Detecting Worm Mutations Using Machine Learning

Oliver Sharma

November 2008

Submitted in fulfilment of the requirements for
the degree of DOCTOR OF PHILOSOPHY

Department of Computing Science
University of Glasgow
United Kingdom

This dissertation is dedicated to my parents.

Abstract

Worms are malicious programs that spread over the Internet without human intervention. Since worms generally spread faster than humans can respond, the only viable defence is to automate their detection.

Network intrusion detection systems typically detect worms by examining packet or flow logs for known signatures. Not only does this approach mean that new worms cannot be detected until the corresponding signatures are created, but that mutations of known worms will remain undetected because each mutation will usually have a different signature. The intuitive and seemingly most effective solution is to write more generic signatures, but this has been found to increase false alarm rates and is thus impractical.

This dissertation investigates the feasibility of using machine learning to automatically detect mutations of known worms. First, it investigates whether Support Vector Machines can detect mutations of known worms. Support Vector Machines have been shown to be well suited to pattern recognition tasks such as text categorisation and hand-written digit recognition. Since detecting worms is effectively a pattern recognition problem, this work investigates how well Support Vector Machines perform at this task.

The second part of this dissertation compares Support Vector Machines to other machine learning techniques in detecting worm mutations. Gaussian Processes, unlike Support Vector Machines, automatically return confidence values as part of their result. Since confidence values can be used to reduce false alarm rates, this dissertation determines how Gaussian Process compare to Support Vector Machines in terms of detection accuracy. For further comparison, this work also compares Support Vector Machines to K-nearest neighbours, known for its simplicity and solid results in other domains.

The third part of this dissertation investigates the automatic generation of training data. Classifier accuracy depends on good quality training data – the wider the training data spectrum, the higher the classifier's accuracy. This dissertation describes the design and implementation of a worm mutation generator whose output is fed to the machine learning techniques as training data. This dissertation then evaluates whether the training data can be used to train classifiers of sufficiently high quality to detect worm mutations.

The findings of this work demonstrate that Support Vector Machines can be used to detect worm mutations, and that the optimal configuration for detection of worm mutations is to use a linear kernel with unnormalised bi-gram frequency counts. Moreover, the results show that Gaussian Processes and Support Vector Machines exhibit similar accuracy on average in detecting worm mutations, while K-nearest neighbours consistently produces lower quality predictions. The generated worm mutations are shown to be of sufficiently high quality to serve as training data. Combined, the results demonstrate that machine learning is capable of accurately detecting mutations of known worms.

Acknowledgements

First of all I would like to thank my supervisor Professor Joseph Sventek for all his trust, support and guidance over the last years. I have benefited greatly from his experience and wisdom.

I would also like to thank my second supervisor Professor Mark Girolami for sharing his insights and expertise, particularly in the field of Machine Learning. Thank you also to Professor Roderick Murray-Smith for his helpful comments during my annual progress vivas.

I am greatly indebted to my friend and former colleague Dr Olufemi Komolafe, who was always kind enough to give feedback and advice.

For their continuous support and encouragement, I am eternally grateful to all my family, especially my parents, my brother Alexander, my grandmother, and Laura.

Last but not least, the daily conversations with colleagues were a constant source of inspiration. Special thanks to Steven Heeps, Alexandros Koliousis, Ross McIlroy, Stephen Strowes, and Zhan Xiaoying.

Contents

Abstract **ii**

Acknowledgments **iv**

1 Introduction **1**

1.1 The worm problem 1

1.2 Thesis statement 4

1.3 Contributions 5

1.4 Publications 5

1.5 Outline 6

2 Worms and Intrusion Detection **8**

2.1 Worms 8

2.1.1 Challenges in defending against worms 9

2.1.2 Worm attack strategies 10

2.1.3 A brief case study of worms 11

2.2 Intrusion detection systems 17

2.2.1 Approaches to intrusion detection 17

2.2.2 Case studies of intrusion detection systems 18

2.3 Summary 23

3 Machine learning **26**

3.1 Introduction 26

3.1.1 Why machine learning? 27

3.1.2 Supervised vs. unsupervised learning 27

3.1.3 Parametric vs. non-parametric methods 28
3.2 Support Vector Machines 29
3.2.1 How Support Vector Machines work 30
3.2.2 An example 31
3.2.3 Non-linearly separable data and the kernel trick 33
3.2.4 Support Vector Machines revisited 35
3.3 Gaussian Processes 36
3.3.1 Why Gaussian Processes? 37
3.3.2 How Gaussian Processes work 38
3.3.3 From regression to classification 41
3.3.4 Kernel functions 42
3.4 K-nearest neighbours 42
3.4.1 Properties 44
3.5 Other machine learning techniques 44
3.6 Summary 46

4 Applying ML to worm detection 47
4.1 Worm model 47
4.2 Architecture 50
4.3 Capturing network traffic 50
4.3.1 A sample implementation 51
4.4 Feature extraction 53
4.4.1 Why features must be extracted 53
4.4.2 Implementing n-gram extraction 55
4.5 Machine learning classifiers 56
4.6 Architecture revisited 57
4.7 Deployment 58
4.8 Summary 59

5 SVMs for worm detection 61
5.1 Experiment design 61
5.1.1 Synthetic mutations 62
5.1.2 Kernel configuration 65

5.1.3 Feature extraction . . . 67
5.1.4 Training data size . . . 67
5.1.5 Data-to-signature ratio . . . 67
5.1.6 False alarm rates . . . 68
5.2 Experimental setup . . . 69
5.3 Analysis . . . 70
5.3.1 Synthetic mutation format . . . 70
5.3.2 Choosing the optimal kernel . . . 71
5.3.3 Feature extraction . . . 76
5.3.4 Training data size . . . 78
5.3.5 Continuous, split, and jumbled signatures . . . 79
5.3.6 Corrupted signatures . . . 80
5.3.7 Mixed data-to-signature ratios . . . 81
5.3.8 False alarms . . . 82
5.4 Summary . . . 82

6 Alternative machine learning methods 84
6.1 Experiment design . . . 85
6.1.1 Support Vector Machines . . . 85
6.1.2 Gaussian Processes . . . 86
6.1.3 K-nearest neighbours . . . 87
6.1.4 Combining classifiers . . . 88
6.2 Analysis . . . 88
6.2.1 Optimal number of neighbours in KNN . . . 88
6.2.2 Comparison of machine learning methods . . . 90
6.2.3 Combining classifiers . . . 96
6.3 Summary . . . 97

7 Generating worm mutations 100
7.1 Structurally mutated worms . . . 100
7.1.1 Distcc worm . . . 101
7.1.2 Minishare worm . . . 102
7.1.3 WarFTP worm . . . 103

7.2 Randomly mutated worms . . . 104
7.2.1 Requirements . . . 104
7.2.2 Architecture . . . 105
7.3 Experiment design . . . 107
7.3.1 Isolated testing network . . . 108
7.3.2 Implementation of a worm testing framework . . . 108
7.3.3 Structurally mutated worms . . . 110
7.3.4 Randomly mutated worms . . . 111
7.4 Analysis . . . 112
7.4.1 Effectiveness of structurally mutated worms . . . 112
7.4.2 Effectiveness of randomly mutated worms . . . 115
7.5 Summary . . . 120

8 Conclusion **122**
8.1 Discussion . . . 122
8.2 Contributions . . . 125

9 Future work **126**
9.1 Sophisticated worm attack strategies . . . 126
9.1.1 Polymorphic blending attack . . . 127
9.1.2 Zero day vulnerabilities . . . 127
9.1.3 Poisoning of benign traffic . . . 128
9.2 Designing a custom kernel . . . 128
9.3 Alternative feature representation . . . 129
9.4 Real-time detection . . . 130
9.4.1 Optimising flow reassembly . . . 130
9.4.2 Partial flow classification . . . 132
9.4.3 Dedicated hardware . . . 133
9.4.4 Reducing the memory footprint . . . 133
9.5 Online training . . . 134
9.6 Cascading classifiers . . . 135
9.6.1 Manual analysis . . . 135
9.6.2 Refined machine learning analysis . . . 136

9.7 Scalability 137
9.8 Summary 138

A Formulas of Support Vector Machines **140**

B ROC Graphs **142**
B.1 ROC graphs 142
B.2 ROC curves 144
B.3 Area under curve 145

Glossary **146**

Bibliography **150**

List of Figures

1.1 Architecture to detect worms with machine learning 3

3.1 Exemplar separating hyperplane of Support Vector Machines . 30
3.2 More than one possible separating hyperplanes 31
3.3 Support vectors and the optimal separating hyperplane 32
3.4 The optimal hyperplane 33
3.5 How SVMs deal with linearly inseparable data 34
3.6 Support Vector Machines cycle 36
3.7 Overlaying the prior with training data yields the posterior . . 39
3.8 Conceptual representation of a Gaussian Process 40
3.9 Response functions in Gaussian Processes 41
3.10 KNN classifiers predict by finding the K nearest points. 43

4.1 A simple worm model 48
4.2 Worm mutations vs. different types of a worm 49
4.3 Machine learning enabled worm detector 50
4.4 Flow table 52
4.5 The flow data-structure 53
4.6 Feature vectors for *ELVIS* and *LIVES* 54
4.7 Feature extraction using a sliding window 56
4.8 Training classifiers to distinguish between flows 57
4.9 Training and classification stages 58
4.10 IDSs are typically deployed behind firewalls 59

5.1 Synthetic flows 62

5.2 Synthetic worms loosely based on the Slapper worm. 63
5.3 Continuous, split, and jumbled signatures 64
5.4 Comparing random vs. SSL padded flows 70
5.5 Configuring the linear kernel 71
5.6 Configuring the RBF kernel 72
5.7 Configuring the string kernel 73
5.8 Prediction accuracy of the linear, RBF, and string kernel. . . . 74
5.9 Training time for the linear, RBF, and string kernel. 75
5.10 Prediction time of the linear, RBF, and string kernel. 75
5.11 Comparing uni-grams, bi-grams, and tri-grams. 77
5.12 Normalised vs. unnormalised features 77
5.13 Impact of training data sizes 78
5.14 Resilience to continuous, split and jumbled signatures 79
5.15 Resilience to signature corruption 80
5.16 Mixed data-to-signature ratios 81
5.17 Trade-off between true and false positive rates 82

6.1 Number of neighbours in KNN 89
6.2 Number of neighbours vs. Prediction time 91
6.3 Comparing the accuracy of SVMs, GPs, and KNN 92
6.4 Accuracy vs. Training data size 93
6.5 GP predictive likelihood 94
6.6 Prediction confidence at various training data sizes. 95
6.7 Mixed flow lengths 96
6.8 ROC graph comparing SVM, GP, KNN 97
6.9 Combining multiple classifiers 98

7.1 Worm mutation generator architecture 106
7.2 Gauging the mutation degree 107
7.3 Worm testing framework 109
7.4 Accuracy of structurally mutated Distcc worm 112
7.5 Accuracy of structurally mutated Minishare worm 113
7.6 Accuracy of structurally mutated WarFTP worm 114

7.7 Accuracy of randomly mutated Distcc worm 116
7.8 Accuracy of randomly mutated Minishare worm 117
7.9 Accuracy of randomly mutated WarFTP worm 118
7.10 Impact of mutation chunk size on accuracy 119
7.11 Impact of mutation probability on accuracy 120

9.1 Real-time flow reassembly using a circular buffer 131
9.2 Partial flow classification 132
9.3 Manual analysis in uncertain cases 135
9.4 Adding new training data in manual analysis 136

B.1 Receiver operator characteristics graph space 143
B.2 ROC curve performances 144

List of Tables

5.1 Comparison of linear, RBF, and string kernel 66
5.2 Approximate data-to-signature ratios for worms 68

6.1 Comparison of SVMs, GPs, and KNNs 86
6.2 Tuned number of neighbours for KNN 90

Chapter 1

Introduction

1.1 The worm problem

Worms are malicious programs that spread over the Internet without human intervention [1, 2]. Like some biological viruses they infect hosts through known weaknesses, cause what damage they can, and then use the host as a springboard to find other vulnerable victims.

The first Internet worm was unleashed in 1988 and brought down hundreds of machines across the USA [3], at the time a significant portion of the early Internet. Worms proliferated as the Internet matured into a global network, wreaking havoc and causing considerable financial damage [4].

None of the damage, however, came close to the $2.6 billion caused by *Code Red* [5, 6]. Code Red exploited a vulnerability in Microsoft's Internet Information Services (IIS) [7] web server to infect its victims. The first, rather unsuccessful version of Code Red, attempted to spread itself by generating a set of random IP addresses that it then tried to infect. Yet, there was a fatal flaw in this version: it used a static seed to generate the IP addresses, which meant that all infected hosts generated the same set of IP addresses. This flaw prevented the worm from spreading far.

Several days after Code Red's arrival, a change in its behaviour was observed: it began to probe new hosts. The change in behaviour was due to an updated version of Code Red, identical in all aspects except for the random

number generator, which now used a dynamic random seed. This *mutation* enabled it to infect 359,000 hosts in less than 14 hours [6].

A worm that spread even faster was the Slammer [8] worm, which infected most of its 75,000 victims within 10 minutes. This worm was the first Warhol [9] worm observed in the wild, a name coined from Andy Warhol's famous quote that "in the future, everybody will have 15 minutes of fame" [10], and based on the worm's ability to spread to most vulnerable machines within 15 minutes.

Despite their prominence, Code Red and Slammer are just two of the more infamous worms drawn from the large pool of lethal worms that have swamped the Internet over the last decade. Study of those worms leads to the following observations about worm behaviour:

- the initial release of each worm is typically followed by one or more mutations
- each mutation tends to be more lethal than its predecessors, by refining the attack or infection strategy
- they spread significantly faster than humans can respond

These observations suggest that an attractive defence strategy against worms is to automate the detection of their mutations, and this strategy is the focus of this dissertation.

Today's *network intrusion detection systems* [11] – software designed to detect security breaches such as worms – typically use either anomaly detection or misuse detection. *Anomaly detection* [12] systems model *normal* traffic and detect intrusions by looking for abnormalities. But normal traffic is often hard to model, especially with traffic such as peer-to-peer and email relaying exhibiting worm-like characteristics. *Misuse detection* [13] systems match network traffic to models of intrusions known as signatures, which means that only intrusions whose signatures are known can be detected.

Although there is substantial ongoing research into improving anomaly detection systems, misuse detection systems have emerged as the *de facto* standard since they are both simple and scalable. When it comes to rapidly spreading intrusions such as worms, however, misuse detection systems have

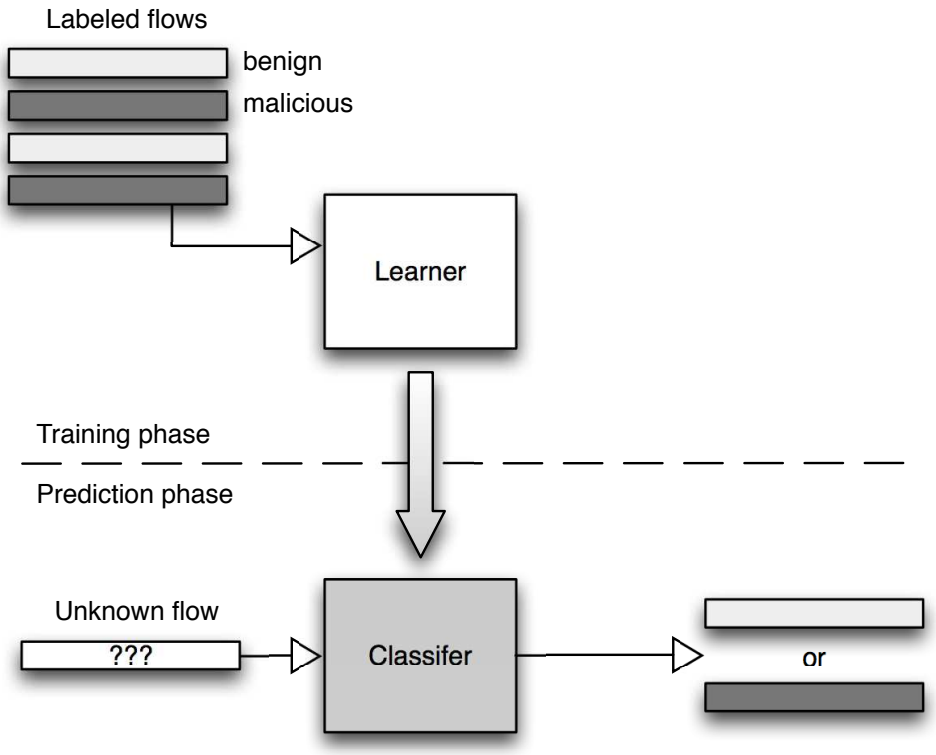


Figure 1.1: Automatically detecting worm mutations with machine learning. The learning phase is trained with network flows labelled as malicious or benign (top), producing a classifier that can distinguish between malicious and benign flows (bottom).

a serious limitation: generating the signature itself [14]. Signatures are typically created by security experts who manually analyse network logs *after* intrusions have occurred, a time consuming and error-prone task. Given the ferociousness of today's worms, by the time a signature has been handcrafted the worm may have already spread to the far ends of the Internet.

This work proposes to use machine learning based pattern recognition techniques to eliminate that bottleneck with regard to worm mutations. The idea is to train a machine learning classifier to distinguish between worm (malicious) and benign flows, and so avoid the need to explicitly generate signatures; by finding distinguishing patterns, machine learning techniques implicitly generate signatures to classify unknown flows as malicious or benign (Figure 1.1). In particular, this work investigates:

- **Support Vector Machines (SVMs)** [15] are a machine learning technique known to perform particularly well at pattern recognition tasks such as text categorisation and hand-written digit recognition. Since detecting worms is effectively a pattern recognition problem, this work applies SVMs to detect mutations of known worms. Specifically, this dissertation investigates the optimal configuration of SVMs and associated kernel functions to classify worms.

- **Comparison of SVMs to other machine learning methods**. A shortcoming of Support Vector Machines is that they do not return a confidence value with their predictions, often leading to unnecessary false alarms. Since Gaussian Processes (GPs) [16] automatically return confidence values with their prediction, this work compares SVMs to GPs in this particular domain. For further comparison, this dissertation also compares SVMs to K-nearest neighbours [17], known for its simplicity and solid results in other domains.

- **Generating worm mutations for training data**. Classifier accuracy depends on good quality training data – the more diverse the training data, the higher the classifier's accuracy. This work describes the design and implementation of a worm mutation generator whose generated mutations can be fed to the machine learning techniques as training data. This work evaluates whether the quality of the generated training data is sufficient for machine learning classifiers.

1.2 Thesis statement

Network intrusion detection systems typically detect worms by examining packet or flow logs for known signatures. This means that worms cannot be detected until after the signatures are created; in particular it means that mutations of known worms will remain undetected since they usually have a different signature.

I assert that it is possible to build an accurate network intrusion detection

system that automatically detects mutations of known worms using machine learning techniques. I will prove this assertion by:

1. Characterising the efficacy of different machine learning techniques for detecting synthetic mutations

2. Applying the most accurate machine learning technique to a wide variety of worm mutations

3. Devising an innovative technique for generating sufficient mutations of known worms to train the chosen technique

1.3 Contributions

This work contributes to the defence against computer worms and their often devastating effects in a number of ways:

- Demonstrating the feasibility of machine learning based pattern recognition techniques to detect mutations of known worms

- Comparing how effectively the Support Vector Machines, Gaussian Processes, and K-nearest neighbours machine learning techniques detect worm mutations

- Designing a framework that automatically generates worm mutations to be used as training data for machine learning classifiers

1.4 Publications

The work reported on in this dissertation has led to the following publication:

> O. Sharma, M. Girolami, and J. Sventek, *Detecting worm variants using machine learning*, in CoNEXT 07: Proceedings of the 2007 ACM CoNEXT conference, (New York, NY, USA), pp. 112, ACM, 2007 [18].

1.5 Outline

The remainder of this work is organised as follows:

Chapter 2 provides background information on worms and network intrusion detection, and describes related work by examining existing worms as well as intrusion detection systems optimised for detecting worms.

Chapter 3 presents an an overview of machine learning, followed by a detailed overview of the three prominent machine learning techniques that this work investigates: Support Vector Machines, Gaussian Processes, and K-nearest neighbours.

Chapter 4 introduces a system that applies machine learning to detect worm mutations. The chapter describes how network data is transformed into a format that is understood by machine learning techniques, and then proposes how to deploy machine learning techniques to detect mutations of known worms.

Chapter 5 investigates whether machine learning techniques in general, and Support Vector Machines in particular, are suited to detect worm mutations. It describes and analyses a set of experiments used to test this suitability.

Chapter 6 compares the Support Vector Machines' results from the previous chapter to two alternative techniques – Gaussian Processes and K-nearest neighbours – and discusses their relative advantages and disadvantages in detecting worm mutations.

Chapter 7 introduces a *worm mutation generator* to overcome lack of training data in the real world that threatens to starve machine learning methods of training data. Specifically it examines whether the generated mutations offer sufficiently high quality training data for the machine learning classifiers.

Chapter 8 summarises the suitability of machine learning for automatically detecting worm mutations, in particular, the relative strengths and

weaknesses of Support Vector Machines, Gaussian Processes, and K-nearest neighbours. It also revisits the effectiveness of the training data produced by the worm mutation generator.

Chapter 9 explores opportunities for future work. Possible opportunities include defending against sophisticated worm attack strategies, designing a custom kernel, alternative feature representation, real-time detection, online training, cascading classifiers, and scaling the infrastructure

Chapter 2

Worms and Intrusion Detection

The previous chapter highlighted the threat posed by worms and their mutations. This chapter now turns to the existing defence mechanisms against this worm menace. To better understand how worms can be detected, the first part of this chapter devotes itself to background information on worms – what kind of damage they cause, how they infect their hosts and how they spread. The second part scrutinises existing network intrusion detection systems, focusing on recent systems designed to detect worms.

