq", [("times", Sequence), ("vals", Sequence)])
PeakSequence = NamedTuple('Peak', [('times', np.ndarray), ('vals', np.ndarray)])

def shift_array(arr, shift):
    result = np.empty_like(arr)
    if shift > 0:
        result[:shift] = 0
        result[shift:] = arr[:-shift]
    elif shift < 0:
        result[shift:] = 0
        result[:shift] = arr[-shift:]
    else:
        result = arr
    return result

def generate_superposed(benign_ts: Sequence,
                        attack_ts: Sequence,
                        attack_start: int) -> LabelledSeq:
    attack_len = len(attack_ts)
    pad_len = len(benign_ts) - attack_len
    attack_ts_padded = np.pad(attack_ts, (0, pad_len), 'constant')
    attack_ts_shifted = shift_array(attack_ts_padded, attack_start)
    superposed_ts = benign_ts + attack_ts_shifted

    changepoints = np.array([attack_start, attack_start + attack_len])

    return LabelledSeq(changepoints, superposed_ts)
```

```
def detect(seq: Sequence, detector: Detector) -> ScoreSequence:
    times_vals = [time_val for time_val in detector(seq)]
    times, vals = zip(*times_vals)
    times = np.array(times)
    vals = np.array(vals)

    return ScoreSequence(times, vals)

def separate_scores(scores: Dict) -> Dict:
    """Some detectors output multiple deviation scores. These should be treated
    as distinct detectors. Thus, we separate them here."""
    result = {}
    for name, (score_times, score_vals) in scores.items():
        if score_vals.ndim == 1:
            score_vals = score_vals.reshape((-1, 1))

        num_cols = score_vals.shape[1]

        for i, separate_score_vals in enumerate(score_vals.T):
            if num_cols == 1:
                key = name
            else:
                key = name + str(i)

            result[key] = ScoreSequence(score_times, separate_score_vals)
    return result

def multiple_detect(seq: Sequence, detectors: Dict):
    """Sometimes we want to perform detection on the same sequence with multiple
    detectors. This function does so, and separates multiple deviation scores
    from within the same detector if necessary."""
    combined_scores = {name: detect(seq, detector)
                       for name, detector in detectors.items()}
    return separate_scores(combined_scores)

def _detect_peaks(score: ScoreSequence) -> PeakSequence:
    """Find the peak times and values in a score sequence."""
    peak_idx = detect_peaks(score.vals, edge=None)
    peak_times = score.times[peak_idx]
    peak_vals = score.vals[peak_idx]

    return PeakSequence(peak_times, peak_vals)

def multiple_detect_peaks(scores: Dict) -> Dict:
    """From a dictionary of scores corresponding to multiple detectors,
    find the peak times and values for each detector score."""
    return {name: _detect_peaks(score) for name, score in scores.items()}

def label_peaks(peaks: PeakSequence,
                changepoints: np.ndarray,
                threshold: int) -> np.ndarray:
    """Label a list of peaks as 1 if they are sufficiently close to a
    changepoint or 0 otherwise."""
    peak_labels = np.zeros_like(peaks.times)
    for cp in changepoints:
        peak_labels |= (np.abs(peaks.times - cp) < threshold)
    return peak_labels
```

```
def multiple_label_peaks(peaks: PeakSequence, changepoints, threshold):
    """Label the peaks from a dictionary of scores corresponding to multiple
    detectors."""
    return {name: label_peaks(peak, changepoints, threshold)
            for name, peak in peaks.items()}
```

Source Code B.7. changedetect/tools.py

```
"""
Collection of tools for manipulating sequences.
"""
from typing import Any, Callable, Iterator, List

from itertools import accumulate, islice, repeat
from functools import partial

import numpy as np

from changedetect.sequence import Sequence

def pad(seq: Sequence, factor: int) -> Sequence:
    """
    Pad a sequence with zeroes until the length of the padded sequence is
    divisible by a given factor.
    """
    overflow_len = len(seq) % factor

    if overflow_len:
        pad_len = factor - overflow_len
        seq = np.pad(seq, (0, pad_len), 'constant')

    return seq

def clip(seq: Sequence, factor: int) -> Sequence:
    """
    Clips a sequence until its length is divisible by a given factor.
    """
    overflow_len = len(seq) % factor