2.1 Worms

Worms[1] [1, 2] are malicious programs that spread themselves to hosts on the Internet by exploiting vulnerabilities [20, 21] such as buffer overflows [22], in software applications. They are essentially computer viruses [23] that spread without the need of human intervention. Although initial worm experiments had already taken place in the early 1980's [1], the first Internet worm wasn't released until 1988. Named after its author, the Morris worm brought down hundreds of machines, at that time a significant portion of the Internet. As the Internet evolved into a global network, worms emerged as a global security threat. Several worms were released [4] in the decades that followed,

[1]The name *worm* originates from the *tapeworm* programs in the science fiction novel *The Shockwave Rider* [19] published in 1975.

wreaking havoc and causing considerable financial damage.

Even when worms are not designed to damage hosts but simply spread as fast as possible, they still severely disrupt networks by generating vast amounts of traffic [24] in similar ways that spam email is clogging the Internet. However, worms usually also carry additional payloads to damage individual hosts, rather than just negatively affecting their network connectivity. This payload allows the worm to execute arbitrary and potentially malicious code, with past damages ranging from defacing websites to corrupting files all the way to erasing entire hard disks. The potential for damage is not limited to software: even physical damage could be caused, for example by forcing insufficiently cooled machines to overheat with incessant heavy workloads.

2.1.1 Challenges in defending against worms

There are a number of challenges in successfully defending against worm attacks, including:

- **Excessive number of vulnerabilities**. Today's Internet has millions of hosts, directly accessible through public Internet Protocol (IP) addresses. Millions more are indirectly accessible behind gateways, routers and firewalls. Combined with the thousands of software applications (and often hundreds of different versions of particular software applications), worms are offered a vast number of potential backdoors through which to spread.

- **Immediate feedback**. With today's high speed Internet connections and trunks, worm authors have close to immediate feedback on whether their worm is successful. This enables worm authors to churn out new, more lethal versions much more quickly than manually maintained network intrusion detection systems can react.

- **Ease of access**. Coding worms was never easier. Websites provide all the necessary ingredients to produce a successful worm, from tutorials and source code, to entire development frameworks [25].

2.1.2 Worm attack strategies

To defend against a worm intrusion one must understand its strategy and line of attack. This section details the two stages in a worm's attack: infection and proliferation.

2.1.2.1 Infection

How do worms infiltrate their victims in the first place? Worms are just computer programs, so for them to self-replicate and propagate, they must be executed with sufficient privileges. If they cannot be executed or do not have sufficient privileges once executed, they are rendered harmless.

But how can one bring a remote host to execute the worm programs? Worms exploit low-level software vulnerabilities in services running on the remote host. These vulnerabilities abound in today's Internet, not least thanks to the large number of hosts that run outdated software whose vulnerabilities have been known for years. Common vulnerabilities include buffer overflows [22], integer overflows [26], incorrect handling of format strings [27], and faulty memory management [28, 29].

Vulnerabilities typically share the common trait of causing a computer's program counter to jump to the beginning of the worm's executable payload, thereby executing the worm program. From then on life is easy for the worm since it then runs at the same privilege as the vulnerable service in question, which more often than not means it has administrator (unrestricted) rights. If not, it is possible to combine the infection vulnerability with a second vulnerability that elevates the privileges of the worm.

2.1.2.2 Proliferation

Once the worm has successfully infected a vulnerable host, it typically self-replicates and spreads itself over the network. There are several spreading strategies it can use, ranging from simply generating a set of random IP addresses, to more sophisticated spreading strategies such as systematically, in a co-ordinated fashion, scanning ranges of IP addresses. The in-depth

discussion of worm spreading strategies in [10] reveals just how rapidly worms can spread when working in co-ordinated fashion.

As with some biological viruses, the number of hosts infected grows exponentially initially, until the majority of vulnerabilities have been infected. In later stages, the growth slows, typically due to network congestion and human intervention[2].

The key defence strategy is to stop worms in their initial stages. Worms such as Slammer [8] have demonstrated that the time window to line up a defence is less than 15 minutes, too short for human intervention. The only way, then, is to automate the detection of the worm and thus halt its spread in that critical first quarter of an hour.

2.1.3 A brief case study of worms

This section deepens the conceptual understanding of the previous section with concrete case studies of some of the most infamous worms. The case studies focus on the worm's infect-spread attack cycles.

2.1.3.1 Code Red

Code Red [6, 5] exploited a buffer overflow vulnerability in Microsoft's Internet Information Services (IIS) [7] web server to infect hosts. Once infected, Code Red generated a set of random IP addresses to which it then tried to spread. There was a fatal flaw in this worm: it used a static seed to generate the IP addresses, which meant that all infected hosts generated the same set of IP addresses. This flaw prevented the worm from spreading far.

Several days after Code Red's initial appearance, a change in its behaviour was observed: it began to probe new hosts. The change in behaviour was due to an updated version of Code Red, identical in all aspects except for the random number generator, which now used a dynamic random seed. This mutation enabled it to infect $359,000$ hosts in less than 14 hours [6].

[2]Interestingly, computer worms spread similarly to human disease; for an approach that uses a cyberspace equivalent of a Centre for Disease Control to help defend against Internet worms, see [30]

2.1.3.2 Slapper

Slapper [31] is a Linux worm that exploits a vulnerability in the OpenSSL [32] module used by older versions of the Apache web server [33]. It was let loose into the wild less than two months after the vulnerability in OpenSSL [32] was disclosed in July 2002.

Although the Slapper worm caused nowhere near as much damage as Code Red had in the previous year, it raised some interesting points. First, it showed that Linux is also susceptible to worm infections, dispelling the common myth of worms being a Windows problem. And second, it was the first worm to build up a peer-to-peer overlay of infected hosts (known as a botnet [34]) that enabled them to be remotely controlled for, say, distributed denial of service attacks [35].

How does Slapper infect its hosts? Slapper is launched with a single parameter – the target's IP address. First, it fingerprints this host to check if it is running a vulnerable version of Apache. It does this by sending an invalid `HTTP GET` request to which Apache responds with its version number. Slapper looks up this version in a hard-coded list, and if present proceeds with the attack.

Next, Slapper initiates an SSL handshake [36] with the server by sending a `client hello` message to OpenSSL. The server responds by sending the client its certificate. Usually a client would now respond by sending the server its public key together with the key's length. But this is where the vulnerability lies: OpenSSL does not check that the key's length is within certain bounds, facilitating a buffer overflow attack.

Slapper fakes its certificate and overstates the key's length so that when the server sends back the key it also sends additional data to Slapper. Since, as the name suggests, OpenSSL is open source, Slapper knows exactly which variables and data structures are returned as a result of requesting a too large key. Among other information, OpenSSL returns a reference to a data structure stored on the heap, which if overwritten allows arbitrary shell code to be executed.

To overwrite the heap value Slapper overflows the buffer a second time.

The second overflow only works because Apache's connections are handled by a process pool rather than a thread pool. Under normal circumstances new connection requests are served from the pool, unless all processes are busy, in which case a new child process is spawned. These child processes are identical to their parents, including the heap allocations. To force a new process to be spawned, Slapper exhausts the process pool by rapidly initiating 20 connections. The next request for a new connection is then guaranteed to be a new child process.

All in all, this makes a total of 23 flows[3] that Slapper needs to infect a host. The actual code run in the exploit loads the worm's source code from the client to the server, compiles, and executes it.

Slapper serves as an excellent case study because it is one of the few worms where the source code is easily available, and as such can be tailored to the purposes of this work. Old versions of Linux and Apache are also easier to obtain and install than legacy editions of Windows and the IIS web server.

2.1.3.3 Slammer

A worm that spread even faster was the Slammer [8] worm, which infected most of its 75,000 victims within 10 minutes. Slammer, which was released in January 2003, exploited a buffer-overflow vulnerability in Microsoft's SQL Server, which was reported 6 months earlier in July 2002. This worm was the first Warhol [9] worm observed in the wild, a name coined from Andy Warhol's famous quote that "in the future, everybody will have 15 minutes of fame" [10], and based on the worm's ability to spread to most vulnerable machines within 15 minutes.

Slammer's strength lies in it's simplicity: it does nothing more than infiltrate the host using the above mentioned vulnerability, and then continuously generates random IP address and spreads itself to those hosts if it finds them to be vulnerable. Although Slammer does not perform anything malicious on the host, the sheer volumes of traffic it generated caused large network

[3]One for the initial probe, 20 to exhaust the pool, and two for the buffer overflows.

outages [8]. Slammer's growth initially followed an exponential curve and was later ironically slowed by the collapse of many networks due to denial of service caused by Slammer itself.

The entire Slammer worm occupies only 376 bytes and fits in a single UDP packet. Due to this small size and because it used UDP rather than TCP, Slammer was often able to slip past heavily congested networks where legitimate traffic could not. Using UDP also meant that the worms spread was limited by available bandwidth rather than network latency, as is the case with TCP-based worms such was Code Red. Another benefit of using UDP over TCP is that UDP does not require a connection to be set up, which meant that packets could simply be fired and forgotten, rather than having to keep track of multiple connections, for example with threads which are limited in number by the operating system, and drastically increase a worm's complexity.

Although Slammer was the first successful Warhol worm, it was not perfect. Moore et al [8] suggest that minor changes, such as a better random IP address generator, could have further improved its spread. Additionally, Slammer was eventually slowed down by blocking UDP port 1434, the MS SQL server port; had it used a more popular service port, such as that of HTTP or DNS, this strategy could have rendered the World Wide Web unusable.

2.1.3.4 Witty

A worm that targeted a buffer overflow vulnerability in several Internet Security Systems (ISS) products was the Witty worm [37], which began to spread in March 2004 and infected around $12,000$ hosts. The worm gets it's name from the payload it carries, which contains the phrase: *insert witty message here*.

The witty worm is interesting due to several distinguishing features. First, it was the first widely propagating worm that carried a destructive payload. Second, it had the shortest known interval between disclosure of the vulnerability and release of the worm, spreading only a day after the vulnerability was

disclosed. And third, it was launched in an organised, highly co-ordinated manner with numerous ground-zero hosts.

Witty's attack cycle is fairly straightforward:

1. Seed the random number generator using system time.
2. Send 20,000 copies of itself to random targets.
3. Select a hard disk at random.
4. If successful overwrite a randomly chosen block on this disk, and start over with step 1.
5. If unsuccessful, start over with step 2.

This process repeats until either the infected machine is rebooted or crashes, for example when Witty manages to overwrite a system critical section of the hard disk.

The Witty worm was observed to have infected 110 hosts within the first ten seconds and 160 at the end of 30 seconds, indicating that a large number of initial hosts were seeded with the worm before launching the first attack wave. Given that the worm began to spread only a day after the vulnerability was disclosed, it is likely that these initial hosts had been previously compromised.

Like the Slammer worm, the Witty worm used the UDP protocol and was therefore bandwidth limited, rather than latency limited as is the case with TCP worms. Additionally, Witty worm cloaked itself by padding its packets with arbitrary data varying in size between 796 and 1307 bytes.

2.1.3.5 Blaster

The Blaster[4] [38] worm was released into the wild in August 2003, and exploited a buffer overflow vulnerability in the Distributed Component Object Model (DCOM) Remote Procedure Call (RPC) [39], a service for communicating with objects distributed across networked hosts, in Microsoft Windows

[4]The Blaster worm is also referred to as the Lovesan worm because it carries the hidden message "I just want to say LOV YOU SAN!!"

XP and Windows 2000. As a side effect, the worm also caused instability in the RPC service on other Windows versions, including Windows NT, Windows XP 64, and Windows Server 2003.

The worm attacks as follows:

1. It connects to TCP port 135 and overflows the DOM RPC service with excessive amounts of data.

2. The buffer overflow overwrites a critical memory section and facilitates shell access on TCP port 4444 with local system access.

3. Via the newly granted shell access, Blaster invokes `tftp.exe`, an FTP client to transfer its payload (`mblast.exe`). Transferring the payload from the attacking host rather than hard-coded servers makes it more difficult to detect the worm.

4. Blaster sets an entry in the Windows registry to launch the executable on the next boot. This keeps Blaster alive on the target host even when the machine is rebooted.

5. The worm reboots the system to launch the executable payload.

6. The infected host then listens on UDP port 69 for connections from newly compromised hosts.

Once it has infected its host, Blaster launches attacks on a regular basis. If the date is between August 15th and December 31st it continuously launches distributed denial of service attacks against *windowsupdate.com*. Blaster also repeats these attacks on the 15th of every month outside the date range. The attack floods destination port 80 with 50 TCP SYN packets (40 bytes each) per second.

To spread itself, Blaster first generates a random IP address of the form `A.B.C.0`, where `A`, `B`, and `C` are random values between 0 and 254. It then incrementally scans the entire subnet `A.B.C.0` – `A.B.C.254` for new victims.

2.2 Intrusion detection systems

Network intrusion detection systems [11] are specialised intrusion detection systems [40, 41] that monitor network traffic for security breaches such as worms. These systems are typically deployed at network gateways, allowing them to act as filters for all incoming traffic.

Since the machine learning worm detectors proposed in this work are intended to augment or replace existing intrusion detection systems, it makes sense to look at the common approaches to intrusion detection and better understand how they work. This will be the task of the first part of this section. The second part then presents an overview of popular network intrusion detection systems, focusing on those features that deal with worms.

2.2.1 Approaches to intrusion detection

There are two common approaches to network intrusion detection: anomaly detection and misuse detection. In *anomaly detection* [12], systems are equipped with a model of *normal* traffic and detect intrusions by comparing traffic to this model and looking for abnormalities. A weakness is that the diversity of network traffic makes it difficult to demarcate *normal* traffic. Email relaying and peer-to-peer queries, for example, exhibit worm-like traffic characteristics. Matters are complicated further because abnormal traffic does not necessarily constitute an intrusion.

In *misuse detection* [13], on the other hand, systems are equipped with models of intrusions, known as signatures, which are matched to network traffic. A *signature* is a fingerprint that can be used to identify intrusions. In its simplest form, it consists of a string of characters (or bytes), but many current intrusion detection systems [42, 43] also support regular expressions [44] and even behavioural fingerprints [45]. A problem with misuse detection is that only intrusions whose signatures are known can be detected.

Although there is substantial ongoing research into improving anomaly detection systems, misuse detection systems have emerged as the *de facto* standard for intrusion detection since they are both simple and scalable.

When it comes to rapidly spreading intrusions such as worms, however, misuse detection systems have a serious limitation: generating the signature itself [14].

Signatures are typically created by security experts who analyse network and host logs *after* intrusions have occurred. This involves sifting through thousands of lines of log files – an error-prone and time-consuming undertaking. Given the ferociousness of today's worms, by the time a signature has been handcrafted the worm may have already spread to the far ends of the Internet.

2.2.2 Case studies of intrusion detection systems

How are the approaches to network intrusion detection from the previous section applied in practice? This section presents popular network intrusion detection systems, focusing on their worm detection features.

2.2.2.1 Snort

Snort [43] is a popular open source network intrusion detection system that belongs to the family of misuse detection systems. Snort works by string matching observed network traffic to a database of known intrusions. To be effective, this database must be updated frequently, which is why there are several mechanisms in place to ensure that new signatures are shared throughout the Snort community quickly and easily.

Snort first divides the traffic based on destination port number. Then a rule (or signature) can restrict itself to particular known header values, or dig deeper and perform content matching on the payload. Since string matching is a relatively expensive operation, it is important to filter out as much as possible before resorting to content inspection.

Initially Snort focused only on the inspection of single packets. After several successful evasions [46], however, Snort was extended to support full TCP flow reassembly. This enables Snort to perform pattern matching across packet boundaries, greatly increasing its success rate.

2.2.2.2 Bro

Bro [42] is another open source network intrusion detection system, although its approach is more refined than Snort's since it works at the application protocol level. However, Bro can handle Snort signatures by means of a conversion script that is shipped with Bro.

Bro works by first assigning a protocol analyser based on preliminary information, primarily the protocol number, that it extracts from the first few bytes of reassembled flows. The protocol analysers themselves then generate events based on various protocol specific data exchanges. For example, a hypertext transfer protocol (HTTP) analyser generates an event for every `HTTP GET` request. These events in turn are parsed by policy scripts that accompany the protocol analysers. These policy scripts are written in a dedicated scripting language designed for easy manipulation of the received events and data.

Bro's stream assembly and protocol analysis is a heavy-weight process and as such should only be done on flows where it is absolutely necessary. To reduce the number of flows it analyses, Bro filters unwanted flows using primitive packet filtering.

2.2.2.3 Earlybird

Earlybird [14] was one of the first research systems dedicated to fingerprinting new worms, thereby automating the signature generation process. It does this by building a histogram of all byte strings in all packets that it observes and constructs signatures for the most frequent ones. Earlybird's fundamental assumption is that since worms try to spread to many hosts in a short period of time, strings that occur frequently at widely dispersed network locations must belong to a worm.

Earlybird reduces false alarm rates by comparing the current string with lists of benign strings that are known to occur both frequently and appear at widely dispersed network locations. These lists of benign strings are known as *white lists*.

2.2.2.4 Autograph

Autograph [47] takes the same approach as Earlybird by assuming that worms generate bursts of similar traffic to widely dispersed network locations. It improves over Earlybird, by (i) using a heuristic filter to narrow down the traffic it has to inspect, and (ii) observing entire flows rather than just single packets.

The heuristic filter simply classifies all flows originating from port-scanning sources as suspicious, based on the observation that many worms scan IP address ranges in search of vulnerable hosts. Autograph generates signatures from the suspicious flow pool by dividing flows into content blocks, and then applying a greedy algorithm to pick out the most prevalent blocks.

Like Earlybird, Autograph attempts to reduce false alarms with white lists. The authors suggest a training period to collect information to produce these white lists.

2.2.2.5 Polygraph

Polygraph [48] is a misuse detection system that generates signatures for polymorphic worms by extracting string similarities in pools of suspicious and innocuous flows. Polymorphic worms are worms that mutate themselves at every hop when spreading throughout the network; the possible resilience of the work proposed in this dissertation to polymorphic worms is discussed in Section 5.1.1.3.

Polymorphic worms use a number of mutation strategies, such as corrupting its own signature, and Polygraph suggests different string matching algorithms, each optimised for a specific strategy. If the mutation strategy is unknown, they suggest experimenting with each algorithm and selecting the one that yields the least false alarms.

Polygraph extends Autograph's framework to fingerprinting polymorphic worms. The authors argue that continuous byte strings, such as used in signatures for Autograph are not sufficient to describe polymorphic worms. The underlying assumption is that a polymorphic worm, although it changes its signature from hop to hop, has at least a small fraction of signature in com-

mon at each incarnation. In most cases that common fraction is the exploit code, which Polygraph tries to deduce with string matching algorithms such as finding the longest common sub-sequence [49].

2.2.2.6 PayL

PayL [50, 51] is an anomaly-based detection system that builds traffic models based on byte frequency distributions in packets.

In the modelling phase, PayL categorises packets according their destination port, length, and byte frequency distribution of their payload. It generates these byte frequency distributions by iterating over the payload using a sliding window of size one-byte, and storing a histogram of observed bytes. The average frequency and standard deviation of each byte over all histograms for packets with the same destination port and length are then stored as the traffic models.

Likewise, in the classification phase packets are first categorised according to their destination port, then length, and the distance of the payload's byte frequency distribution to the model's byte frequency distribution. Anomalous packets are those were this distance is above a certain predefined threshold.

An extension using n subsequent bytes rather than just single byte distributions was proposed in [51]; this technique, known as *n-gram extraction*, is used in this dissertation and covered in detail in Section 4.4.

2.2.2.7 Ensemble of one-class classifiers

Similar to this dissertation, Perdisci et al [52] use Support Vector Machines to detect intrusions. The fundamental difference is that Perdisci et al [52] take the anomaly detection approach, while this dissertation takes the misuse detection approach. The model of normal traffic is based on PayL's n-gram extraction, except that higher n-grams are extracted with 2_v-gram approximation[5].

[5]This technique uses a sliding window size of 2 and approximates greater values of n by incrementing the window by v steps, when traversing the data.

Another difference is that Perdisci et al [52] propose an ensemble of one-class SVMs, while this dissertation proposes binary (two-class) Support Vector Machines. Furthermore, the one-class SVMs in Perdisci et [52] use only a single kernel, whereas this work investigates a number of kernels. Kernels and binary classifiers will be covered in the background section on Support Vector Machines (Section 3.2).

The final difference is that Perdisci et al [52] investigate a wide range of n-gram values with 2_v-gram approximation. They show that 2_v-grams yield good results, however propose the use of multiple classifiers, each operating in a different feature space, for optimal results. By contrast this dissertation investigates only n-gram values in the range 1-3, and will demonstrate in Section 5.3.3.1 that 2-grams yield solid results. Feature extraction using n-grams is described in detail Section 4.4.

2.2.2.8 Vigilante

Vigilante [53] is a misuse detection system that, unlike other intrusion detection systems discussed so far, monitors code executed on individual hosts rather than network traffic flows. Vigilante determines whether a network connection is malicious by running all instructions in a shielded virtual machine, where it monitors for illegal memory accesses and buffer overflows.

An advantage of Vigilante is that it can detect rapidly spreading worms where exploits are unknown without blocking innocuous traffic. Another advantage is that Vigilante can rapidly contain a worm in its early stages because it disseminates any intrusion it detects to nearby Vigilante hosts, effectively building a distributed worm detection system.

On the downside, being a host-based detection system by nature means that Vigilante (i) has to be installed on each host in the network, (ii) monitoring machine instructions in a virtual environment is computation-intensive, and (iii) programming bugs such as pointer errors could be wrongly flagged as intrusions, resulting in a high false alarm rate.

2.2.2.9 Honeycomb

Honeycomb [54] is a misuse detection system that generates signatures by observing traffic in honeypots. *Honeypots* [55] are virtual systems that have been deployed on hosts that have not been assigned a domain name and are not publicised in any way. They lure worms, based on the assumption that any attempted communication with these machines must be malicious since the only way of obtaining their IP address is through random probing. Honeycomb generates a signature for incoming traffic by finding the longest common substring in two network connections.

False alarms are easily generated if legitimate traffic reaches the honeypots, for example by attackers who deliberately send benign traffic to these honeypots.

2.3 Summary

This chapter presented background information and related work on worms and intrusion detection systems.

Worms are malicious programs that automatically spread themselves to hosts on the Internet by exploiting vulnerabilities of known services, often wreaking havoc and caused considerable financial damage. The exemplar worms covered in this chapter were:

- **Code Red**, which exploited a vulnerability in Microsoft's Internet Information Services web server and infected 359,000 hosts in less than 14 hours.
- **Slapper**, a Linux worm that exploited a vulnerability in the OpenSSL module of the Apache web server, and built a peer-to-peer overlay of infected hosts, allowing them to be remote controlled.
- **Slammer**, which was the first Warhol worm released into the wild that spread to 75,000 victims within 10 minutes using a vulnerability in Microsoft's SQL server.

- **Witty** exploited a buffer overflow vulnerability in several Internet Security Systems just a day after the vulnerability was disclosed, and cloaking its rapid spread by padding itself with arbitrary data.

- **Blaster** targeted Windows' DCOM RPC with a buffer overflow, downloading its payload with an FTP call-back, and registering itself with the Windows Registry to survive reboots.

Intrusion detection systems are special purpose monitoring systems that attempt to detect intrusions such as worms. Generally. there are two approaches to network intrusion detection: *anomaly detection*, which builds models of normal traffic and flags all traffic that does not conform to this model, and *misuse detection*, which scans for signatures of known intrusions. Exemplar intrusion detection systems covered in this chapter include:

- **Snort and Bro**, two widely used open source misuse detection systems. Snort matches network packets to rules with string matching, while Bro works with protocol specifics (such as `HTTP GET` requests). All rules must be handcrafted.

- **Earlybird, Autograph, and Polygraph**, three intrusion detection systems that automatically generate signatures by using string algorithms, such as finding the longest common subsequence, to differentiate between pools of benign and malicious flows.

- **PayL and One-Class**, two anomaly detectors that use byte frequency distributions to model the traffic.

- **Vigilante**, a host-based intrusion detection system that runs instrumented software in a shielded environment that monitors for malicious code such as illegal memory accesses and buffer overflows.

- **Honeycomb**, an intrusion detection system that generates signatures by observing traffic in virtual systems that have been deployed on decoy hosts that have not been assigned a domain name and are not publicised in any way.

The following chapter presents background information of Machine Learning in general, and three techniques (Support Vector Machines, Gaussian Processes, and K-nearest neighbours) in particular. Chapter 4 then combines the background of this and the next chapter to present a network intrusion detection system that uses machine learning to automatically detect worm mutations.

Chapter 3

Machine learning

This chapter presents background information on Machine Learning, specifically three prominent supervised learning methods: Support Vector Machines [15], Gaussian Processes [16], and K-nearest neighbors [17].

3.1 Introduction

Machine learning [56, 57, 58] is a sub-discipline of artificial intelligence that involves developing algorithms to automatically recognise patterns using statistical classification. Examples of patterns it can recognise include fingerprints, images, handwriting, voice recordings, and, as this dissertation will show, worms.

Formally, in statistical classification [57] a pattern is represented by d features occupying a point in a d-dimensional space (the *feature space*). The d-dimensional vector defining a particular pattern is called the *pattern vector*. Consider the pattern recognition problem of classifying a person as male or female given only statistical facts about that person. A possible feature set consists of the height, weight, and age. The feature space would be 3-dimensional, and a 36 year old person weighing 72kg and measuring 1.80m would occupy the pattern vector $< 36, 72, 180 >$ in that feature space.

The aim of machine learning is to select features so that pattern vectors belonging to different categories (such as male and female) can be partitioned

into disjoint regions within the feature space. Seen from a different angle, the effectiveness of a feature set is determined by how well it partitions pattern vectors into different regions.

3.1.1 Why machine learning?

What makes machine learning attractive for worm detection? The answer is that machine learning offers a number of key advantages over rival pattern recognition methods.

One of the simplest such rival methods is *template matching* (for example [59]), where the pattern to be recognised is matched against stored prototypes (templates), taking into account possible mild alterations such as rotation or scaling. Simple template matching is computationally demanding and fails to recognise distortions to the pattern such as a change of viewpoint. Most of all, template matching depends on the availability of templates to match against, whereas machine learning techniques are able to learn these patterns from training data.

Another rival pattern recognition method is the syntactic approach [57]. The underlying idea is that a pattern is composed of smaller sub-patterns that are themselves composed of sub-patterns, and so on, down to the simplest sub-patterns called *primitives*. The pattern can then be seen as a sentence in a language where the primitives are the alphabet. A language syntax – consisting of the alphabet and a set of grammatical rules – governs how complex patterns can be created. Although intuitive, the challenges posed by the syntactic approach is how to subdivide patterns in the first place, especially when noise is introduced (as is often the case in network flows carrying worms). Another drawback is the heavy computation required to perform the subdivisions.

3.1.2 Supervised vs. unsupervised learning

In machine learning the learning stage can be either supervised or unsupervised. In *supervised learning* the learning stage is told what class a training

data item belongs to, for example that pattern $< 36, 72, 180 >$ belongs to the male class. In *unsupervised learning* only the pattern vectors but not the class they belong to are fed to the learning stage, leaving it to the learning stage to cluster the pattern vectors.

Another way of looking at the difference between supervised and unsupervised learning is that in supervised learning the training data is already grouped into regions before it enters the learning stage, and the goal is to classify unknown pattern vectors based on their proximity to a particular group. In unsupervised learning, on the other hand, the learning stage must first discover groups of similar pattern vectors in the input data.

This dissertation will use supervised learning because of the similarity between worm traffic and normal traffic [60].

3.1.3 Parametric vs. non-parametric methods

Besides supervised and unsupervised learning, machine learning techniques can be either parametric or non-parametric. A *parametric model* is a data set that can be described by a finite number of parameters, for example a normally distributed data set can be defined by its mean and standard deviation. A *parametric method* estimates the parameters of a model; this learned model is then used for classification or regression.

Parametric methods are appealing: once the parameters are learned the computations are highly efficient because they operate on the model parameters and not on the data items in that model. But parametric methods have two severe drawbacks. First a bad model (caused by over-fitting or insufficient data) could lead to gross errors in classification or regression. And second, many real world problems (such as worm detection) cannot be modelled accurately with a set of parameters.

Because of the risks posed by parametric models, this dissertation devotes itself exclusively to non-parametric methods. Non-parametric methods have the additional advantage that they require no prior knowledge about the distribution and any inference can thus be made directly from the training data. In other words, non-parametric models offer greater flexibility since

they require only a similarity measure to be defined between objects.

3.2 Support Vector Machines

Support Vector Machines [61, 15, 62, 63] is a supervised learning method for automatic pattern recognition. It emerged in the mid 1990s from a combination of two independent research streams:

- Advances in Computation Learning Theory, a mathematical discipline that analyses machine learning algorithms, and
- Development of *kernel functions* [64, 65] that efficiently transform non-linearly separable data into linearly separable data

Combining the two streams resulted in a machine learning method that uses an optimisation algorithm rather than a time-consuming greedy search.

Since their inception, Support Vector Machines have been successfully applied to solve a large number of real-world pattern recognition problems including text categorisation [66], image classification [67], and hand-written character recognition [15].

Support Vector Machines can be used for classification and regression. *Regression* finds the curves of best fit for a given data set, while *classification* categorises a given data set into two or more classes. This work is interested in classification only, specifically in classifying network flows as malicious (carrying a worm) or benign. For this reason the remainder of this section will discuss Support Vector Machines in the context of classification only; for information on regression, see [68].

The rest of this section is organised as follows. First comes an overview of how Support Vector Machines work. This overview will provide an introduction to Support Vector Machines in a systems context, with emphasis on detecting worm mutations; interested readers can find the underlying mathematics in Appendix A[1]. This section then provides an example application

[1]For a formal treatment of Support Vector Machines, see also [63]

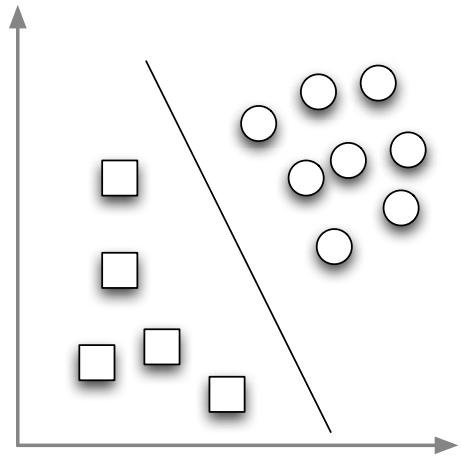


Figure 3.1: Classification involves dividing the training data by a separating hyperplane. In 2-dimensional space as in the above figure the hyperplane is a line.

of Support Vector Machines. The final parts of this section are devoted to kernels, which are needed to classify more complex data sets.

3.2.1 How Support Vector Machines work

Classification in general is achieved by dividing the training data into disjoint groups. If the data is linearly separable, then in 2-dimensional space these disjoint groups can be pictured as being separated by a dividing line, as shown in Figure 3.1. More generally for higher dimensions the data will be separated by a hyperplane, sometimes also called the *decision surface*. Equipped with the hyperplane, the classifier can then label a given test data point based on its position relative to the hyperplane.

For a given data set there may be more than one separating hyperplane, as depicted in Figure 3.2, and individual classification techniques can be distinguished by which hyperplane they choose. In particular, the crucial difference between Support Vector Machines and other classification techniques such as perceptrons [69] is that Support Vector Machines find the optimal separating hyperplane by maximising the margin between the hyperplane and a subset of training data points called the *support vectors*, pictured in

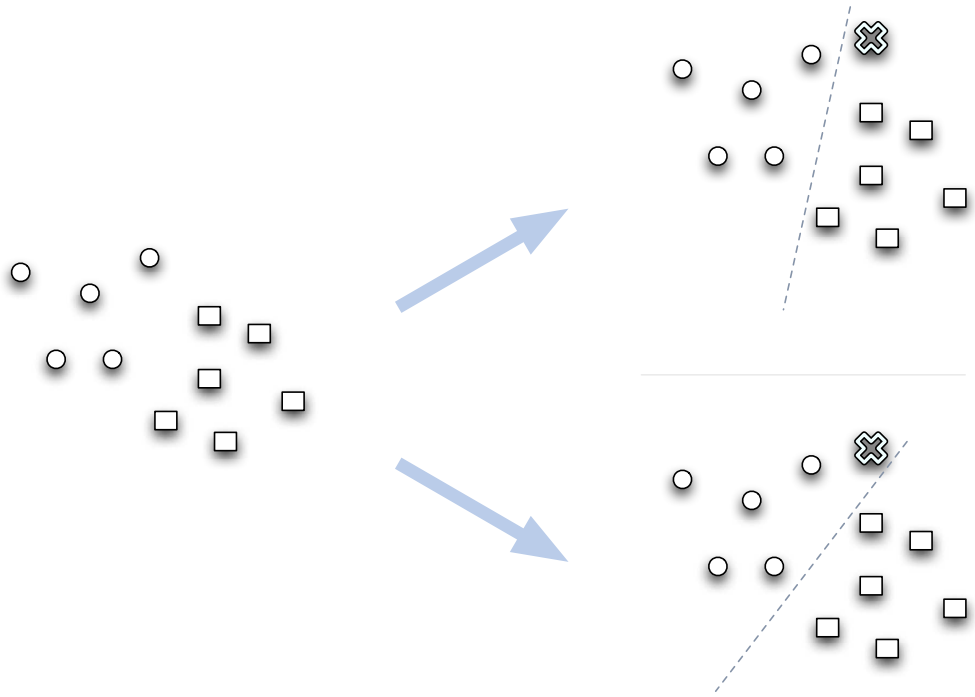


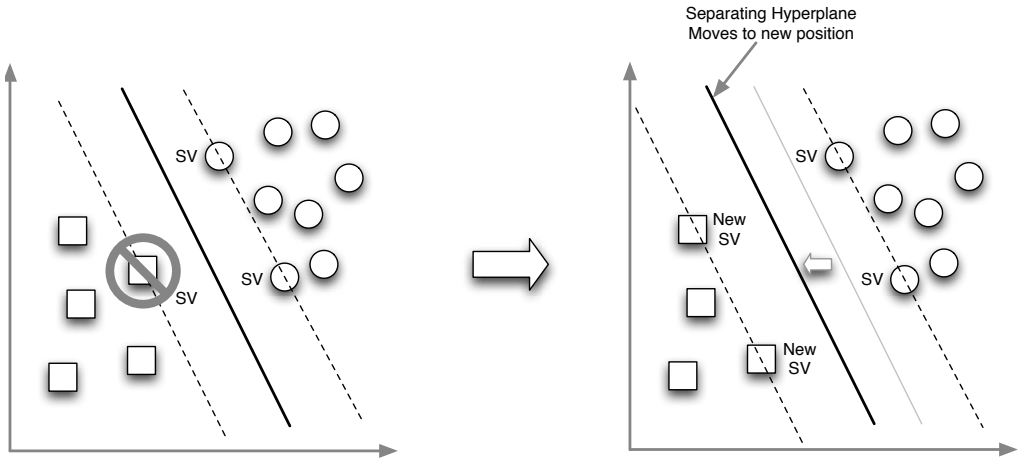
Figure 3.2: There may be more than one separating hyperplane for a given data set; `x` will be classified depending on which separating hyperplane is used.

Figure 3.3. Support vectors are the key data points close to the hyperplane that, if removed, would change the location of the hyperplane.

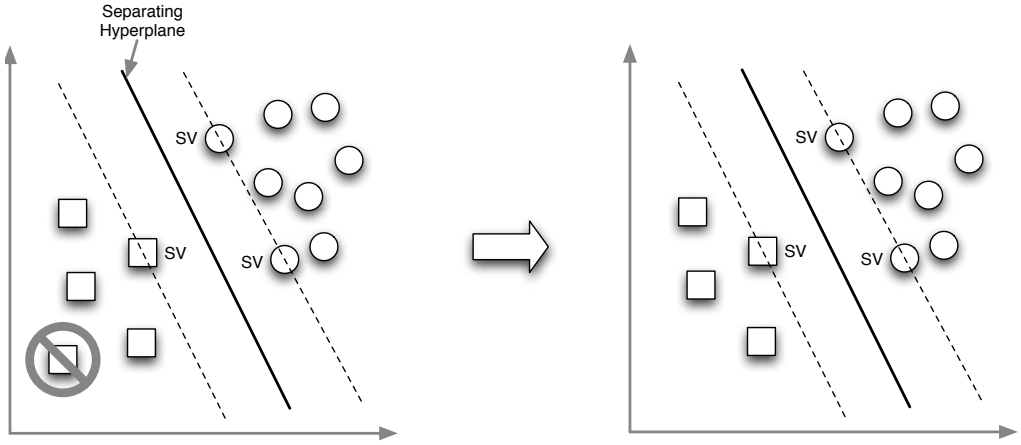
3.2.2 An example

As an example of how Support Vector Machines classify data, consider again determining a person's gender given only a set of statistical data about that person. To make matters a bit simpler than in the earlier example, assume that the features height and weight are enough to determine a person's gender.

Finding the hyperplane can be visualised as plotting the given heights and weights in a two dimensional co-ordinate system and drawing a line (the separating hyperplane) that divides the points into regions *male* and *female*. While there are many possible separating lines, the optimal hyperplane would be the one which maximises the distance between training points of the male



(a) Removing a support vector affects the position of the hyperplane.



(b) Removing a vector that is not a support vector, does not affect the postion of the hyperplane.

Figure 3.3: Support Vector Machines find the optimal separating hyperplane by maximising the margin between the hyperplane and a subset of the training data points called the *support vectors*. By definition, removing a support vector moves the location of the hyperplane.

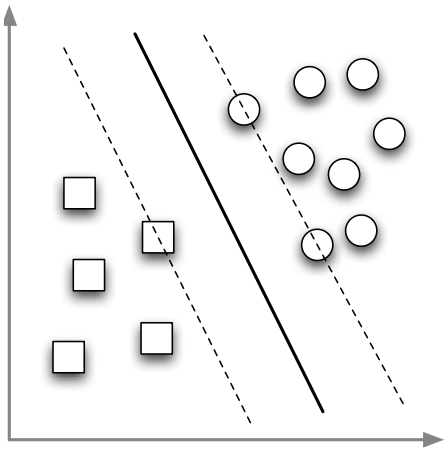


Figure 3.4: The optimal hyperplane maximises the distance from any training point.

and female classes, as shown in Figure 3.4.

3.2.3 Non-linearly separable data and the kernel trick

As mentioned earlier in this section, given a set of pattern vectors and their classification, Support Vector Machines determine the maximum margin separating hyperplane. Unfortunately, this is not always that straightforward – it is often the case that data is simply not linearly separable. This is where the so-called *kernel trick* [61] comes to the rescue.

A *kernel* is defined as a measure of similarity between two pattern vectors. Conceptually, kernels equip Support Vector Machines with the ability to map non-linearly separable data points into a different dimension where they are linearly separable. Consider as an example the data points in Figure 3.5: the data set on the left is not linearly separable, yet looking at the data, it is easy to picture an elliptical boundary that distinguishes between the two classes. The trick, then, is to map the data points into a dimension where they are linearly separable, as shown in the right diagram.

Mapping data points to a higher dimension is seemingly costly. More dimensions would mean larger vectors, which in turn would mean larger memory requirements and longer calculation times. Thanks to kernels, however,

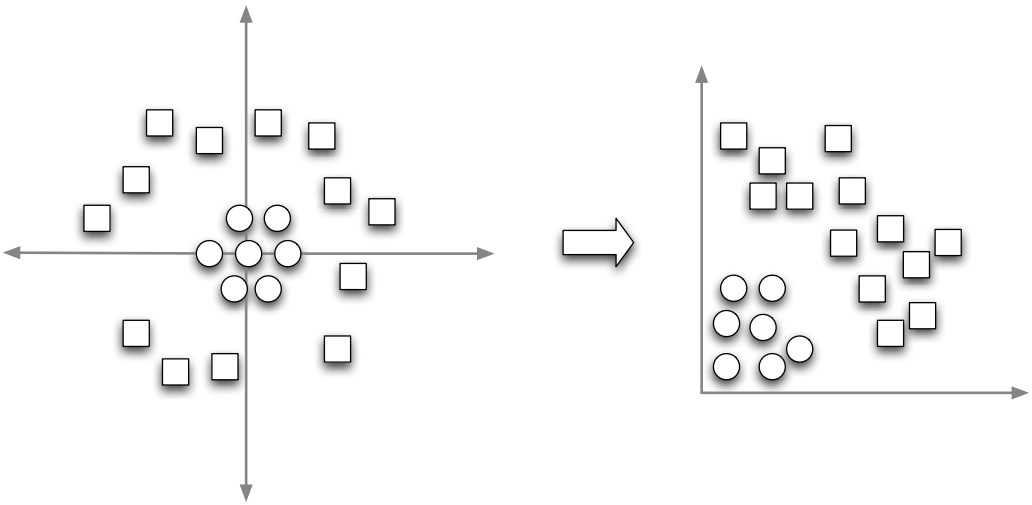


Figure 3.5: Support vector machines deal with linearly inseparable data by implicitly mapping it to a dimension where it is separable.

SVMs do not need to store these high dimensional vectors explicitly. Kernels are a means of implicitly defining a feature space via the inner product in that space. In other words, kernels encapsulate the mapping into higher dimensions and the similarity calculations in that dimension without having to store the mapped data points [63].

Since the kernel handles the critical job of finding a way to separate – and thus classify – the data, it is the key parameter in a Support Vector Machine. Unfortunately, there is no silver bullet choice of kernel: each kernel has its advantages and disadvantages for the data in question. For this reason, one of the aims of this work is to find the optimal kernel configuration for Support Vector Machines to detect worms. To this end, the following candidate kernels have been examined:

- **Linear kernel** [15]. This is the standard SVM kernel that tries to find a dividing hyperplane by calculating the dot product on pattern vectors in the original feature space (no mapping to a higher dimension is performed). Being the standard kernel and operating in the input feature space renders it a simple kernel as a basis for comparison.

- **Radial basis function kernel (RBF)** [15]. This kernel applies a

Gaussian function to the pattern vectors, implicitly taking them to a higher (infinite) space. Being a radial basis function it operates solely on the length of the patterns and not the direction or position. The RBF kernel has become popular thanks to its good performance in a wide range of applications [66, 70].

- **String kernel** [71]. Originally developed for categorising text documents [71] such as spam email, the string kernel maps the input strings into the feature space generated by all sub-strings of a given length, to which it then applies the inner product. Since a network flow is a stream of bit strings the string kernel offers itself as a strong candidate [72].

Of the above kernels, the RBF kernel is often suggested as a reasonable first choice [73] because of its solid general-purpose performance, and unlike the linear kernel it can deal with non-linearly separable data by implicitly mapping the input feature space into a higher dimension. This dissertation investigates in Chapter 5 whether this holds true for worm detection. Additionally, it has been shown that the linear kernel is a special case of the RBF kernel [74] – intuitively a very wide and short Gaussian distribution can be thought of as linear.

What about other kernels such as the polynomial and sigmoid kernels? A disadvantage of the polynomial kernel [15] is that it has more hyperparameters than the RBF kernel, making the search for an optimal combination more time consuming. Additionally, the RBF kernel has less numerical difficulties, while the sigmoid kernel has been shown invalid for certain parameters [75].

3.2.4 Support Vector Machines revisited

Figure 3.6 summarises the Support Vector Machines cycle discussed so far, consisting of a *learner* and *classifier* stage. The learner is supplied with labelled training data, from which it derives the Support Vectors with the help of the kernel. Depending on the kernel, this will be in the input feature space (as is the case for linear kernels) or in a richer feature space (as is the

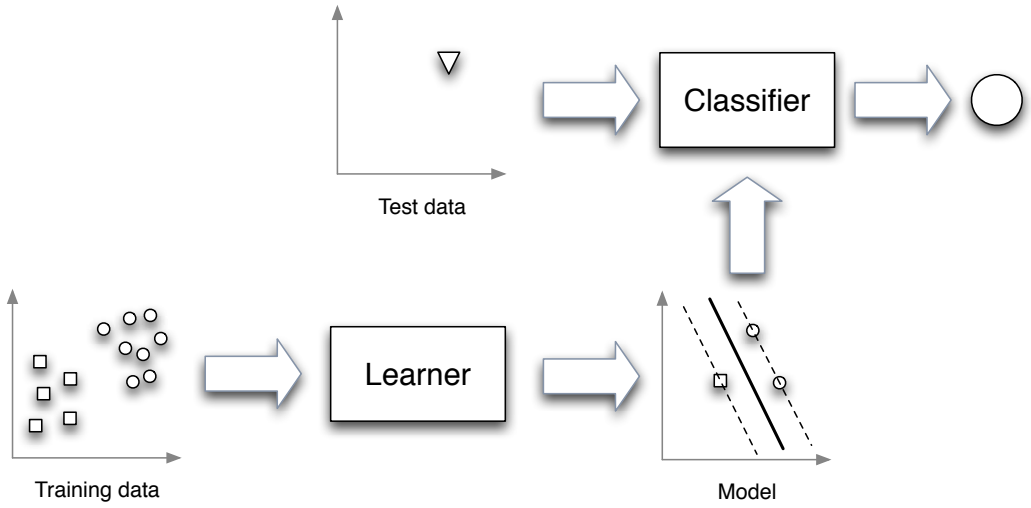


Figure 3.6: **Support Vector Machines cycle**. Training data is fed into the learner, which then builds a model that allows the classifier to classify unknown test data.

case with RBF or string kernels). Equipped with the support vectors, the classifier can then classify test data by calculating the closeness of the test data to each support vector, again using the kernel.