    if overflow_len:
        seq = seq[:-overflow_len]

    return seq

def aggregate(seq: Sequence, factor: int) -> Sequence:
    """
    Split a sequence into sub-sequences of length factor, take the average of
    each sub-sequence, return the resulting 'aggregated' sequence.
    """
    if factor > len(seq):
        raise StopIteration

    result = clip(seq, factor)
    result = result.reshape(-1, factor)
    result = np.mean(result, axis=1)

    return result
```

```
def iterate(f: Callable, x: Any) -> Iterator:
    return accumulate(repeat(x), lambda fx, _: f(fx))

def multiagg(seq: Sequence, factor: int, times: int) -> Iterator[Sequence]:
    aggregate_factor = partial(aggregate, factor=factor)
    aggregate_iter = iterate(aggregate_factor, seq)
    result = islice(aggregate_iter, times)
    return result
```

Source Code B.8. changedetect/detect_peaks.py [44]

```
# %load ./../functions/detect_peaks.py
"""Detect peaks in data based on their amplitude and other features."""

from __future__ import division, print_function
import numpy as np

__author__ = "Marcos␣Duarte,␣https://github.com/demotu/BMC"
__version__ = "1.0.4"
__license__ = "MIT"


def detect_peaks(x, mph=None, mpd=1, threshold=0, edge='rising',
                 kpsh=False, valley=False, show=False, ax=None):

    """Detect peaks in data based on their amplitude and other features.

    Parameters
    ----------
    x : 1D array_like
        data.
    mph : {None, number}, optional (default = None)
        detect peaks that are greater than minimum peak height.
    mpd : positive integer, optional (default = 1)
        detect peaks that are at least separated by minimum peak distance (in
        number of data).
    threshold : positive number, optional (default = 0)
        detect peaks (valleys) that are greater (smaller) than `threshold`
        in relation to their immediate neighbors.
    edge : {None, 'rising', 'falling', 'both'}, optional (default = 'rising')
        for a flat peak, keep only the rising edge ('rising'), only the
        falling edge ('falling'), both edges ('both'), or don't detect a
        flat peak (None).
    kpsh : bool, optional (default = False)
        keep peaks with same height even if they are closer than `mpd`.
    valley : bool, optional (default = False)
        if True (1), detect valleys (local minima) instead of peaks.
    show : bool, optional (default = False)
        if True (1), plot data in matplotlib figure.
    ax : a matplotlib.axes.Axes instance, optional (default = None).

    Returns
    -------
    ind : 1D array_like
        indeces of the peaks in `x`.
```

```
    Notes
    -----
    The detection of valleys instead of peaks is performed internally by simply
    negating the data: `ind_valleys = detect_peaks(-x)`

    The function can handle NaN's

    See this IPython Notebook [1]_.

    References
    ----------
    .. [1] http://nbviewer.ipython.org/github/demotu/BMC/blob/master/ +
               notebooks/DetectPeaks.ipynb
    Examples
    --------
    >>> from detect_peaks import detect_peaks
    >>> x = np.random.randn(100)
    >>> x[60:81] = np.nan
    >>> # detect all peaks and plot data
    >>> ind = detect_peaks(x, show=True)
    >>> print(ind)

    >>> x = np.sin(2*np.pi*5*np.linspace(0, 1, 200)) + np.random.randn(200)/5
    >>> # set minimum peak height = 0 and minimum peak distance = 20
    >>> detect_peaks(x, mph=0, mpd=20, show=True)

    >>> x = [0, 1, 0, 2, 0, 3, 0, 2, 0, 1, 0]
    >>> # set minimum peak distance = 2
    >>> detect_peaks(x, mpd=2, show=True)

    >>> x = np.sin(2*np.pi*5*np.linspace(0, 1, 200)) + np.random.randn(200)/5
    >>> # detection of valleys instead of peaks
    >>> detect_peaks(x, mph=0, mpd=20, valley=True, show=True)

    >>> x = [0, 1, 1, 0, 1, 1, 0]
    >>> # detect both edges
    >>> detect_peaks(x, edge='both', show=True)