3.3 Gaussian Processes

An alternative machine learning technique that has proven its worth in classification is Gaussian Processes [16, 76, 77]. Like Support Vector Machines, Gaussian Processes are a supervised learning method in which the classifier is first trained with labelled training data from which it then infers its predictions.

Unlike Support Vector Machines, which divide the training data into disjoint groups with a hyperplane and then classify test data based on its position relative to the hyperplane, Gaussian Processes derive a function that fits the training data and accurately predicts the classification for an unknown test data point. More formally, given a training data set of N observations $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$ where each x_i is an input pattern vector and y_i its corresponding output, Gaussian Processes find a function $f(x)$ such that

$f(x_n) = y_n \ \forall (x_n, y_n) \in N$ and that predictions can be made for unknown values of x.

As with Support Vector Machines, there is an important distinction between regression and classification: if the function $f(x)$ is continuous then it can be used for data regression, while if $f(x)$ is discrete it can serve for data classification. Contrary to Support Vector Machines, however, where regression and classification can be treated in separate discussions, the two are inextricably linked in Gaussian Processes. In particular, classification in Gaussian Processes involves first finding a regression (continuous) function $f(x)$ before applying a *response function* [16] that returns a discrete value for classification.

The remainder of this section is structured as follows. The first part argues the case for Gaussian Processes as a potentially attractive alternative to Support Vector Machines, particularly for worm detection. This part also outlines how Gaussian Processes fit into the this dissertation. The next part explains how Gaussian Processes work; as with Support Vector Machines this explanation will be from a systems design perspective. The final parts of this section describe how squashing functions facilitate classification, and how kernels (like their SVM counterparts) allow Gaussian Processes to handle complex data sets.

3.3.1 Why Gaussian Processes?

What makes Gaussian Processes an attractive machine learning technique compared to Support Vector Machines? A shortcoming of Support Vector Machines is that they only return a classification, but no indication of how confident they are that this classification is correct. Gaussian Processes, on the other hand, automatically return confidence values as part of their result.

Confidence values add another angle to the classification that can help reduce false alarm rates. Equipped with the confidence value, the classifier can make a more informed decision, for example by accepting a proposed classification only if the confidence value is above a certain threshold while flagging it for further inspection otherwise.

It should be noted that workarounds for calculating a confidence measure in Support Vector Machines have been proposed, for example [78]. But due to the ad hoc nature of these methods and unlike Gaussian Processes they do not cater for the predictive variance in the regression function.

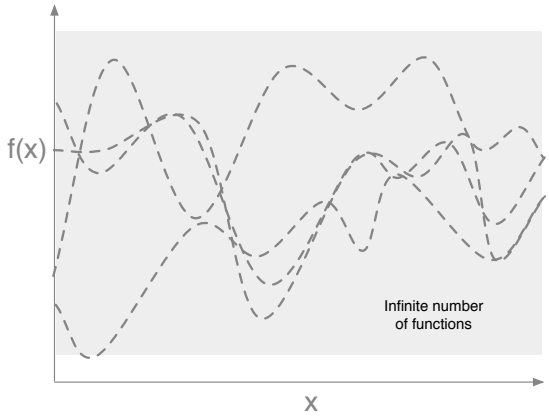
3.3.2 How Gaussian Processes work

As mentioned at the outset of this section, Gaussian Processes map the input training data to a function f that can make predictions for all possible input values. But given a set of training data there is a potentially infinite number of functions that fit the training data points, so which function will yield the most accurate predictions for unknown data values?

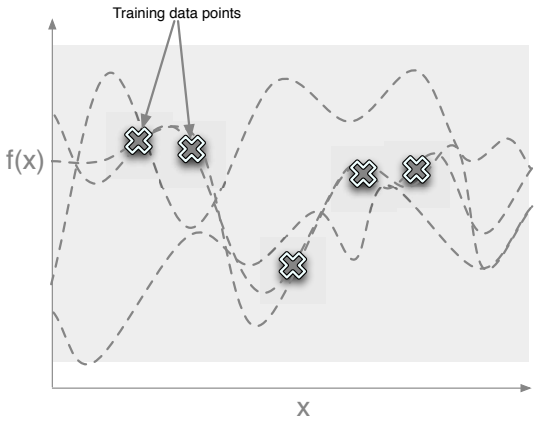
Conceptually, the basic idea [76] is to first narrow down the set of possible functions to those more likely to fit the data at hand, for example because they are smoother. This filtering occurs before any training data is seen and the resulting function set is called the *prior* distribution; a sample is pictured in Figure 3.7(a). The next step is to overlay the training data points onto the prior as shown in Figure 3.7(b). Discarding the functions that do not pass through all the training data points yields the *posterior* shown in Figure 3.7(c).

In the posterior shown in Figure 3.7(c), the solid line is the mean of the remaining (not-discarded) functions – this is the regression function f. The shaded area stretches twice the standard deviation of a given x value and denotes the confidence value at that x – the larger the area the lower the confidence. The confidence around the training data items is high, as expected.

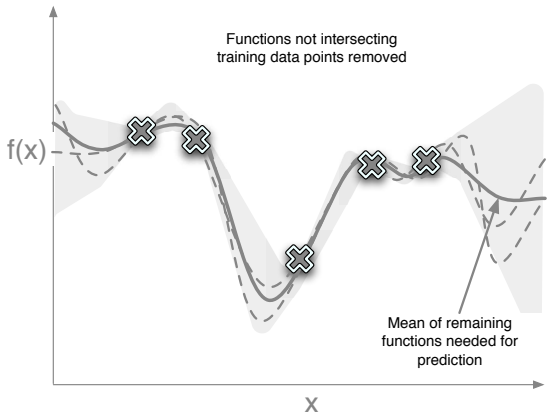
An issue side-stepped in the above explanation is that the prior and posterior distributions consist of an infinite set of (Gaussian) distributions, but to be practically feasible the calculations must be performed in finite time. Loosely speaking, the key concept is to group the infinite set of distributions so that they can be treated as one entity and defined with a finite set of properties. Applying this concept, a *Gaussian distribution* is a distribution defined by its mean and standard deviation. A *Gaussian process*, still loosely



(a) Prior



(b) Prior and training data



(c) Posterior

Figure 3.7: The prior (a) is a pre-selected set of functions likely to fit the task at hand. Overlaying the prior with training data points (b), and discarding those functions that do not pass through all training data points yields the posterior (c).

speaking, can be thought of as a group of (possibly infinite) Gaussian distributions, and that group can be defined by the mean and standard deviation of all the distributions.

Seen from a slightly more formal angle, a Gaussian distribution can be viewed as a function $f(x)$ that operates on scalar values[2] x, while a Gaussian process operates on a set of i Gaussian distributions $f_i(x)$. The mean value of the Gaussian distribution is that of evaluating $f(x)$ for all x; the mean value of the Gaussian process is that of evaluating all $f_i(x)$ for all x.

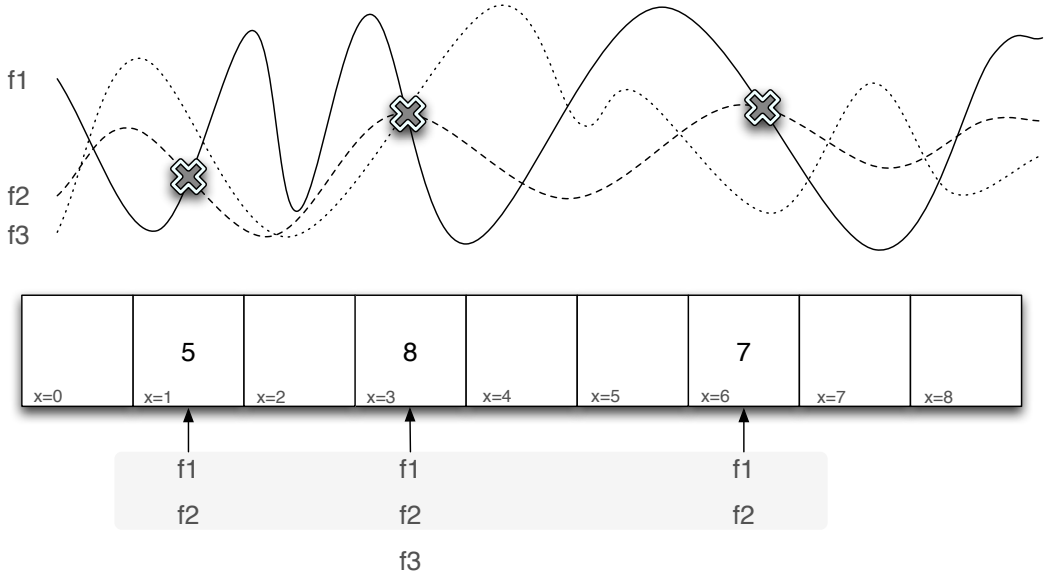


Figure 3.8: Gaussian Processes can be seen as filling a long vector where the entries hold the mean common to all Gaussian distributions in the process.

How does this help with the challenge of computing with infinite prior and posterior distribution sets in finite time? Another way to look at a function $f(x)$ is as a very long vector, where the indices consists of the x values and the entries for each index x are the corresponding $f(x)$. Figure 3.8 extends this idea to Gaussian processes: the functions $f_1 - f_3$ are random samples from the prior distribution. The crosses mark the training data points $\{(1, 5), (3, 8), (6, 7)\}$, with corresponding entries in the function vector.

[2]Or more generally, vector values for multivariate distributions

From a Gaussian process perspective, the vector entries can be seen as buckets to be filled with Gaussian distributions where the mean is 5 when $x = 1$, 8 when $x = 3$, and 7 when $x = 6$. The distributions common in all buckets (shaded grey) are the posterior distributions; in Figure 3.8, only f_1 and f_2 meet that requirement.

Gaussian processes, then, effectively work backwards in computing the posterior distributions by viewing the training data as the mean values common to all distributions in the posterior.

3.3.3 From regression to classification

The previous section explained how Gaussian processes, given a training data set, derive a function that allows predictions of unknown test data. But there is still a missing piece to the puzzle: the output of the regression stage is a continuous function, which can lie in the range $-\infty$ to ∞, and is therefore unsuitable for classification.

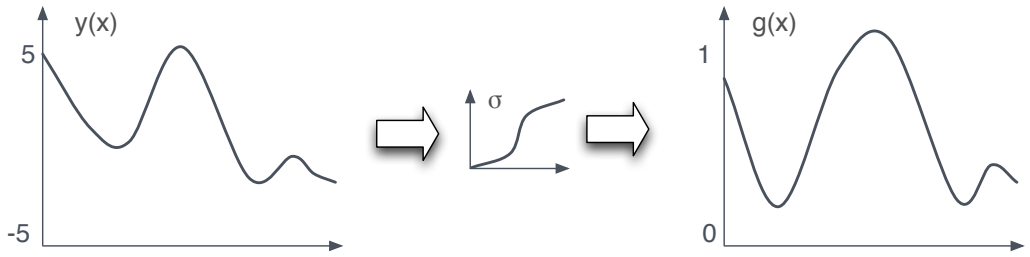


Figure 3.9: The classification function $g(x)$ is obtained by squashing $y(x)$ through the response function σ

The approach taken by Gaussian processes is to apply a *response function* [76]. The underlying idea is that classes are assigned numeric labels, for example benign=0 and malicious=1, and the response function squashes the regression from $-\infty$ to ∞ to fit into the range $[0, 1]$ (Figure 3.9). The squashed result can then be used for classification.

3.3.4 Kernel functions

Like Support Vector Machines, Gaussian Processes belong to the set of kernel machines, with the kernel determining the similarity or closeness between two data points. In the Gaussian Processes literature, kernels are commonly referred to as *covariance functions*, but for consistency this dissertation will continue to refer to them as kernel functions. The kernel heavily influences the smoothness of the prior distributions, and as such it is a key ingredient to the classifier.

The approach taken in this dissertation is to find the optimal kernel for Support Vector Machines, and to use this kernel for comparison in Gaussian Processes. The underlying rationale is that since a kernel measures similarity between data points, for a given data type (such as network traffic) the kernel should work equally under Support Vector Machines and Gaussian Processes. For a rigourous treatment of kernel functions in Gaussian Processes, including the conditions a function must fulfil to be considered a kernel, see [76].

3.4 K-nearest neighbours

K-nearest neighbours [17] is one of the simplest pattern recognition algorithms that nonetheless often performs well. To classify an unknown test data point, the K-nearest neighbour algorithm compares that test data point to all given training data points and performs a majority vote on the classes of the K nearest neighbours (Figure 3.10). The test data point is classified under the class of the winner of that vote.

In contrast to Gaussian Processes, K-nearest neighbours is a classification algorithm by nature. Although K-nearest neighbours is a supervised learning method, unlike Support Vector Machines and Gaussian Processes it defers all calculations until classification.

The naive implementation of the algorithm iterates over all training data points and calculates the distance to the test data, orders the training data points by that distance, and then casts a majority vote on the top K training data points. While easily implemented, this implementation suffers from

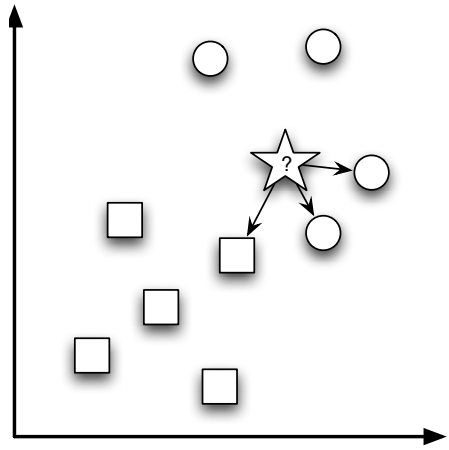


Figure 3.10: The K-nearest neighbour algorithm makes its prediction by finding the K points that are closest to the point in question (the star). It then takes a majority vote of the classes of these K points to determine the prediction for the point in question.

limited scalability as it grows linearly with the number of training data points. For this reason a number of optimisations have been proposed, for example partitioning the feature space [79].

Critical to the predication accuracy is the number of neighbours (K). The optimal value for K depends on the type of data, although in general increasing K should raise the accuracy by mitigating the negative impact of noise on the classification accuracy. On the downside, increasing K blurs the boundaries between the classes. A common method for choosing K is cross-validation, whereby the training data is split into an actual training data set with the remaining data items used to validate the results. Note that for $K = 1$ the algorithm degenerates to simply finding the nearest neighbour.

The K-nearest neighbour works for any types of data points, as long as a function is available that returns the distance between two data points (for some definition of distance). The standard distance function is the Euclidean distance function, which also works well for multi-dimensional data points.

In this dissertation K-nearest neighbours serves two purposes. The first is to determine whether a simple algorithm such as K-nearest neighbours can

be applied to a complex pattern recognition problem such as detecting worm mutations. And the second is to see how K-nearest neighbours compares to the (more complex) machine learning techniques of Support Vector Machines and Gaussian Process in detecting worm mutations.

3.4.1 Properties

K-nearest neighbours offers both positive and negative properties:

- **Simplicity**. Both from a conceptual and implementation point of view the simplicity of K-nearest neighbours makes it an attractive machine learning method.
- **Incremental learning**. A nice side effect of delaying calculations to classification time is that K-nearest neighbours algorithm can learn incrementally as new training data becomes available. Thus, unlike Support Vector Machines and Gaussian Processes, it does not need to be retrained with the entire (original plus new) training data set.
- **Training data percentage sensitivity**. A drawback is that classes with a higher percentage of training data tend to dominate the prediction simply by their larger presence. The dominance could be reduced by taking into account the distances during the majority vote.
- **Noise sensitivity**. Another drawback is that the accuracy of K-nearest neighbours can be severely affected by noise or other irrelevant features, leading to over-fitting. As mentioned in the previous section this can be counterbalanced by increasing the number of neighbours (K), but this has the negative effect of blurring the class boundaries.

3.5 Other machine learning techniques

What about other machine learning techniques? This section discusses some of the other popular machine learning techniques used for classification, and explains why this dissertation decided not to apply them to worm detection.

Linear Discriminant Analysis (LDA) such as Fisher's discriminant analysis [80], tries to find a linear combination of features that separate two or more classes. LDA projects the feature vectors down to a single dimension with a *linear discriminant function* in order to classify the data. This dissertation does not consider LDA because of known difficulties with non-linear data (such as network traffic), and because it has been shown to be outperformed by Support Vector Machines on standard benchmarks [81].

Quadratic Discriminant Analysis (QDA) is a generalisation of LDA that dissects classes with a quadric surface, with the advantage that it does not rely on the assumption that the covariance of all classes are identical. Quadratic Discriminant Analysis has been shown to be outperformed by Support Vector Machines, for example in [82].

Decision tree learning [83] builds a decision tree from data by repeatedly partitioning the input space until a node consists only of a single class. The branches are conjunctions that lead to the classes, and classification involves moving down the branches until hitting a leaf node. Although decision trees are effective and efficient for small data sets, they do not scale well for large data sets, winding up in complicated trees that consume large amounts of memory. Since this dissertation expects to deal with large volumes of network traffic data, decision trees were not further considered.

Another popular alternative classification technique is offered by Artificial Neural Networks [84]. However, in recent years Gaussian Processes and Support Vector Machines have repeatedly been shown to outperform Artificial Neural Networks [77, 85], one of the reasons being that they are less prone to overfitting. Another reason is that Artificial Neural Networks are a parametric method, and as mentioned in Section 3.1.3, this dissertation favours non-parametric models because of their flexibility in that only a similarity measure between objects need to be defined.

3.6 Summary

Machine learning is a sub-discipline of artificial intelligence that involves developing algorithms that automatically recognise patterns using statistical classification. Examples of patterns to be recognised include fingerprints, images, handwriting, voice recordings, or as in this dissertation, worms.

This work focuses on three prominent supervised learning methods:

- **Support Vector Machines** learn patterns by dividing labelled training data into disjoint groups and finding the separating hyperplane that maximises the margin between the hyperplane and the support vectors. Equipped with the hyperplane, the classifier can then label a given test data point based on its position relative to the hyperplane.

- **Gaussian Processes** learn patterns by mapping the input training data to a function that can make predictions for all possible input values. The basic idea is to first narrow down the set of possible functions to those more likely to fit the data at hand, and then overlaying the training data points and discarding the functions that do not pass through all training data points. Gaussian Processes are an attractive machine learning technique since, unlike Support Vector Machines, they return a confidence value as part of their classifications.

- **K-nearest neighbours** is one of the simplest pattern recognition algorithms that surprisingly often performs well. The K-nearest neighbour algorithm classifies an unknown test data point by calculating the distance of this point to all given training data points and performing a majority vote on the K nearest neighbours' classes.

The following chapter ties together this chapter and the previous chapter by showing how machine learning can be applied to worm detection.

Chapter 4

Applying machine learning to worm detection

The last two chapters looked at intrusion detection systems and machine learning as separate, standalone subjects. This chapter merges these two subjects by presenting the design of an intrusion detection system that detects worm mutations using machine learning.

The questions this chapter answers: (i) how to capture network traffic accurately, (ii) how to convert the captured traffic to a format understood by the machine learning classifier, (iii) where the machine learning modules fit into the system, and (iv) where and how such a system can be deployed.

Before answering these questions, this chapter first defines a suitable worm model that will serve as a basis for this dissertation.

4.1 Worm model

For the purposes of this dissertation, a worm can be thought of as a stream of binary data that consists of the following three parts, illustrated in Figure 4.1:

1. **Application specific data** consisting of protocol headers and other control information required to communicate with the vulnerable service.

Figure 4.1: The worm model used throughout this dissertation consists of (i) application specific data required to communicate with the vulnerable service, (ii) exploit code that attacks the vulnerability and injects the malicious code, and (iii) the executable payload which is executed upon successful infiltration of the target.

2. **Exploit code** that attacks the service by exploiting a vulnerability, and that injects the malicious code into the application.

3. **Executable payload** that runs upon successful infiltration of the system by the exploit code; typically, this executable contains potentially malicious actions and necessary information on how to spread further.

With this model in mind, how can one differentiate between two different types of worms (for example Code Red vs. Slapper) and worm mutations (such as Code Red version 1 and 1.1)?

This dissertation defines a *worm mutation* as a worm that carries the same exploit but a different executable payload – that is, mutations of a worm exploit the same vulnerability but execute a different payload on successful infiltration. Figure 4.2 illustrates this difference: the bottom of the figure shows that a worm that is identical to another in every aspect except the exploit code is considered a different worm; on the other hand, worms, that share the same exploit like those in the centre of the figure, no matter how much they differ otherwise, are considered to be mutations of each other.