    >>> x = [-2, 1, -2, 2, 1, 1, 3, 0]
    >>> # set threshold = 2
    >>> detect_peaks(x, threshold = 2, show=True)
    """

    x = np.atleast_1d(x).astype('float64')
    if x.size < 3:
        return np.array([], dtype=int)
    if valley:
        x = -x
    # find indices of all peaks
    dx = x[1:] - x[:-1]
    # handle NaN's
    indnan = np.where(np.isnan(x))[0]
    if indnan.size:
        x[indnan] = np.inf
        dx[np.where(np.isnan(dx))[0]] = np.inf
    ine, ire, ife = np.array([[], [], []], dtype=int)
    if not edge:
        ine = np.where((np.hstack((dx, 0)) < 0) & (np.hstack((0, dx)) > 0))[0]
```

```
    else:
        if edge.lower() in ['rising', 'both']:
            ire = np.where((np.hstack((dx, 0)) <= 0) &
                           (np.hstack((0, dx)) > 0))[0]
        if edge.lower() in ['falling', 'both']:
            ife = np.where((np.hstack((dx, 0)) < 0) &
                           (np.hstack((0, dx)) >= 0))[0]
    ind = np.unique(np.hstack((ine, ire, ife)))
    # handle NaN's
    if ind.size and indnan.size:
        # NaN's and values close to NaN's cannot be peaks
        ind = ind[np.in1d(ind, np.unique(np.hstack(
            (indnan, indnan-1, indnan+1))), invert=True)]
    # first and last values of x cannot be peaks
    if ind.size and ind[0] == 0:
        ind = ind[1:]
    if ind.size and ind[-1] == x.size-1:
        ind = ind[:-1]
    # remove peaks < minimum peak height
    if ind.size and mph is not None:
        ind = ind[x[ind] >= mph]
    # remove peaks - neighbors < threshold
    if ind.size and threshold > 0:
        dx = np.min(np.vstack([x[ind]-x[ind-1], x[ind]-x[ind+1]]), axis=0)
        ind = np.delete(ind, np.where(dx < threshold)[0])
    # detect small peaks closer than minimum peak distance
    if ind.size and mpd > 1:
        ind = ind[np.argsort(x[ind])][::-1]  # sort ind by peak height
        idel = np.zeros(ind.size, dtype=bool)
        for i in range(ind.size):
            if not idel[i]:
                # keep peaks with the same height if kpsh is True
                idel = idel | (ind >= ind[i] - mpd) & (ind <= ind[i] + mpd) \
                    & (x[ind[i]] > x[ind] if kpsh else True)
                idel[i] = 0  # Keep current peak
        # remove the small peaks and sort back the indices by their occurrence
        ind = np.sort(ind[~idel])

    if show:
        if indnan.size:
            x[indnan] = np.nan
        if valley:
            x = -x
        _plot(x, mph, mpd, threshold, edge, valley, ax, ind)

    return ind


def _plot(x, mph, mpd, threshold, edge, valley, ax, ind):
    """Plot results of the detect_peaks function, see its help."""
    try:
        import matplotlib.pyplot as plt
    except ImportError:
        print('matplotlib is not available.')
    else:
        if ax is None:
            _, ax = plt.subplots(1, 1, figsize=(8, 4))
```

```
        ax.plot(x, 'b', lw=1)
        if ind.size:
            label = 'valley' if valley else 'peak'
            label = label + 's' if ind.size > 1 else label
            ax.plot(ind, x[ind], '+', mfc=None, mec='r', mew=2, ms=8,
                    label='%d %s' % (ind.size, label))
            ax.legend(loc='best', framealpha=.5, numpoints=1)
        ax.set_xlim(-.02*x.size, x.size*1.02-1)
        ymin, ymax = x[np.isfinite(x)].min(), x[np.isfinite(x)].max()
        yrange = ymax - ymin if ymax > ymin else 1
        ax.set_ylim(ymin - 0.1*yrange, ymax + 0.1*yrange)
        ax.set_xlabel('Data #', fontsize=14)
        ax.set_ylabel('Amplitude', fontsize=14)
        mode = 'Valley detection' if valley else 'Peak detection'
        ax.set_title("%s (mph=%s, mpd=%d, threshold=%s, edge='%s')"
                     % (mode, str(mph), mpd, str(threshold), edge))
        # plt.grid()
        plt.show()
```

Source Code B.9. run_experiments.py

```
"""
Run the experiments for each unexplored algorithm and pickle the results.

v4: * Fixed multi_gamma_likelihood()
v3: * Refactored the script to make it a bit more reuseable.
v2: * Changed from np.genfromtxt() to np.load(), had to convert csvs to npys.
      Can expect ~500 times speedup in loading parts.
    * Added an if statement to check for cases where not a single changepoint
      is correctly found. If so, fpr tpr threshold auc are all set to NaN.
"""

import os
from time import time
from glob import glob

import numpy as np
import pandas as pd
from scipy import stats

from changedetect.window import SlidingWindowIterator
from changedetect.representation import (
    Gamma,
    MultiScaleGamma,
    Temporal,
    MultiScaleTemporal
)
from changedetect.deviation import (
    L2Distance,
    LogLikelihood,
    MultiScaleLogLikelihood
)
from changedetect.detector import LocalDetector
from changedetect.evaluator import (
    generate_superposed,
    multiple_detect,
    multiple_detect_peaks,
```

```
    multiple_label_peaks
)

def run_single_gamma_l2():
    global ref_len, test_len, win_shift, agg_factor, agg_times, cp_interval, \
           benign_filepaths, attack_filepaths, freq, num_attack_starts

    detector_name = 'single_gamma_l2'
    detector_params = {'ref_len': ref_len,
                       'test_len': test_len,
                       'win_shift': win_shift,
                       'cp_interval': cp_interval}

    detector = LocalDetector(
        ref_iter = SlidingWindowIterator(0, ref_len, win_shift),
        test_iter = SlidingWindowIterator(ref_len, test_len, win_shift),
        ref_rep = Gamma(),
        test_rep = Gamma(),
        dev = L2Distance()
    )

    detector_dict = {detector_name: detector}

    run_experiments(detector_name, detector_dict, detector_params)


def run_single_gamma_likelihood():
    global ref_len, test_len, win_shift, agg_factor, agg_times, cp_interval, \
           benign_filepaths, attack_filepaths, freq, num_attack_starts

    detector_name = 'single_gamma_likelihood'
    detector_params = {'ref_len': ref_len,
                       'test_len': test_len,
                       'win_shift': win_shift,
                       'cp_interval': cp_interval}

    detector = LocalDetector(
        ref_iter = SlidingWindowIterator(0, ref_len, win_shift),
        test_iter = SlidingWindowIterator(ref_len, test_len, win_shift),
        ref_rep = Gamma(),
        test_rep = Temporal(),
        dev = LogLikelihood(stats.gamma)
    )

    detector_dict = {detector_name: detector}

    run_experiments(detector_name, detector_dict, detector_params)


def run_multi_gamma_l2():
    global ref_len, test_len, win_shift, agg_factor, agg_times, cp_interval, \
           benign_filepaths, attack_filepaths, freq, num_attack_starts

    detector_name = 'multi_gamma_l2'
    detector_params = {'ref_len': ref_len,
                       'test_len': test_len,
                       'win_shift': win_shift,
                       'agg_factor': agg_factor,
```

```
                      'agg_times': agg_times,
                      'cp_interval': cp_interval}

    detector = LocalDetector(
        ref_iter = SlidingWindowIterator(0, ref_len, win_shift),
        test_iter = SlidingWindowIterator(ref_len, test_len, win_shift),
        ref_rep = MultiScaleGamma(agg_factor, agg_times),
        test_rep = MultiScaleGamma(agg_factor, agg_times),
        dev = L2Distance()
    )

    detector_dict = {detector_name: detector}

    run_experiments(detector_name, detector_dict, detector_params)


def run_multi_gamma_likelihood():
    global ref_len, test_len, win_shift, agg_factor, agg_times, cp_interval, \
           benign_filepaths, attack_filepaths, freq, num_attack_starts

    detector_name = 'multi_gamma_likelihood'
    detector_params = {'ref_len': ref_len,
                       'test_len': test_len,
                       'win_shift': win_shift,
                       'agg_factor': agg_factor,
                       'agg_times': agg_times,
                       'cp_interval': cp_interval}

    detector = LocalDetector(
        ref_iter = SlidingWindowIterator(0, ref_len, win_shift),
        test_iter = SlidingWindowIterator(ref_len, test_len, win_shift),
        ref_rep = MultiScaleGamma(agg_factor, agg_times),
        test_rep = MultiScaleTemporal(agg_factor, agg_times),
        dev = MultiScaleLogLikelihood(stats.gamma, np.sum)
    )

    detector_dict = {detector_name: detector}

    run_experiments(detector_name, detector_dict, detector_params)