This definition of worm mutations is based on the understanding that worm authors tend to refine either the damage that their worms cause, or how the worms spread – both refinements that will alter the executable payload. For example, Code Red 1.1 refined the executable payload of Code Red 1 to find more potential victims by improving the random IP address generator. The application specific data, while constant, will be similar to application specific data of legitimate software communicating with the vul-

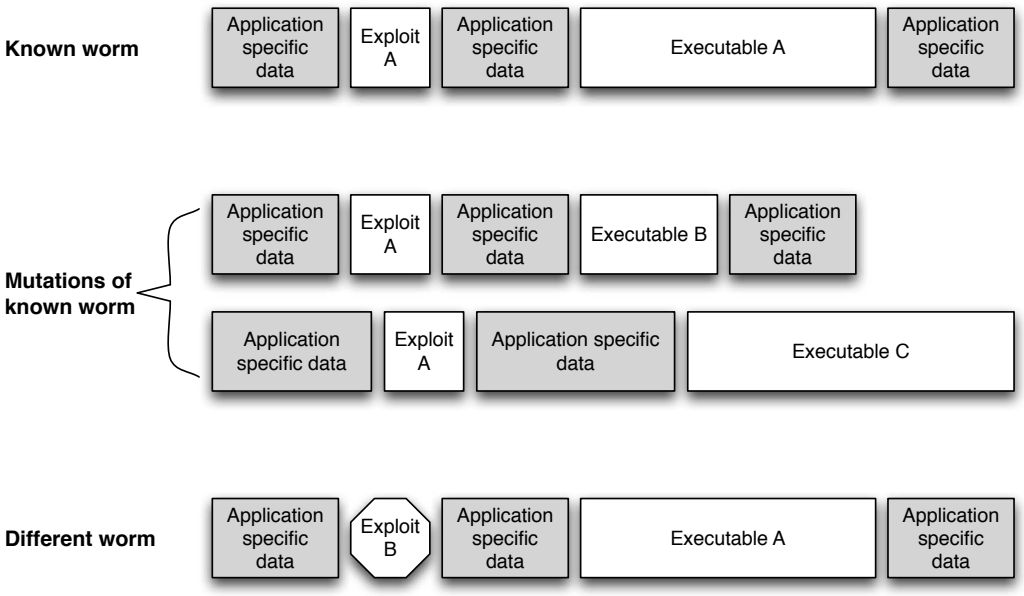


Figure 4.2: This dissertation distinguishes between mutations of the same worm and different worms by their exploit code. Worm mutations share the same exploit code, while different types of worm do not.

nerable service. This leaves the exploit code as (a) the most likely lowest common denominator between worm mutations, and (b) the distinguishing factor to legitimate traffic.

As mentioned in Section 2.2.1, intrusion detection systems typically identify worms by their signature, which can be thought of as a character string or a regular expression, that is present in all mutations of the worm and not in benign flows, and as such uniquely identifies the worm. The signatures in existing intrusion detection systems typically only match part of the exploit (if at all), plus additional code outside the exploit that matches the worm. As such, these conventional signatures will usually not detect mutations.

Note that since this model treats worms as byte streams, the definition of a worm mutation is a binary equivalence one. The model does not cater for the possibility of changing the exploit code's binary while keeping the underlying semantics unchanged, for example by modifying the machine code to use different registers or inserting `NOOP` instructions.

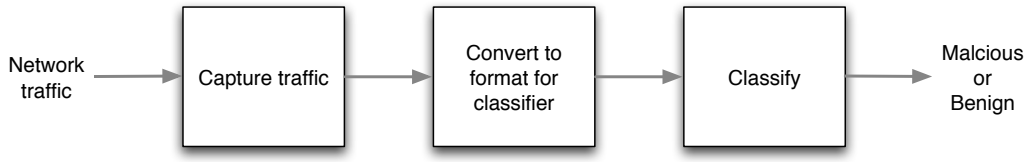


Figure 4.3: High level architecture of the machine learning enabled worm detector.

4.2 Architecture

Figure 4.3 presents a high-level architecture of a machine learning enabled worm detector, consisting of three stages: capturing traffic, converting the traffic into a format that is readable by machine learning techniques, and classifying the flows as malicious or benign.

The rest of this chapter discusses each of these components in turn, and then revisits the architecture with the information from these discussions.

4.3 Capturing network traffic

There are various levels at which the network traffic can be analysed, for example, at the packet level by examining each packet individually. A disadvantage of examining traffic at this level is that malicious contents such as worm exploits can easily be split up over several packets. Even when worm authors themselves do not split their worms, if a large packet traverses a network with a small maximum transmission size, the packet will be split into smaller packets by routers along the way. Moreover, one of the routers may experience congestion, causing the packets to arrive out of order. Thus scanning at packet level is too fine-grained and will potentially miss worms spanning multiple packets.

At the other end of the analysis spectrum is application protocol-level analysis, as is done by Bro [42], discussed in Section 2.2.2.2. Protocol level analysers generate events based on various protocol specific data exchanges. For example, a hypertext transfer protocol (HTTP) analyser could generate an event for every `HTTP GET` request. The events in turn are parsed by policy

scripts that accompany the protocol analysers.

An advantage of application protocol-level analysis is the high abstraction level it offers, allowing decisions to be made with greater certainty. Two problems with application protocol-level analysis are that (i) they are heavy-weight in terms of processing due to the overhead of parsing byte streams into state machines of those protocols, and (ii) to make the most of this setup, protocol-level analysers for all protocols would have to be written.

A third alternative is to analyse the network at the flow level. Flows are typically defined as transmission control protocol (TCP) [86] or user datagram protocol (UDP) [87] packet streams, where a stream is identified by the source Internet Protocol (IP) address, source port number, the destination IP address, and the destination port number. In addition, TCP packets include a sequence number that determines the order in which these packets are to be reassembled, as the TCP protocol allows packets to arrive in arbitrary order to handle the above mentioned queuing delays at routers.

Analysing flows rather than the individual packets that make up a flow ensures that worms can be scanned in their entirety. Flows can then be treated as byte arrays for pattern recognition. Because flow-level analysis is flexible and fine-grained enough to allow worms to be scanned in their entirety without incurring the overhead of parsing protocols, the machine learning worm detector will work exclusively at the flow level. The impact of flow-level analysis on real-time reassembly of these flows on high speed networks is discussed in Section 9.4 as future work.

4.3.1 A sample implementation

To help understand how flows can be reassembled and scanned for signatures concurrently, this section briefly discusses an implementation of a lightweight packet capture and flow reassembly system.

The system consists of three main components: a packet capturer, a flow table, and a flow analyser. The packet capturer simply takes one packet from the wire at a time, using the the *pcap* [88] packet capturing library, and adds it to the flow table, as illustrated in Figure 4.4. The flow table is

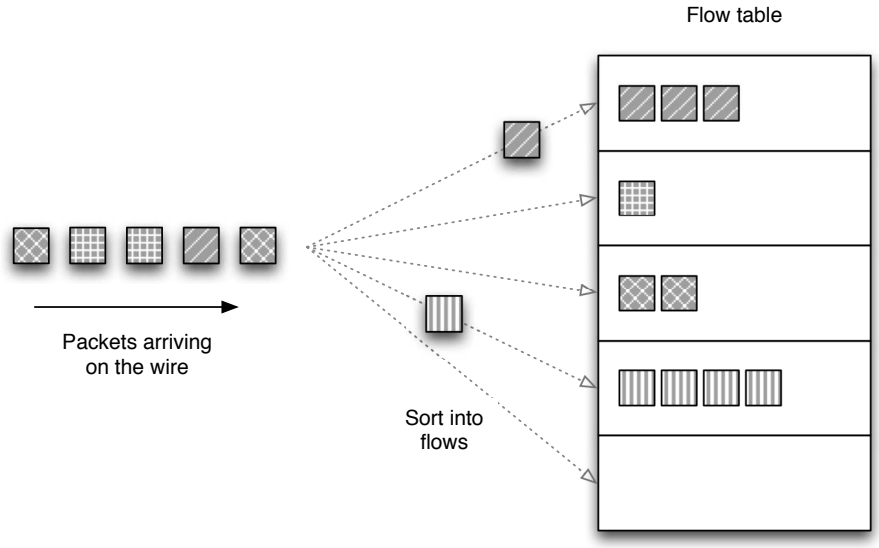


Figure 4.4: Individual packets are sorted into flows as they arrive on the wire.

a data structure that holds packets sorted by flow. The flow analyser runs in its own thread, stepping through the flow table sequentially looking for suspicious flows. In this multi-threaded architecture the flow analyser works concurrently to the packet capturer[1].

The flow analyser encapsulates the logic that labels flows malicious or benign – this is where the machine learning algorithm sits, which could be modularised to allow for multiple worm detectors. The flow table is implemented as a hashtable to achieve lookup and insertion in constant time. The hashtable's key is the 5-tuple <source IP address, destination IP address, source port, destination port, protocol> that uniquely describes a flow.

The flow data structure consists of two packet buffers and a read-write lock, as illustrated in Figure 4.5. The packets are added to the working buffer until the flow analyser arrives to inspect that flow. At this point, the read-write lock is set and from that point any incoming packets land in the holding buffer. When the flow analyser completes, the lock is released

[1]Note that conventional threads such as pthreads [89], are not well suited for high-rate packet capturing, for reasons outlined in [90], which suggests Protothreads [91] as a viable alternative.

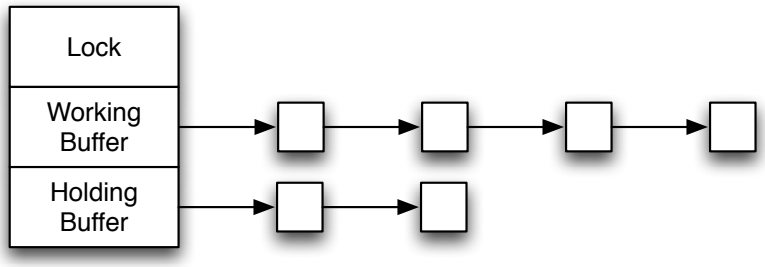


Figure 4.5: A flow consists of two packet buffers and a read-write lock. Maintaining a holding buffer for when the flow analyser examines the flow prevents packets from being discarded.

and the packets moved from the holding buffer to the working buffer. This structure allows packets to be examined without having to block the entire flow or dropping packets.

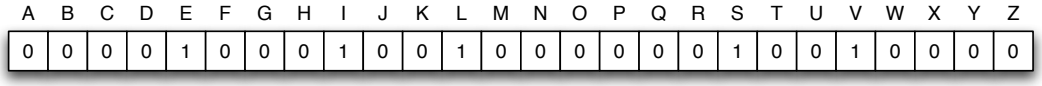
4.4 Feature extraction

As discussed in Section 3.1, machine learning techniques perform statistical analysis on input data that can be captured in *feature (or pattern) vectors*. A feature vector is a row of data where each column represents a specific character in a fixed alphabet. Often, however, data such as images, text, or as in this work, network flows, cannot readily be captured as feature vectors, and features must be extracted explicitly. The remainder of this first section explains in greater detail why features must be extracted from network flows and then exactly how features are extracted.

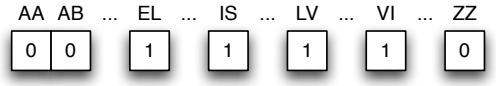
4.4.1 Why features must be extracted

To better understand why features must be extracted, consider a case insensitive feature vector that represents words by using 26 columns, one for each letter in the alphabet. The value held in each column denotes the presence (1) or absence (0) of that letter in the word. The word *ELVIS*, for example, would then be represented by the feature vector shown in Figure 4.6(a).

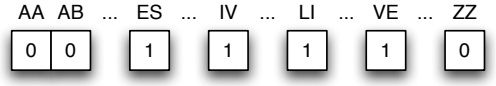
A shortcoming of this simple representation is that any permutation of the



(a) Uni-gram vector for *ELVIS* (or *LIVES*)



(b) Bi-gram vector for *ELVIS*



(c) Bi-gram vector for *LIVES*

Figure 4.6: Feature vectors representing the words *ELVIS* and *LIVES* using uni-grams, and bi-grams.

word *ELVIS* would produce the same feature vector; for instance, *LIVES* and *LEVIS* both collide as feature vectors with *ELVIS*. To overcome this problem, one can use a feature vector that represents all possible combinations of two characters, from AA to ZZ. Figures 4.6(b) and 4.6(c) show that the feature vectors for *ELVIS* and *LIVES* are now different. Feature vectors with one character per entry are called *uni-grams*, those with two characters are called *bi-grams*, and the generalisation to n characters per entry are called n-grams.

The more characters one combines per entry in the feature vector, the more precisely the word is represented, but the greater also the number of columns. In the case of tri-grams, for example, *ELVIS* would be represented by entries at `ELV`, `LVI`, `VIS`. Using 5-grams, the entire word can be represented in one column of the feature vector, albeit at the cost of 26^5 columns in the feature vector. In general, the number of columns grows exponentially with n.

In addition to marking the presence or absence of character combinations, one can keep frequency counts of these combinations in the feature vector

columns. This technique is known as *n-gram extraction*. The frequency counts can be *normalised* (scaled) to the total count to measure the relative probability of a feature vector. Whether normalised feature vectors can better detect worm mutations than unnormalised (that is, with raw frequency counts) feature vectors is investigated in Section 5.3.3.2.

One would expect that, given infinite memory and processing time, the higher the value of n, the more precise the information obtained. This, however, is not the case. Essentially, the probability at which individual n-grams occur is estimated by measuring their relative frequencies. Nevertheless, there will almost always be situations where one encounters an n-gram that has not been seen before. Using frequency counts, this n-gram would be estimated at a nonzero probability, even though its relative probability is zero, a problem known as the *zero frequency problem* [92].

Lower values of n reduce the zero frequency problem. For example, using 5-grams an unlikely word such as *SLIVE* would receive a non-zero probability even though it is improbable that it will reoccur. Using 3-grams, however, *SLIVE* would occupy (among others) the features *SLI* and *IVE*, which are more likely to occur again in words such as *SLIDE* and *HIVE*.

Section 5.1.3 evaluates the trade-offs involved in selecting the value of n and how it impacts worm detection.

4.4.2 Implementing *n*-gram extraction

Since worms are often transmitted in the form of binary executables, the full byte range is used as the alphabet. That is, rather than using just letters `A` to `Z` or even displayable ASCII characters, the byte values 0 to 255 are used, and hence no distinction between displayable and non-displayable characters is made.

The fundamental data structure in the n-gram extractor implementation is an integer array of frequency counts. The length of this array is dependent on the n-gram size, requiring 256^n elements, and is practically feasible only for small values of n. Assuming 4-byte integers, frequency arrays for n-grams sizes 1, 2, 3, and 4 consume 1 KB, 256 KB, 64 MB, and 16 GB of memory

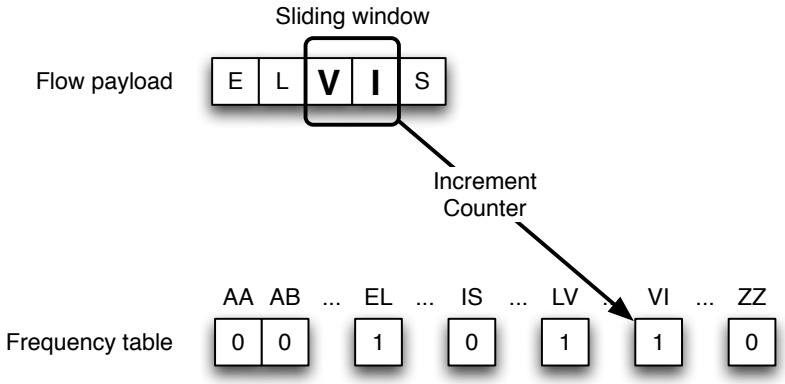


Figure 4.7: Feature extraction using a sliding window and bi-gram frequency table.

respectively. For n-gram sizes of 4 and above, these frequency tables would need to be implemented using data-structures such as hashtables or Bloom filters [93] to be practically feasible. Alternatively, n-grams for higher values could be estimated by, for example, using the $2v$-gram [52] technique.

The frequency table is populated by a sliding window in the captured flow. At each position of the sliding window, its value is read and the corresponding count incremented in frequency table entry, as shown in Figure 4.7. Thus the processing cost of n-gram extraction grows linearly with the flow size.

4.5 Machine learning classifiers

The previous section explained how to transform the network data into a form that is readable by machine learning algorithms by extracting n-gram feature vectors. This section now explains where the machine learning algorithms fit into the picture, and how they detect worms by using feature vectors.

The machine learning algorithms operate in two phases: a *training phase* and a *prediction phase*, as shown in Figure 4.8. In the training phase, the machine learning algorithms are supplied with a set of flows that are marked either malicious or benign, depending on whether they contain a worm. Conceptually, the training phase then derives a pattern that distinguishes mali-

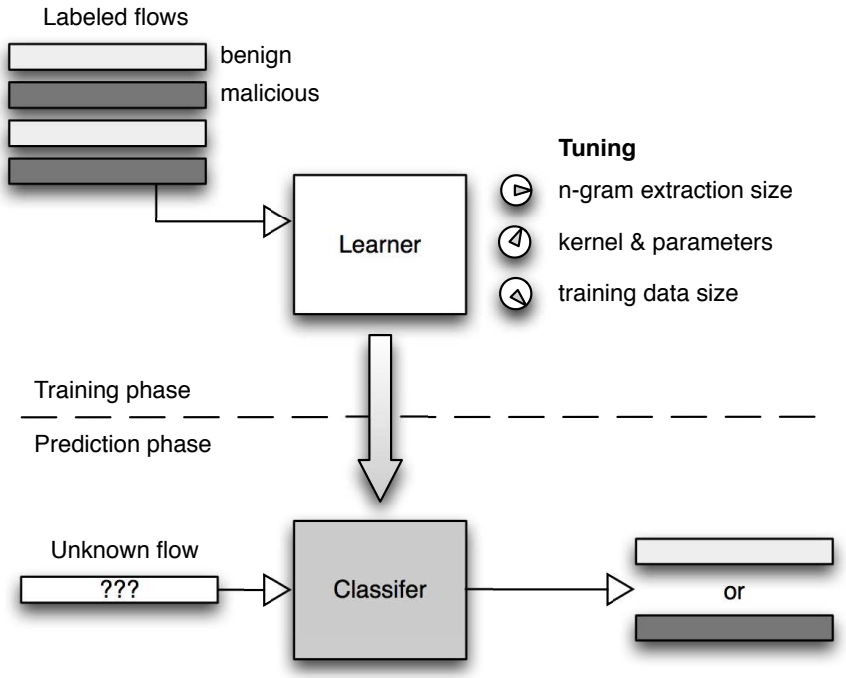


Figure 4.8: Machine learning algorithms train classifiers to distinguish between malicious and benign flows.

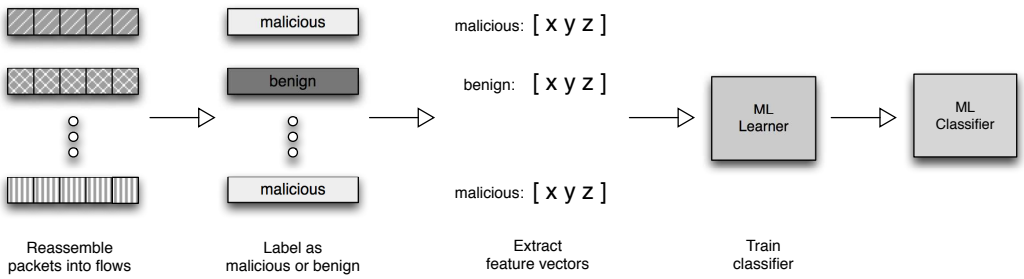
cious and benign flows. For example, for Support Vector Machines this pattern consists of the support vectors that define the separating hyperplane, as covered in Section 3.2.2.

The product of the training phase is a machine learning classifier capable of labelling unknown flows as malicious or benign. In a production environment, the training phase may be performed offline, while the prediction phase classifies live flows as they arrive on the wire.

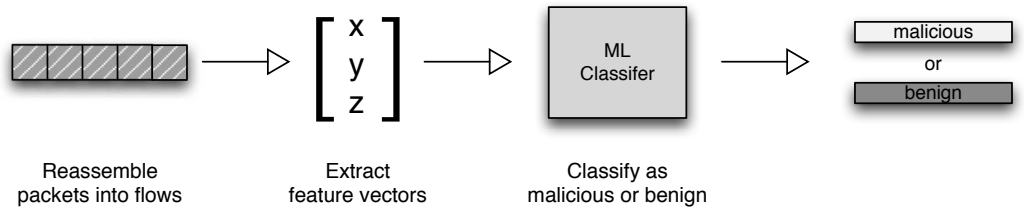
4.6 Architecture revisited

Figure 4.9 summarises the stages involved in both the training and classification phases. The training phase captures network traffic at the flow level, extracts feature vectors using n-grams and then implicitly derives a pattern to distinguish between malicious and benign feature vectors.

Similarly, in the classification stage, which is performed online, feature



(a) Training phase



(b) Classification phase

Figure 4.9: Stages involved in the training and classification phases.

vectors are extracted from captured flows. The classifier then uses information from the training phase to classify feature vectors.

4.7 Deployment

Network intrusion detection systems are typically deployed behind company firewalls, acting as a second line of defence for the local area network, as shown in Figure 4.10. Installing the intrusion detection system behind the firewall makes sense. The firewall's job is to block unwanted traffic, typically by disabling traffic to ports that are known to be out of service or that are intended for internal traffic only. Since the data blocked by the firewall is unwanted, there is no point to load this traffic onto the intrusion detection system since it would only needlessly increase its workload.

Where does the machine learning worm detector fit into the picture? It could be deployed (a) as a module injected into an existing network intrusion

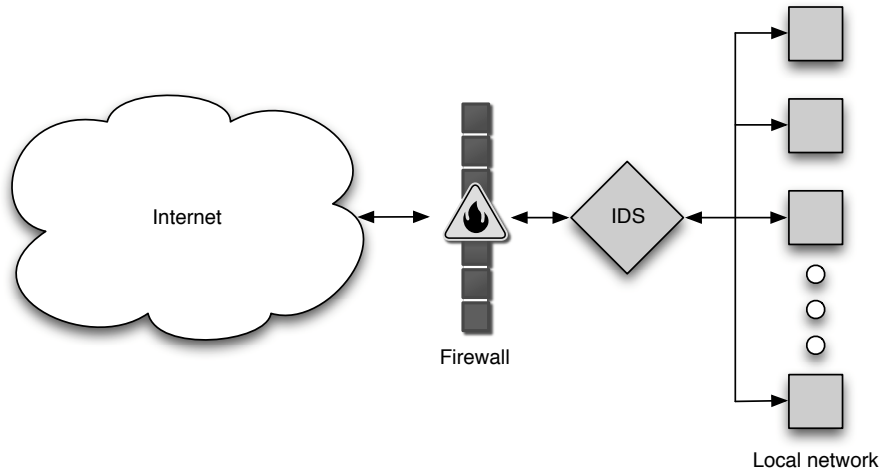


Figure 4.10: Intrusion detection systems are typically deployed behind an organisation's firewall and provide the second line of defence to all hosts on the local area network.

detection system such as Snort or Bro, (b) as a standalone system, or (c) in parallel to other intrusion detection systems, working together for example with majority voting. To keep the worm detector independent of the constraints and weaknesses of other intrusion detection systems, this dissertation builds the worm detector as a standalone unit.