# Run experiments
# TODO: There's definitely a better way to structure this. The detector object
#       should carry information like its name and params.
def run_experiments(detector_name, detectors, detector_params):
    global benign_filepaths, attack_filepaths, freq, num_attack_starts

    num_benign = len(benign_filepaths)
    num_attack = len(attack_filepaths)

    ref_len = detector_params['ref_len']
    test_len = detector_params['test_len']
    cp_interval = detector_params['cp_interval']

    col_names = ['benign_name',
                 'attack_name',
                 'attack_start',
                 'attack_end',
```

```
                'threshold',
                'detector_name',
                'detector_params',
                'score_times',
                'score_vals',
                'peak_times',
                'peak_vals',
                'peak_labels']

    data = []

    start_time = time()

    for i, benign_filepath in enumerate(benign_filepaths):

        # Load benign time series
        benign_ts = np.load(benign_filepath)
        benign_name = os.path.basename(benign_filepath).split('.')[0]

        for j, attack_filepath in enumerate(attack_filepaths):

            # Load attack time series
            attack_ts = np.load(attack_filepath)
            # Take first 100 seconds of attack as in Booters paper.
            attack_ts = attack_ts[:100 * freq]
            attack_name = os.path.basename(attack_filepath).split('.')[0]

            # Calculate evenly spaced attack start times
            attack_starts = (np.linspace(ref_len + test_len,
                                         (len(benign_ts) - len(attack_ts)
                                          - ref_len - test_len),
                                         num_attack_starts)
                             .astype(int))

            for k, attack_start in enumerate(attack_starts):

                # Generate labelled time series
                changepoints, superposed_ts = generate_superposed(
                    benign_ts, attack_ts, attack_start)

                # Run detectors
                scores = multiple_detect(superposed_ts, detectors)

                # Detect peaks in scores
                peaks = multiple_detect_peaks(scores)

                # Label peaks
                labels = multiple_label_peaks(peaks, changepoints, cp_interval)

                # Append to the data array
                for name in scores:
                    score_times, score_vals = scores[name]
                    peak_times, peak_vals = peaks[name]
                    peak_labels = labels[name]

                    data.append((benign_name,
                                 attack_name,
                                 changepoints[0],
```

```
                                changepoints[1],
                                cp_interval,
                                name,
                                detector_params,
                                score_times,
                                score_vals,
                                peak_times,
                                peak_vals,
                                peak_labels))

                print(detector_name,
                      f'[{i+1}/{num_benign}]', benign_name,
                      f'[{j+1}/{num_attack}]', attack_name,
                      f'[{k+1}/{num_attack_starts}]', attack_start)

    print('Creating dataframe.')
    df = pd.DataFrame(data=data, columns=col_names)

    print('Writing to file.')
    try:
        os.mkdir('output')
    except FileExistsError:
        pass
    df.to_pickle('output\\' + detector_name + '.pkl')

    duration = time() - start_time
    print(f'Finished: {duration}')


if __name__ == '__main__':
    # Dataset parameters - remain the same for all detectors.
    benign_filepaths = glob('..\\..\\Datasets\\MAWI\\npy\\*.npy')
    attack_filepaths = glob('..\\..\\Datasets\\Booters\\npy\\*.npy')

    freq = 1000
    num_attack_starts = 1000

    # Detector params
    # General
    ref_len = 60 * freq       # = 1 min.  Scherrer et al. use 1 min and 10 min.
    test_len = 10 * freq      # = 10 sec.  Scherrer et al. use 1 min.
    win_shift = 10 * freq     # = 10 sec.  Scherrer et al. use 1 min.

    # Multi-aggregation
    agg_factor = 2            # Scherrer et al. & Dewaele et al. use 2.
    agg_times = 7             # Scherrer et al. use 10, Dewaele et al. use 7.
    cp_interval = 5 * freq    # = 5 sec.