In advanced configurations the firewall and network intrusion detection system work together as a team. In such configurations the network intrusion detection system sends rules to the firewall upon detecting malicious traffic. To keep the worm detector simple, this dissertation does not add co-operation with firewalls to its worm detector.

4.8 Summary

This chapter described how machine learning techniques can be applied to detect worm mutations:

- **Worm model**. The worm model used in this dissertation and defined in this chapter is a stream of binary data that consists of three parts: application-specific data, an exploit code, and an executable payload.

A worm mutation is as a worm that exploits the same vulnerability as the original worm, but executes a different payload on infiltration.

- **Capturing network traffic**. Network traffic is captured at the flow level rather than at the packet level so that worms spanning multiple packets can be scanned for in their entirety.

- **Feature extraction** is performed on captured flows to create feature vectors that the machine learning algorithms can understand. The feature vectors are n-grams, with optimal value of n to be determined in later chapters.

- **Machine learning algorithms** operate in two phases. In the training phase, the machine learning algorithm is supplied with a set of flows that are labelled as either malicious or benign from which it implicitly deduces a distinguishing pattern. The product of the training phase is a classifier capable of classifying unknown flows.

- **Deployment** of the worm detector is intended behind company firewalls, acting as a second line of defence for the local area network.

This chapter set the scene for the following chapters, which investigate whether the machine-learning enabled worm detector introduced in this chapter can successfully detect worm mutations.

Chapter 5

Support Vector Machines for worm detection

The previous chapter introduced an architecture that uses machine learning to detect worm mutations. This chapter puts this architecture to the test.

Specifically, this chapter investigates whether Support Vector Machines (SVMs) are suitable for detecting worm mutations. It compares the efficacy of various SVM configurations and associated kernel functions in classifying worm mutations. This chapter will show that Support Vector Machines are suited for detecting mutations of known worms, and that the optimal configuration is a linear kernel with unnormalised bi-gram frequency counts as input.

5.1 Experiment design

The design of this chapter's experiments fall into three parts: (i) the worm mutations used for training and test data, (ii) the feature extraction from network flows, and (iii) finding the optimal configuration of the Support Vector Machine.

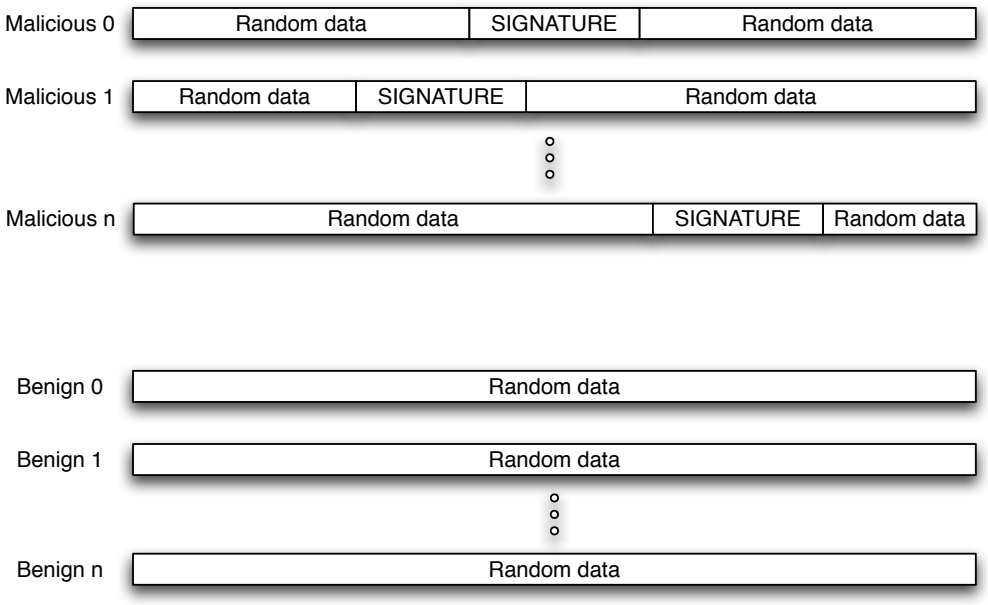


Figure 5.1: Synthetic flows. Malicious flows consist of random data but contain the worms signature at an arbitrary position, whereas benign flows are just random data.

5.1.1 Synthetic mutations

A wide spectrum of training data is critical to the SVM's success – the wider the spectrum, the more likely the training phase can detect a pattern common to the worm mutations. However, for a thorough investigation the spectrum of available worm mutations is too narrow, which is why the experiments in this chapter rely on synthetically generated flows. By synthetically generating flows, the experiments gain not just control over the number of mutations, but also their size and the contents of their payload.

The synthetic flows will either be malicious or benign, as shown in Figure 5.1. *Malicious flows* are where the worm's signature has been injected into the random data, effectively simulating a worm mutation. Recall from the worm model definition in Section 4.1 that a signature is a string or regular expression that identifies the worm. To be consistent with this worm model, the signatures injected into the flows are part of the exploit code. *Benign flows* are simply random data simulating normal traffic.

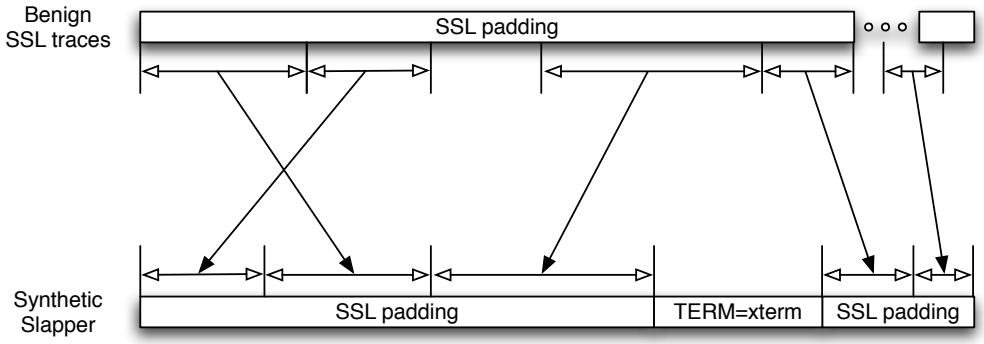


Figure 5.2: Synthetic worms loosely based on the Slapper worm consist of random-sized chunks of SSL traces used as padding.

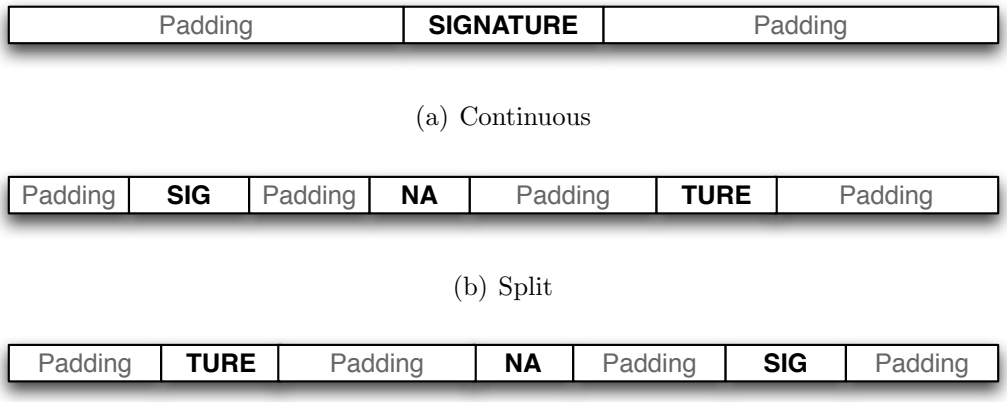
The challenge in using random padding lies in finding a random distribution that realistically emulates normal traffic. To overcome this problem, the synthetic flows are padded with randomly selected chunks of benign flow traces. To verify this approach, the classifier's performance on flows using padding generated with a uniform random number generator and padding generated from benign flow traces are compared.

Key to the validity of this chapter's experiments is that although the signature of a synthetically generated worm is known, this information is not passed on to the SVM. The Support Vector Machine is only told whether the flow is benign or malicious, and must deduce the signature itself.

5.1.1.1 Synthetic Slapper worm

To make the synthetic flows realistic, the injected worm signature is loosely based on the Slapper [31] worm, which has a widely publicised and fairly simple signature. The Slapper worm allows the attacker to run arbitrary and potentially malicious code on the infected host before building a botnet[1] of all infected hosts. The Slapper worm exploited an OpenSSL [32] vulnerability in older versions of the Apache web server, and has a signature that carries

[1]A *botnet* is a network of infected hosts that can be remote controlled to perform malicious tasks.



(a) Continuous

(b) Split

(c) Jumbled

Figure 5.3: Continuous, split, and jumbled signatures.

the string `TERM=xterm`[2]. The synthetic flows were generated from randomly selected benign SSL traces, with the string `TERM=xterm` injected into the malicious flows (Figure 5.2).

5.1.1.2 Continuous, split, and jumbled signatures

Thus far this chapter has assumed that all signatures are *continuous* – a single block of data as shown at the top of Figure 5.3(a). Given that continuous data can be easily matched in network flows, worm authors have attempted to distort their worms to escape detection.

The easiest way to distort the worm is by splitting the continuous blocks at arbitrary locations, resulting in *split signatures* as shown in Figure 5.3(b). A further refinement is to randomly rearrange the split blocks, leaving a *jumbled signature* as depicted in Figure 5.3(c).

[2]Note that this is an oversimplified version of the signature used by Snort and Bro. Both additionally consider some of Slapper's other actions, such as probing servers and exhausting their connection pools.

5.1.1.3 Corrupted signatures

How does the classifier handle corrupted signatures? To answer this question, a test set was created containing corrupted (or dirty) signatures. To create this test set, parts of the signatures in the test set were randomly corrupted to various degrees using a uniform random number generator. The classifier's accuracy distinguishing malicious from benign flows was then observed.

The detector sits at the TCP layer of the network stack, meaning it deals with fully reassembled TCP flow payloads and therefore should not have to worry about data corruption caused by communication errors, such as dropped, reordered, or corrupted packets. Nonetheless there are two good reasons to investigate how the classifier copes with corrupted signatures.

First, polymorphic worms often mutate themselves by corrupting their signatures. Recall that polymorphic worms are worms that try to evade detection by mutating themselves from hop to hop as they traverse the network. Understanding, then, how the classifiers cope with corrupted signatures would give insight as to how well they would be able to detect polymorphic worms.

And second, since a corrupted signature is by definition a worm mutation, investigating corrupted signatures gives further insight into how robust the classifiers are in detecting worm mutations.

5.1.2 Kernel configuration

The choice of kernel is arguably the most important Support Vector Machine parameter. Section 3.2.3 selected three candidate kernels to test: the linear kernel, the radial basis function (RBF) kernel, and the string kernel. Table 5.1 summarises the advantages and disadvantages of these kernels.

Within each kernel, optimal values for the following parameters need to be determined:

- The linear kernel has only a single parameter that can be tuned – the C value. This parameter determines the softness of the margin classes. The softer the margin, the more erroneously placed data points

Kernel	**Advantages**	**Disadvantages**
Linear	• simplest kernel • only one hyperparameter	• works only in input space
RBF	• good general performance • implicitly maps data to higher dimension	• tune two hyperparameters vs. only one for linear
String	• optimised for string matching	• slow compared to linear and RBF

Table 5.1: Comparison of the linear, radial basis function (RBF), and string kernels.

it tolerates. Essentially it is the trade-off between fitting the training data and maximising the margin.

- There are two parameters that can be tuned in RBF kernels: C and *gamma*. The RBF's C parameters serves the same purpose as the linear kernel's. The *gamma* value controls the width of the RBF kernel, and hence the smoothness of the RBF.

- In the string kernel the two key parameters are the substring length, and whether to consider all lengths up to the substring length or just the given length itself. The effect of varying the substring length relative to the signature length for both fixed and variable-length strings is investigated.

After determining optimal parameter values for the individual kernels, their performance is compared in terms of classification accuracy, training time, and prediction time. Once these experiments yield an optimal kernel, the optimal n-gram size and whether normalising extracted features improves classifier performance is investigated.

5.1.3 Feature extraction

As discussed in Section 4.4, feature extraction for SVMs involves extracting n-gram frequency counts from the flow payloads. One of this chapter's aims is to investigate the trade-offs involved in selecting the value of n.

Since this work's ultimate goal is to detect worms in a real-life environment – facing limited memory and processing power – this work restricts itself to 1-, 2-, and 3-grams, consuming 1 KB, 256 KB, 64 MB respectively[3]. Higher values of n and alternative representations are picked up again in future work (Section 9.3).

Of the three candidate kernels introduced in Section 3.2.3, only the linear and RBF kernel require n-gram extraction since both require vectors as inputs, whereas the string kernel works with character (byte) strings natively.

5.1.4 Training data size

As mentioned previously, a classifier's accuracy can be greatly affected by the spectrum of the training data. By employing synthetic worms the experiments are not limited by the number of available worm mutations, which leads to the question how sensitive the classifier accuracy really is to the training data spectrum.

To answer this question, the experiments gauge the training data size to test the impact on classifier accuracy. Ideally the classifier should demonstrate good accuracy even for small training data sets, as the number of available worm mutations to serve as training data in the wild will be limited.

5.1.5 Data-to-signature ratio

The worm model in Section 4.1 defined a worm mutation as a worm that carries a different executable payload but uses the same exploit to gain access into the remote system. But the executable payload is not limited to be the same size as that carried by the original worm, which means that mutations

[3]These size values are calculated assuming 4-byte integers

Worm	**Payload** (bytes)	**Signature** (bytes)	**Ratio**
Code Red	4039	396	9:1
Slammer	404	16	24:1
Witty	1184	64	18:1
Distcc	1740	307	6:1
Minishare	8600	2252	4:1

Table 5.2: Approximate data-to-signature ratios for past worms. The values for Distcc and Minishare are based on the average sizes of the mutations generated in Chapter 7.

of the same worm may have different sizes. Rather than compare worm mutations in terms of their absolute size, this dissertation compares worms based on the amount of total data relative to the size of exploit code – the *data-to-signature ratio*.

Worm mutations in the real world thus have different data-to-signature ratios, and so the experiments investigate the impact of mixed data-to-signature ratios in training and test data sets on classifier accuracy. The expected result is lower accuracy for higher ratios as the large amounts of padding data swamp the signature.

Table 5.2 shows approximate data-to-signature ratios for the Code Red [6], Slammer [8], Witty [37], Distcc [94] and Minishare [95] worms. All the worms have a data-to-signature below 25:1; this chapter will use this figure as a yardstick when evaluating the performance of the classifier.

5.1.6 False alarm rates

Thus far, accuracy served as the measure of how well flows are classified. While accuracy is an intuitive measure, it does not tell the whole story. In intrusion detection, the false alarm rate is important, since falsely classifying flows as malicious (*false positive*) means that legitimate traffic is blocked. Conversely, falsely classifying flows as benign (*false negative*) means that

worms pass undetected.

Receiver operator characteristics (ROC) [96] curve analysis presents a way of quantifying the trade-off between the detection rate and the false alarm rate. ROC curves have their roots in signal detection and medical decision-making, and have recently become a popular way of analysing machine learning classifiers. Appendix B provides a detailed overview of ROC curve analysis.

The ROC curve for a classifier is generated by plotting the true positive rate versus the false positive rate at various confidence intervals. The true positive rate is calculated by dividing the number of true positives by the total number of positives; similarly the false positive rate is calculated by dividing the number of false positives by the total number of negatives. A single classification will produce a single point in the ROC space and a curve is obtained by varying the classifier's confidence (see also Appendix B).

The performance metric that can be obtained from these curves is the total area under the curve (AUC) [97]. The larger the area, the better the classifier.

5.2 Experimental setup

All experiments were performed on an Intel Xeon with a 2.40GHz CPU and 1GB of RAM. Libsvm [98] version 1.8 was used as the Support Vector Machine implementation for linear and RBF kernels, and libs [99] version 1.3, a libsvm modification, was used for string kernels.

As a baseline for each experiment, a training set of 100 flows, half malicious and benign, was used. The implications of this training data size is investigated in Section 5.3.4 below. A test set of size 1000, again half malicious and half benign flows was used; each experiment was repeated 50 times[4] with different data sets to obtain averages and standard deviations.

Note that the equal ratio of malicious to benign flows in the test set is

[4]The number of iterations (repeats) was investigated, showing that the accuracy levels stabilised at 50 iterations.

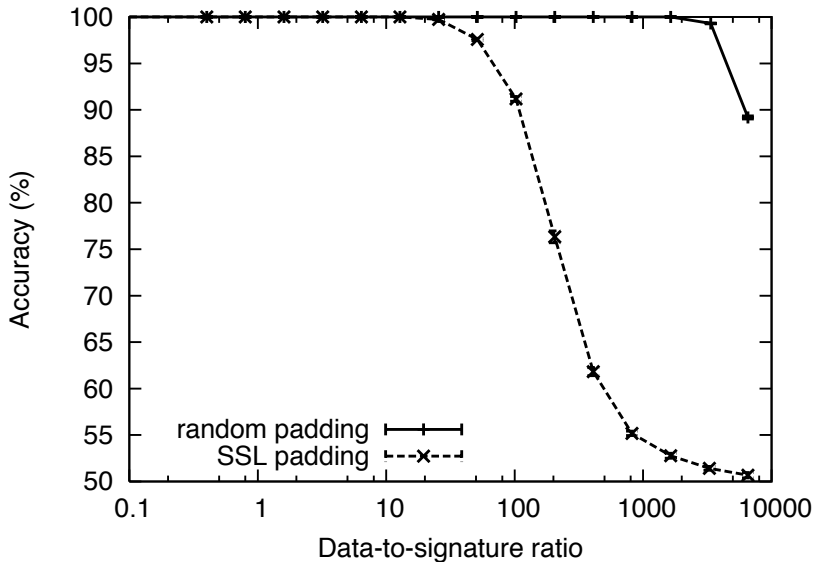


Figure 5.4: Prediction accuracy at various data-to-signature ratios for flows padded with random and SSL data.

not representative of real network traffic, where the proportion of benign flows is likely to be much higher. However, the aim of the experiments is to determine with which accuracy the classifier can classify unknown network flows, and for an unbiased result there should be an equal probability that it receives either a malicious or a benign flow.

5.3 Analysis

This section presents the results of the experiments described in the previous section.

5.3.1 Synthetic mutation format

Figure 5.4 shows the prediction accuracy at various data-to-signature ratios for flows padded with random and SSL data. As expected, it is much harder to distinguish between malicious and benign worms when padded with SSL data. A likely explanation is that the SSL padding tends to have large chunks repeated among all flows, thereby diluting the signature. Using uniform

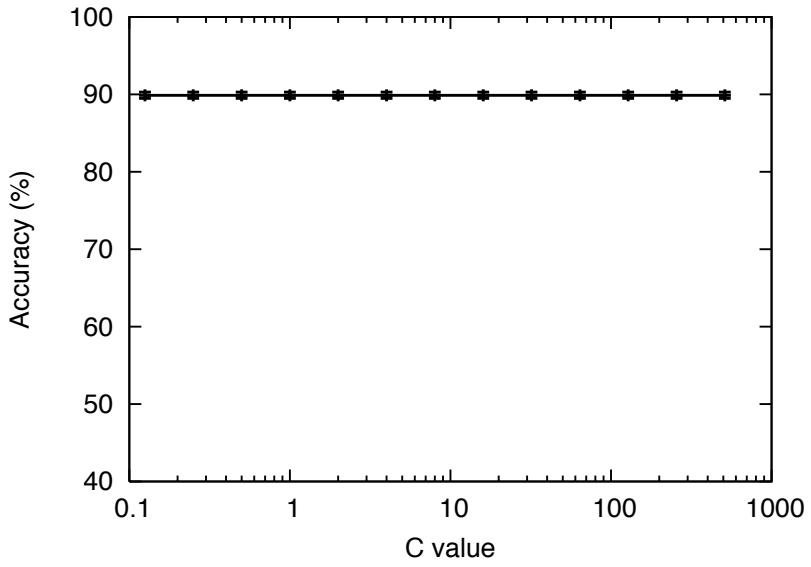


Figure 5.5: Prediction accuracy at various C values using the linear kernel and a data-to-signature ratio of 100 : 1.

random data as padding, on the other hand, highlights the signature.

Since SSL padded flows are both more realistic and constitute a harder problem, henceforth only results for SSL padded data are shown.

5.3.2 Choosing the optimal kernel

This section answers the question as to which is the optimal kernel for the SVM. It does so by first examining each of the linear, RBF, and string kernel in turn to find each kernel's optimal configuration before comparing the kernels themselves.

5.3.2.1 Configuring the linear kernel

The effect of varying the C parameter was investigated by performing a 10-fold cross validation. A dataset of 1000 entries was split into 10 equal parts, of which 1 was used to train the classifier and the other 9 to test it. This was repeated with a range of C values.

As the graph in Figure 5.5 shows, altering the C value did not affect the classifier's performance; henceforth Libsvm's default C value of 1 will be

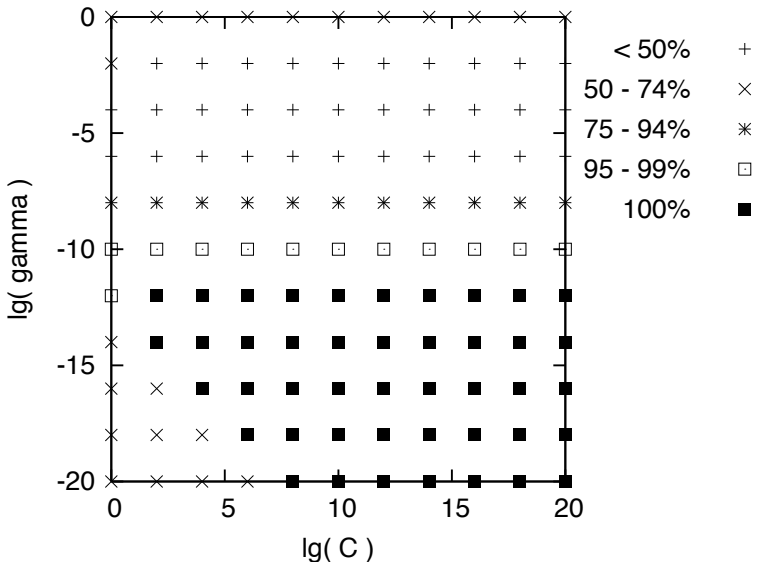


Figure 5.6: RBF kernel grid search plotting gamma against C.

used. The results indicate that the data classes are totally linearly separable.

5.3.2.2 Configuring the RBF kernel

A grid-based cross validation was performed to determine the optimal parameter values for the training data. That is, an exhaustive range of C and *gamma* combinations were tested, each time using a 10-fold cross validation as used in previous case with the linear kernel's C value. Libsvm's [98] *grid.py* script was used to perform this cross-validation.

Figure 5.6 shows results obtained from this grid search using a data set with a data-to-signature ratio of 25:1, with C ranging from 2^0 to 2^{20} and *gamma* ranging from 2^0 to 2^{-20}. All points that achieved 100% accuracy are optimal parameter combinations for the RBF kernel; henceforth a parameter combination of $C = 2^4$ and gamma$= 2^{-14}$ will be used.

5.3.2.3 Configuring the string kernel

Figure 5.7 shows the effect of varying the substring length relative to the signature length for both fixed and variable-length strings. The results show that fixed length substrings and variable length substrings achieve similar

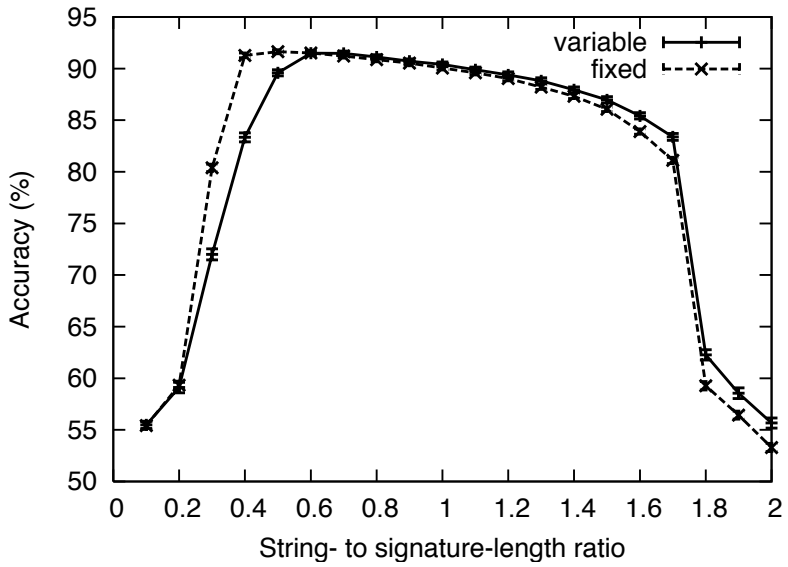


Figure 5.7: String kernel accuracy at various string-length to signature-length ratios comparing fixed and variable length strings. The data-to-signature ratio was 100:1.

prediction accuracies, with the performance peaking at a string-to-signature ratio of approximately 0.5. Given the synthetic Slapper's 10 character signature, this corresponds to a substring length of 5.

Although fixed and variable length substrings achieve comparable levels of accuracy, the processing time with variable-length strings is orders of magnitude greater than using fixed-length strings, due to the number of substrings increasing exponentially when considering all possible substrings up to a certain length rather than just the substring length itself.

Based on the above findings the optimal string kernel configuration – and the one used henceforth – is a fixed length substring to signature ratio of 0.5.

5.3.2.4 Comparing prediction accuracy

Now that the the optimal parameter settings for the individual kernels has been found, their classification accuracy can be compared. The graph in Figure 5.8 shows that all three kernels show similar accuracies. By a small margin the string kernel exhibits the best accuracy, closely followed by the

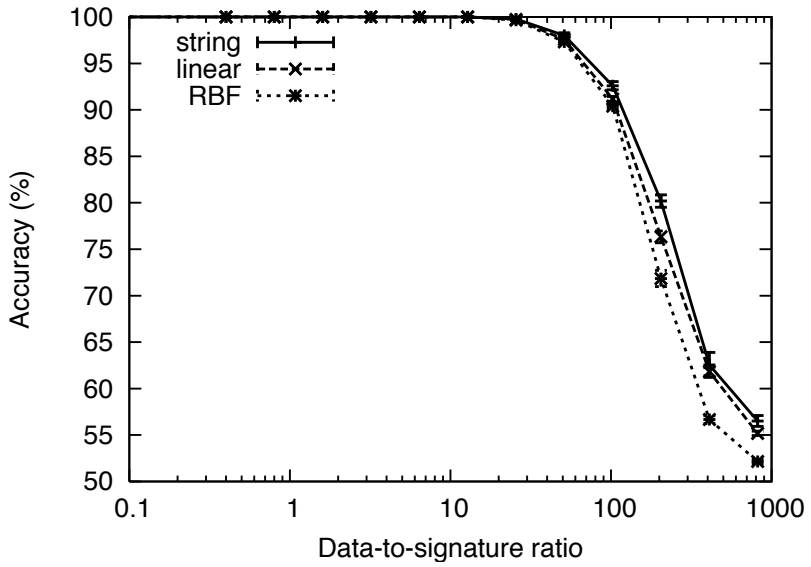


Figure 5.8: The prediction accuracy at various data-to-signature ratios for the linear, RBF, and string kernel.

linear kernel, and finally the RBF kernel. The accuracy trails off at 50% since at this point the classifier is randomly guessing whether a flow is malicious or benign, and hence is only as accurate as flipping a coin.

Since the results show no clear winner in accuracy, the performance in training and prediction times are compared next.

5.3.2.5 Comparing training time

The graph in Figure 5.9 compares the time taken to train the classifier. As expected, the time taken increases with data-to-signature ratio due to the increase in total data. The results show that training SVMs using linear and RBF kernels is considerably quicker than training with a string kernel.

5.3.2.6 Comparing prediction time

Arguably, the training time is not the key performance measure, since one can train classifiers offline; prediction (classification), on the other hand, happens online and is thus pivotal in the choice of kernel. Figure 5.10 shows the time taken to predict a single flow's class. As was the case with training time, the

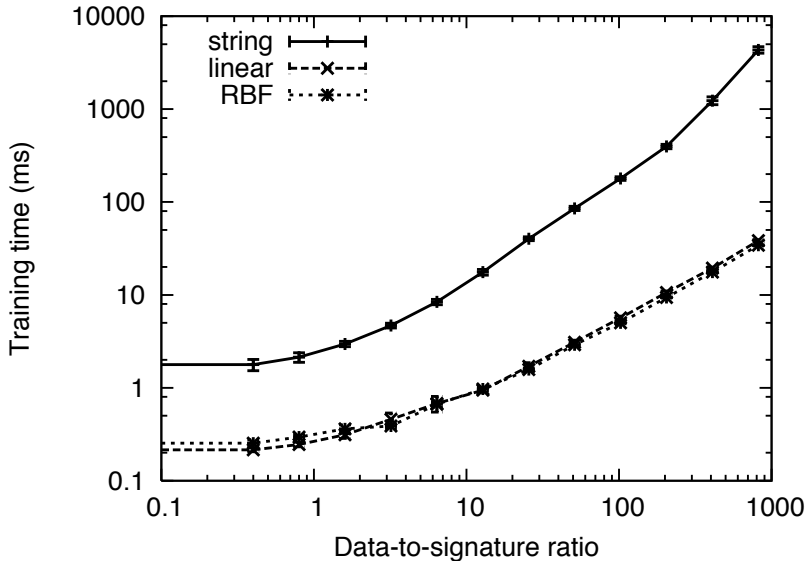


Figure 5.9: Time taken to train the linear, RBF, and string kernel. This is the time for a single flow, and must be multiplied by the training data size for an estimate of the total training time.

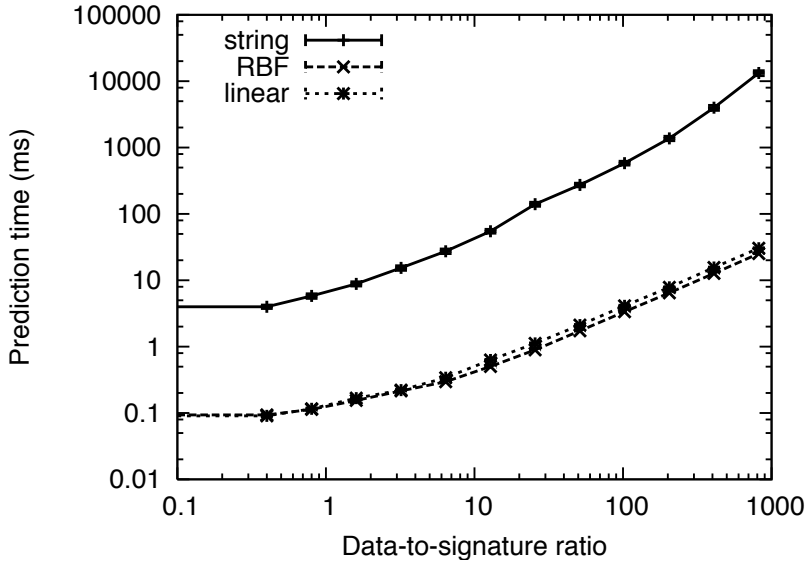


Figure 5.10: Time taken to predict a flow's class using the linear, RBF, and string kernel.

string kernel's prediction time is between one and two orders of magnitude higher than for the linear and RBF kernels.

5.3.2.7 Summary

Which kernel is best? The string kernel proved only marginally more accurate than its competitors, leaving classification time as the tie breaker. This takes the string kernel out of the race since it is much slower in both training and classification, and leaves only the linear and RBF kernel. Since the RBF kernel is sensitive to it's parameter values (C and $gamma$) and is more complex than the linear kernel, the linear kernel was selected as the optimal kernel for this scenario.

5.3.3 Feature extraction

Having established the linear kernel as the kernel of choice, the next step is to examine the other factors that affect a classifier's performance. This section finds the optimal n-gram size, and determines whether normalising the frequency count improves accuracy.

5.3.3.1 Optimal n-gram size

The graph in Figure 5.11 shows the prediction accuracy for uni-grams, bi-grams and tri-grams (using a linear kernel). It illustrates that prediction accuracy increases with higher values of n, as expected. Using tri-grams, all flows at a data-to-signature ratio of approximately 12:1, up to a data-to-signature ratio of 100:1 are classified with 90% accuracy.

Unsurprisingly, uni-grams perform considerably worse than bi-grams and tri-grams. The accuracy gain of tri-grams over bi-grams is negligible, which means the best choice for this scenario is to use the smaller of the two for performance reasons.

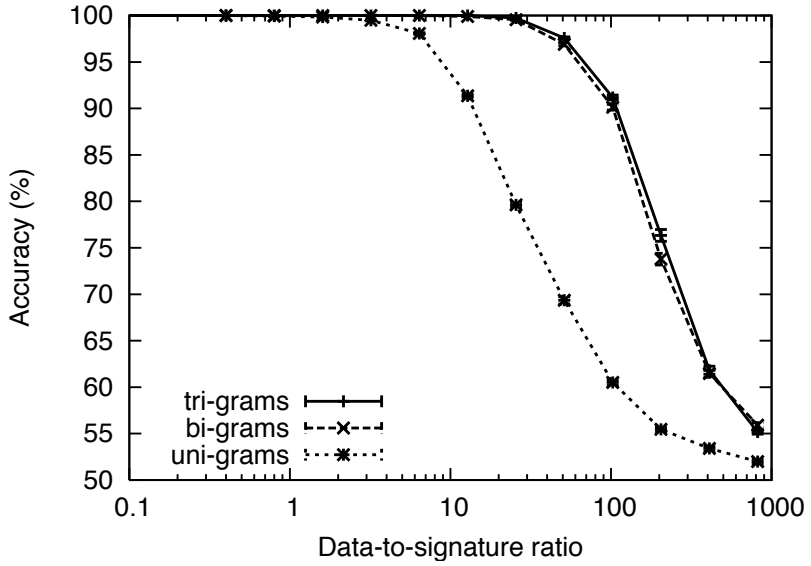


Figure 5.11: Prediction accuracy at various data-to-signature ratios for uni-grams, bi-grams and tri-grams.

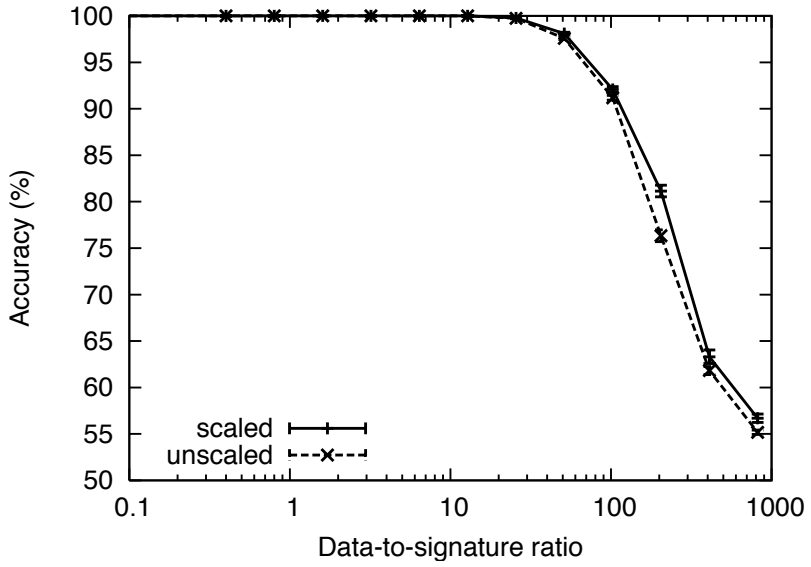


Figure 5.12: Prediction accuracy versus data-to-signature ratios for normalised (scaled) and unnormalised (unscaled) data.

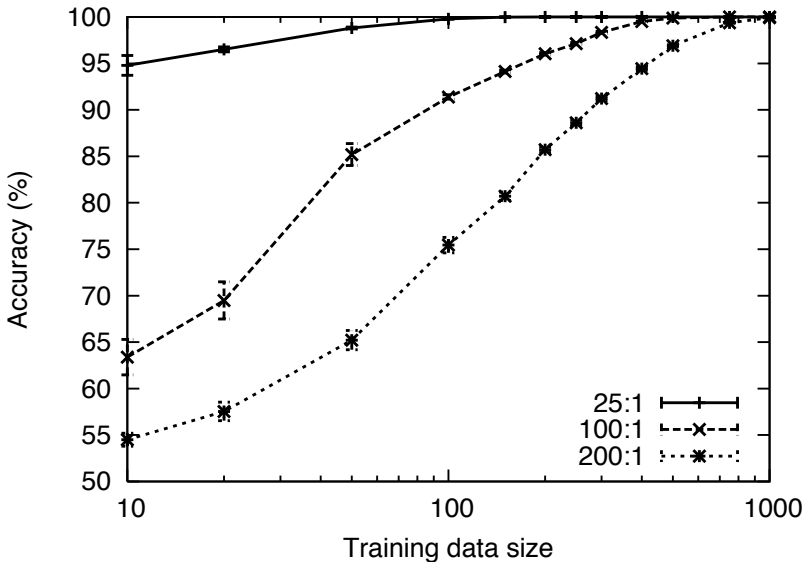


Figure 5.13: The effect of training data size on prediction accuracy for various data-to-signature ratios.

5.3.3.2 Impact of normalising frequency counts

The graph in Figure 5.12 compares the accuracy of unnormalised versus normalised frequency counts. Normalising improves accuracy as expected, but only marginally so. Since normalising comes at a great computational cost and with hardly noticeable accuracy gains, all further experiments are conducted with unnormalised feature vectors.

5.3.4 Training data size

The graph in Figure 5.13 shows how accuracy varies with training data size. As anticipated, the accuracy increases with training data size: the larger the training data size, the higher the chance of the worm's distinguishing signature surfacing as the common pattern.

The results are highly encouraging. As shown in Table 5.2, worms tend to have a data-to-signature lower than 25:1, for which the classifier achieves above 90% accuracy even for the smallest training data size. Accuracy climbs with increasing training data size, reaching 100% for training data sizes of 100 and above.

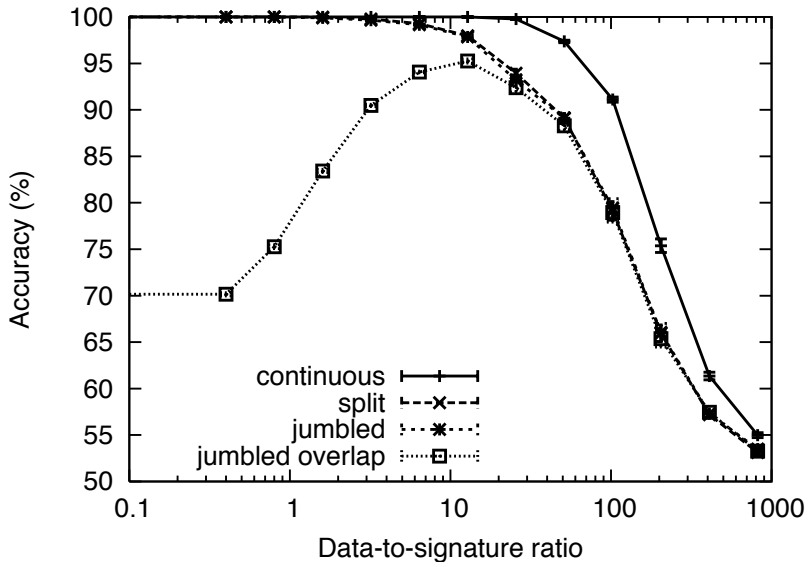


Figure 5.14: Prediction accuracy at various data-to-signature ratios for different synthetic worm models.

Regrettably, in a real-life setting, one cannot pick the training data size as the number of available traces is constrained[5]. However, this information can be used to guarantee certain levels of accuracy. Consider as an example, that an accuracy of at least 90% needs to be achieved and that flows with a data-to-signature ratio of 100 are used. This graph can then be used to determine that a training data size of 100 is required to attain the desired accuracy level.

5.3.5 Continuous, split, and jumbled signatures

The graph in Figure 5.14 shows the prediction accuracy for the different signature classes, where the classifier was trained solely with continuous signatures. The graph reveals that the classifier is still able to detect distorted signatures, even when the signature is split or jumbled, albeit at a slight cost of accuracy. Interestingly, split and jumbled signatures are recognised with near identical accuracy, showing that the classifier is equally robust to the

[5]Chapter 7 suggests a workaround of this constraint by generating worm mutations automatically from a given source worm.

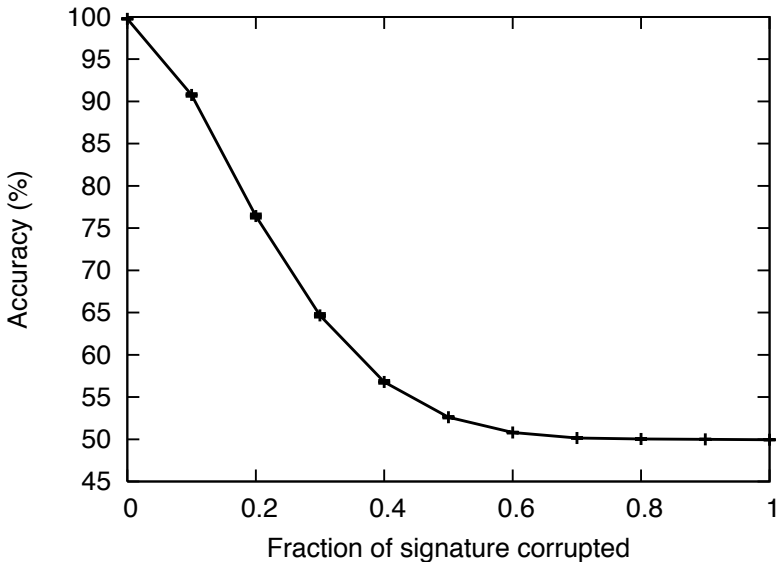


Figure 5.15: Prediction accuracy at various degrees of signature corruption. Data-to-signature ratio is 25:1.

location and order of the worm signature.

Unlike jumbled signatures, which consist of split signature fragments that have been reordered, jumbled with overlapping signatures do not respect other signature fragments' boundaries, and hence overlapping may occur. This explains the bell shape of its graph. For small data-to-signature ratios, the chances of overlapping fragments is significantly higher, thereby corrupting the signature's characters. This is confirmed by the relatively poor accuracy for data-to-signature ratios of less than 10. Beyond that size overlapping is mitigated, and the accuracy curve eventually approaches that of split and jumbled signatures.

5.3.6 Corrupted signatures

Figure 5.15 plots accuracy against various degrees of corruption and shows that the classifier remains reasonably accurate up to 10% corruption, after which it declines steeply. These results are encouraging since they show that the classifiers do not fail abruptly when a signature is corrupted, hinting at resilience to polymorphic worms.