    # Run detectors
    run_single_gamma_l2()
    run_single_gamma_likelihood()
    run_multi_gamma_l2()             # Scherrer et al.'s detector
    run_multi_gamma_likelihood()
```

Source Code B.10. compute_roc.py

```
"""
```

```
Compute ROC curves from the experimental results data.
"""

import numpy as np
import pandas as pd

from sklearn import metrics

from sys import argv


def compute_roc_df(filepath):
    """Compute the ROC curves for each (detector, benign_ts, attack_ts)
    combination in a dataframe, and write the results to a new dataframe."""
    df = pd.read_pickle(filepath)

    grouped = df.groupby(['benign_name', 'attack_name', 'detector_name'])

    columns = ['benign_name', 'attack_name', 'detector_name', 'peak_vals',
               'peak_labels', 'fpr', 'tpr', 'thresholds', 'auc']
    data = []
    for name, group in grouped:
        peak_vals = np.concatenate([x for x in group.peak_vals])
        peak_labels = np.concatenate([x for x in group.peak_labels])

        if 1 not in peak_labels:
            fpr, tpr, thresholds = np.nan, np.nan, np.nan
            auc = np.nan
        else:
            fpr, tpr, thresholds = metrics.roc_curve(peak_labels, peak_vals,
                                                     drop_intermediate=True)
            auc = metrics.auc(fpr, tpr)

        data.append(list(name) + [peak_vals, peak_labels, fpr, tpr, thresholds,
                                  auc])

    rocdf = pd.DataFrame(data=data, columns=columns)

    rocdf.to_pickle(filepath[:-4] + '_roc.pkl')


if __name__ == '__main__':
    if len(argv) == 2:
        compute_roc_df(argv[1])
    else:
        print('Inappropriate argument.')
```

Source Code B.11. extract_timeseries.py

```
"""
Extract time series from all pcaps in a given directory.

@author: Wasim
"""

import time
import glob
```

```
from datetime import datetime
from datetime import timedelta

import dpkt


def extract_timeseries(packets_pcap_filename, timeseries_csv_filename,
                       interval_length):
    """Extracts time series of packet counts from pcap file.

    Had to resort to datetime module due to an accumulation of floating
    point errors each time the next window_end time was calculated, if
    we worked directly on the packet_timestamp floats.

    Args:
        packets_pcap_filename (string): Path to a packet trace pcap file.
        timeseries_csv_filename (string): Path to the output time series csv file.
        interval_length (int): Sample interval length in milliseconds.
    """
    timer = time.time()
    with open(packets_pcap_filename, 'rb') as packets_pcap_file, \
         open(timeseries_csv_filename, 'w') as timeseries_csv_file:

        packets_pcap_reader = dpkt.pcap.Reader(packets_pcap_file)

        # Convert interval_length to timedelta
        interval_length = timedelta(milliseconds=interval_length)

        # Read first packet for starting time, convert to datetime
        start_time, _ = next(packets_pcap_reader)
        start_time = datetime.utcfromtimestamp(start_time)

        # First window starts at first timestamp
        window_end = start_time + interval_length
        count = 1   # Count starts at 1 since we've already read a packet

        # Loop through remaining packets to construct time series
        for packet_timestamp, _ in packets_pcap_reader:
            packet_timestamp = datetime.utcfromtimestamp(packet_timestamp)
            if packet_timestamp < window_end:
                # Packet is in the current window, increase current window count
                count += 1
            else:
                # First packet of the next window, write current window count
                # to file.
                timeseries_csv_file.write("{}\n".format(count))
                # Go to the next window
                window_end += interval_length
                count = 1

    print(f"Finished: {time.time() - timer}")

    # Want to also write some metadata to file:
    # Start date and time (nice format)
    # End date and time (nice format)
    # Interval length (ms)
    # Number of windows
```

```
    with open(timeseries_csv_filename[:-3] + 'meta', 'w') as metadata_file:
        metadata_file.write('Start time: {}\n'.format(start_time.isoformat()))
        metadata_file.write('End time: {}\n'.format(packet_timestamp.isoformat()))
        metadata_file.write('Window length: {}\n'.format(interval_length))


if __name__ == "__main__":
    PATH = 'C:\\WAIKATO_TRACES\\pcap\\*.pcap'
    for pcapname in glob.glob(PATH):
        csvname = pcapname[:-4] + 'csv'
        print(f'Extracting time series from {pcapname} to {csvname}')
        extract_timeseries(pcapname, csvname, 1)
```