5.3.7 Mixed data-to-signature ratios

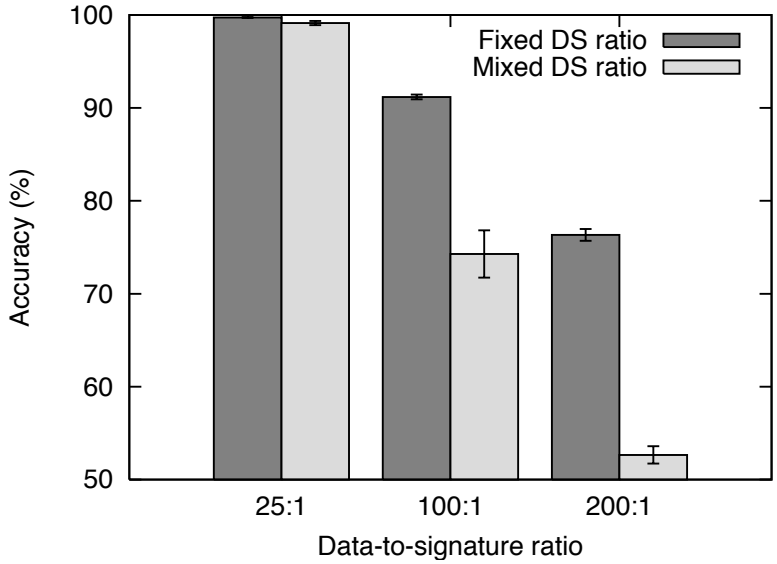


Figure 5.16: Prediction accuracy at various data-to-signature ratios comparing classifiers trained and tested with a single data-to-signature ratio and mixed data-to-signature ratio.

The graph in Figure 5.16 compares uniform with mixed data-to-signature ratios. The mixed data-to-signature ratios were normally distributed so that in each comparison, the mean of the mixed data-to-signature ratios was set equal to the corresponding uniform data-to-signature ratio. The standard deviation for the mixed data-to-signature ratios is 20 for this graph[6].

The results indicate that the classification accuracy for mixed data-to-signature ratios is lower than for data sets with uniform data-to-signature ratios. Classifiers can cope better with mixed data-to-signature ratios for data sets with low data-to-signature ratios (such as 25:1), but are more severely affected for higher data-to-signature ratios (such as 100:1 and 200:1). This is good news given that worms typically have data-to-signatures less than 25:1, as shown in Table 5.2.

A likely explanation is offered by the way n-gram counts are extracted

[6]Standard deviations in the range 1 to 20 were tested; the trend was that accuracy decreases with increasing standard deviation, as expected.

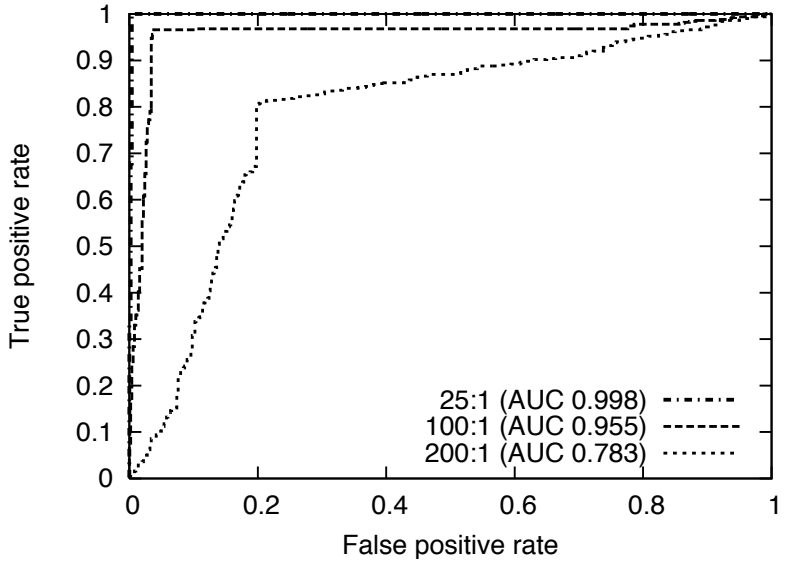


Figure 5.17: ROC curve showing the trade-off between true and false positive rates for various data-to-signature ratios.

from flows. By definition, flows with a higher data-to-signature ratio contain more application specific data and executable payload, which the machine learning classifier treats as noise. Mixed data-to-signature ratios may therefore dilute the signal (signature relative to data), making it harder for the classifier to spot the signature.

5.3.8 False alarms

The graph in Figure 5.17 shows how the AUC decreases for increasing data-to-signature ratios, as expected. Note that the AUC should never be less than 0.5 since the diagonal line joining co-ordinates $(0, 0)$ and $(0, 1)$ signals random guessing by the classifier. The confidence values were obtained by calculating the distance to the hyperplane [78].

5.4 Summary

This chapter tested the suitability of Support Vector Machines for detecting worm mutations, with encouraging results:

- **High accuracy for typical worm data-to-signature-ratios**. Worms typically have data-to-signature ratios below 25:1, and for these ratios the classifier consistently achieved very high accuracy.

- **Choice of the linear kernel as the optimal kernel**. Although all three kernels (linear, RBF, and string) demonstrated similar accuracies, the linear kernel and RBF kernels clearly outperformed the string kernel on training and classification time. The tie between the linear and RBF kernels was ultimately decided in favour of the linear kernel due to its simplicity.

- **Choice of bi-gram feature vectors**. Bi-gram feature vectors yielded the best accuracy-performance trade-off, clearly outperforming uni-grams on accuracy while closely matching the accuracy of tri-grams; at the same time bi-grams occupy significantly less memory than tri-grams.

- **Choice of unnormalised feature vectors**. Normalising the frequency counts improves accuracy only marginally but at the cost of significantly increasing training and prediction times.

- **Resilience to signature distortions**. SVMs demonstrated resilience to split and jumbled signature mutations. SVMs also demonstrated limited resilience to signature corruption, with accuracy hardly affected until 10% corruption.

- **Resilience to mixed data-to-signature ratios**. Mixed data-to-signature ratios reduce the accuracy only to minor degree for small data-to-signature ratios.

- **False alarm rates**. Receiver operator characteristics graphs confirm that the false alarm rates for low data-to-signature ratios is lower than for high data-to-signature ratios.

The following chapter compares these results with two alternative machine learning techniques: Gaussian Processes, and K-nearest neighbours.

Chapter 6

Alternative machine learning methods

The experiments in the previous chapter demonstrated that Support Vector Machines can successfully detect worm mutations. Nevertheless, Support Vector Machines exhibit some weaknesses, and this chapter explores whether alternative machine learning techniques can match, or surpass, the performance of Support Vector Machines.

One such weakness of Support Vector Machines, already alluded to in Section 3.3.1, is that they do not return any measure of confidence with their prediction; the only way to obtain such a confidence measure is with ad-hoc workarounds such as calculating the distance to the hyperplane. Without a confidence measure, the worm detector has no chance to second-guess or override the prediction in uncertain cases.

Another weakness is that Support Vector Machines are natively binary classifiers, but in practice it is often necessary to distinguish between more than two classes. So far this dissertation was content with a binary classifier that is trained with a single worm and can predict whether a flow carries mutations of that specific worm. To extend the classifier to support multiple worms, each worm would have to be assigned its own class, something Support Vector Machines do not natively support. Workarounds have been proposed, such as cascading multiple binary classifiers in *one versus rest* [56]

fashion. For a comparison of methods to add multi-class classification to Support Vector Machines, see [100].

In light of these weaknesses, are there better alternatives to Support Vector Machines to detect worm mutations? This chapter looks at two candidates introduced in Chapter 3 and compares their performance to Support Vector Machines:

- *Gaussian Processes* [16] return confidence values with their predictions, and this chapter investigates whether these values could help reduce false alarm rates relative to Support Vector Machines without sacrificing speed. Gaussian Processes also natively support multiple classes.
- *K-nearest neighbour* [17] is a simple and intuitive algorithm, and this chapter explores whether K-nearest neighbours can match the level of accuracy of Support Vector Machines. Like Gaussian Processes, K-nearest neighbours natively supports multiple classes, though like Support Vector Machines it does not return confidence values.

Table 6.1 summarises the advantages and disadvantages of the three machine learning techniques investigated in this chapter.

6.1 Experiment design

This section describes in-turn the configurations and experimental setup of Support Vector Machines, Gaussian Processes and K-nearest neighbours methods compared in this chapter's experiments.

6.1.1 Support Vector Machines

The experiments will use the optimal Support Vector Machine configuration established in the previous chapter, namely a linear kernel using unnormalised bi-gram feature vectors.

Method	Advantages	Disadvantages
SVM	• previous chapter demonstrated suitable to detect worm mutations • highly optimised implementations available	• no confidence value • multiple classes not natively supported
GP	• returns confidence value • multi-class support	• implementations not optimised
KNN	• simple and intuitive algorithm • multi-class support • supports incremental learning	• no confidence value • slow for large training data sizes

Table 6.1: Advantages and disadvantages of Support Vector Machines, Gaussian Processes, and K-nearest neighbours.

6.1.2 Gaussian Processes

As mentioned at the outset of this chapter, a shortcoming of Support Vector Machines is that they do not return a confidence value with their predictions, while Gaussian Processes return such confidence values as part of their results automatically. Since confidence values could be used to reduce the number of false alarms, the aim is to show how effective Gaussian Processes are in detecting worm mutations.

Like Support Vector Machines, Gaussian Processes are kernel machines, with the kernel measuring the similarity between two points. Section 5.3.2 of the previous chapter showed that the linear kernel was the optimal kernel for Support Vector Machines, implying that the data (the worm flows) are linearly separable. Based on this finding, the experiments in this chapter will employ the linear kernel for Gaussian Processes as well. Besides the kernel there are no further configuration parameters for Gaussian Processes.

For the experiments in this work the Matlab implementation for the Gaussian Processes classifier in [101] was ported to C so that it can be plugged into the architecture described in Section 4.2. The implementation stays true to the original Matlab implementation by using matrices and associated op-

erations; the C port uses the Gnu Scientific Library (GSL) [102] to handle matrix operations. The implementation was validated by comparing results for a wide range of input test data with the original Matlab implementation.

6.1.3 K-nearest neighbours

K-nearest neighbours is arguably the most straightforward supervised machine learning algorithm. Like Support Vector Machines and Gaussian Processes, K-nearest neighbours is a kernel machine. For the same reasons as Gaussian Processes – the previous chapter established the data as linearly separable and the linear kernel as the optimal kernel – this chapter's experiments use a linear kernel for K-nearest neighbours as well. This means all three classifiers will be equipped with a linear kernel.

This leaves the K value, the number of nearest neighbours participating in the majority vote, for which the experiments first need to find an optimal value before comparing K-nearest neighbours to Support Vector Machines and Gaussian Processes. As mentioned in Section 3.4, increasing K tends to reduce the impact of noise, but simultaneously blurs the boundaries between classes.

The implementation loads the training data into a GSL matrix [102], for compatibility reasons with the more complex classifiers that heavily depend on matrices. For the same reason, every entry of the test data is loaded into a GSL vector [102] one at a time. The remaining implementation follows straight forwardly from the algorithm:

1. Compute the measure of similarity (in this case the Euclidean norm) between the current test datum and every training data entry.

2. Sort these distances to find the K smallest distances.

3. Look up the class for each of these K training data entries and perform a majority vote.

6.1.4 Combining classifiers

What happens to the accuracy if the three machine learning classifiers are combined? While several methods of combining different classifiers have been proposed [103, 104, 105], this chapter's experiments take the simplest approach, which casts a majority vote of the individual classifiers outputs.

6.2 Analysis

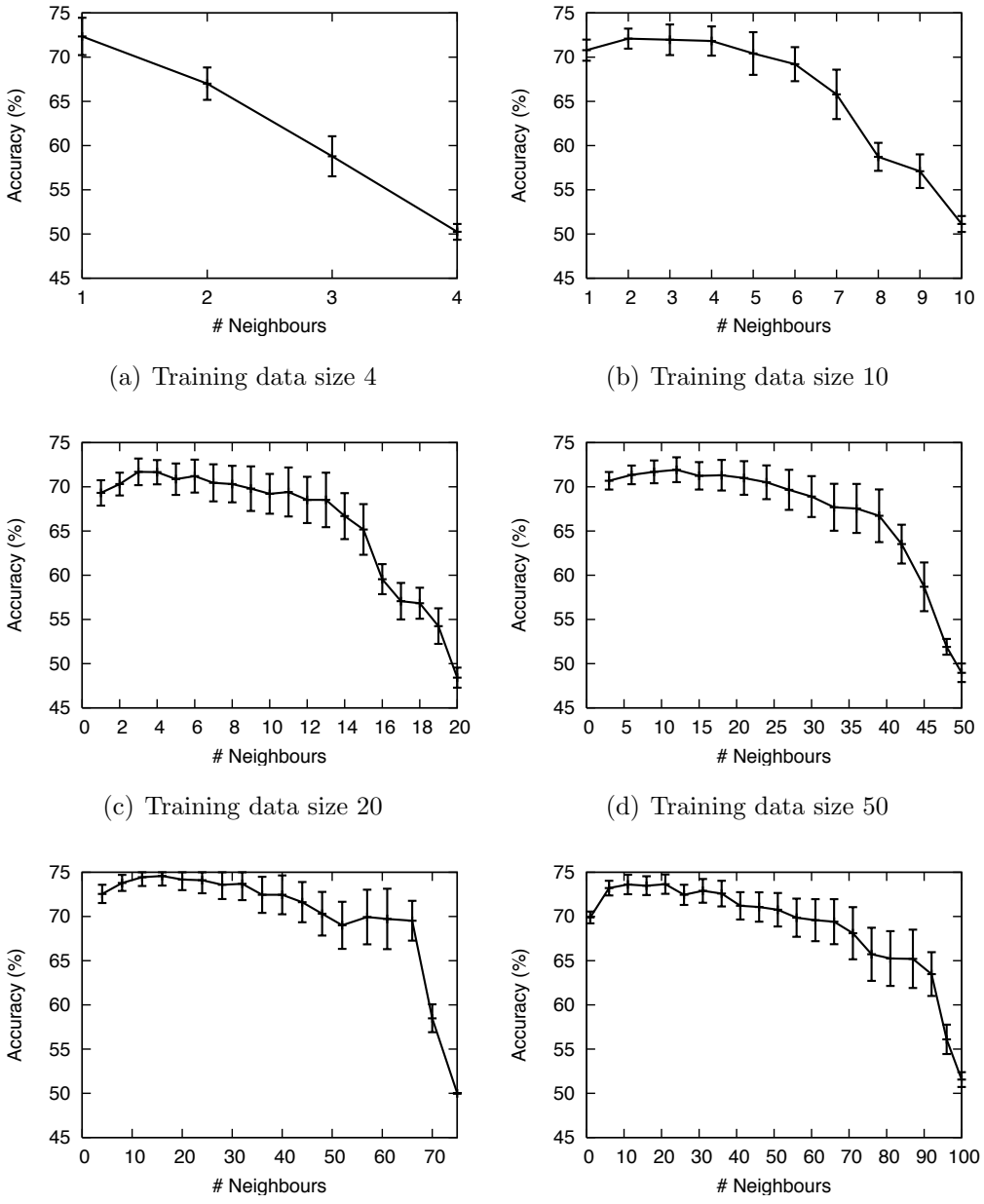
This section analyses the results of comparing Support Vector Machines with two alternative machine learning techniques: Gaussian Processes (GPs) and K-nearest neighbours (KNNs). Initially, this section first establishes the optimal K-nearest neighbours configuration to use in this comparison.

6.2.1 Optimal number of neighbours in KNN

The expected result is that accuracy rises with the number of neighbours, K, up to a certain threshold. Beyond this threshold, the vote will include relatively distant data points that will distort the vote and lead to wrong classifications. The aim, then, is to find the optimal number of neighbours just before that threshold is exceeded. Furthermore, the prediction time may rise with the number of neighbours, and thus another aim of this section is to determine whether there is an accuracy-speed trade-off for increasing K.

The graphs in Figure 6.1 show how the classification accuracy is affected by the number of neighbours for various training data sizes, with the number of neighbours varying from 1 to the corresponding training data size. The results show that there is no optimal number K across the various training data sizes, implying that for best results K must be tuned to the training data size. Table 6.2 lists the tuned K values, extracted from the results in Figure 6.1, with which K-nearest neighbours will be configured for the further experiments in this chapter.

The graph in Figure 6.2 shows how prediction time varies with the number of neighbours. The results show that, prediction time is unaffected by K.



(a) Training data size 4

(b) Training data size 10

(c) Training data size 20

(d) Training data size 50

(e) Training data size 75

(f) Training data size 100

Figure 6.1: Impact of the number of neighbours on the accuracy of KNN for various training data sizes. The data-to-signature ratio was 25:1.

Training data size	**Number of Neighbours (K)**
4	1
10	2
20	3
50	11
75	11
100	11

Table 6.2: Tuned K values extracted from Figure 6.1, with which K-nearest neighbours will be configured hence forth.

This is because K is only used in the algorithm once the training data items have been compared to the test data and sorted by distance to the test data, respectively taking $O(n)$ and $O(nlogn)$, where n is the training data size. Only then are the K nearest neighbours looked up, which is a constant operation.

6.2.2 Comparison of machine learning methods

Equipped with the optimal configuration for K-nearest neighbours, this section compares the classifiers in terms of accuracy, confidence value, prediction and training time, as well as training data size and false alarm rates.

6.2.2.1 Accuracy

Figure 6.3 compares the Gaussian Processes, K-nearest neighbours, and Support Vector Machines accuracy. The graph confirms that Gaussian Processes exhibit a similar accuracy to Support Vector Machines, being only slightly less accurate than the SVMs. For data-to-signature ratios below 100:1 it is approximately 2% less accurate. The results are similar enough, however, to consider GPs as a serious alternative to SVMs to detect worm mutations.

The results further show that the K-nearest neighbours algorithm performs considerably worse than both the Gaussian Processes and the Support

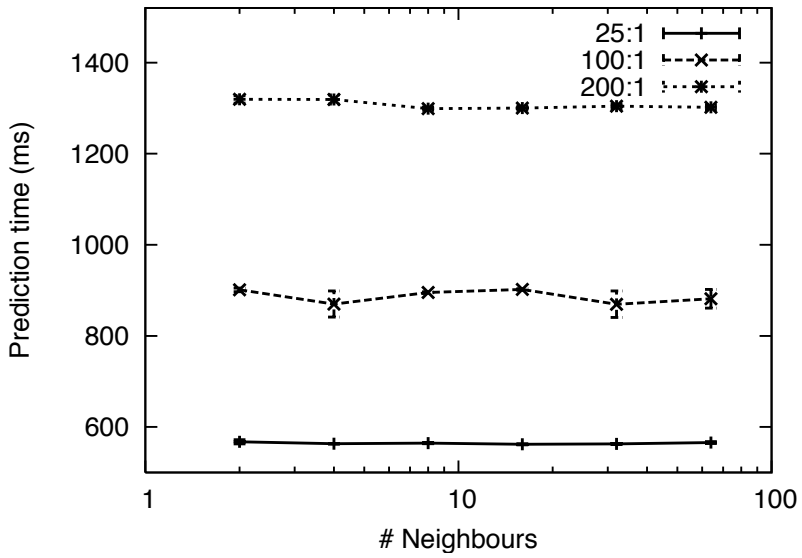


Figure 6.2: Impact of the number of neighbours on the prediction time for various data-to-signature ratios.

Vector Machines. For data-to-signature ratios below 100:1 the KNN classifier is approximately 20% to 25% worse than the other two classifiers.

6.2.2.2 Training data size

Section 5.3.4 showed that for Support Vector Machines, classification accuracy is affected by the training data size. The graph in Figure 6.4 shows that Gaussian Processes are similarly affected by the number of training data entries, with the prediction accuracy rising steadily with increasing training data size.

Although the K-nearest neighbours algorithm does not train a classifier as such (it only keeps a database of raw training data), it could still be affected by the training data size, albeit to a lesser extent. The results suggest, that unlike Support Vector Machines and Gaussian Processes, the KNN's classification accuracy does not increase steadily with training data size. Rather, prediction accuracy hovers at $71-75\%$, reflecting that the value of K is well-tuned to the training data size.

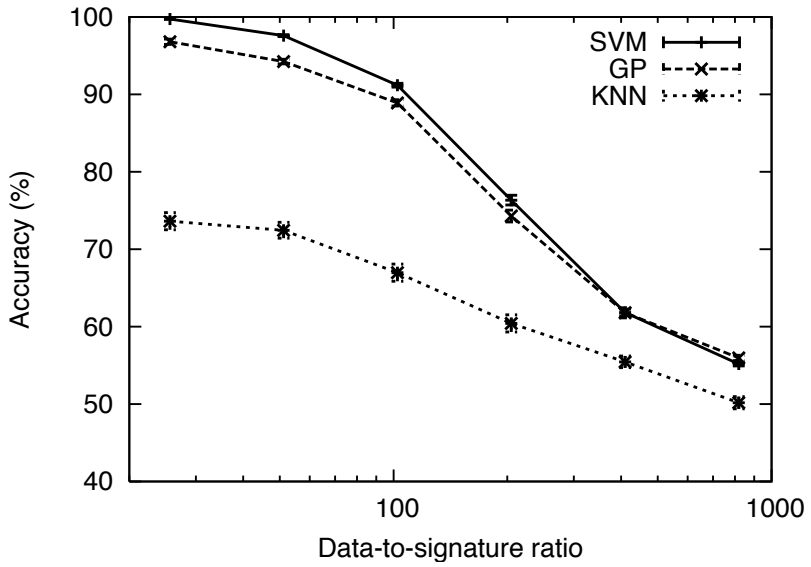


Figure 6.3: Comparing the accuracy of SVMs, GPs, and KNN. Training data size is 100.

6.2.2.3 Predictive likelihood

Since the previous section established that Gaussian Processes yield similar accuracy to Support Vector Machines, this section investigates whether the confidence value returned by the Gaussian Processes can help the decision process to further improve the results, and possibly give Gaussian Processes the edge over Support Vector Machines.

In Gaussian Processes the confidence measure is given by the *predictive likelihood*, which is the average of the logarithms of all predictive posteriors. A predictive likelihood of zero signals perfect confidence in the result. The actual values for predictive likelihood – that is, which values indicate high or low confidence – are problem specific and will be established in this section.

The graph in Figure 6.5 shows that the predictive likelihood decreases as the data-to-signature ratio increases. This is in step with the accuracies shown in the previous section implying that as the classification accuracy decreases, so does the predictive likelihood. Overlaying the GP's accuracies from Figure 6.3 with the predictive likelihoods from Figure 6.5 reveals that for a training data size of 100 a predictive likelihood between -0.655 and

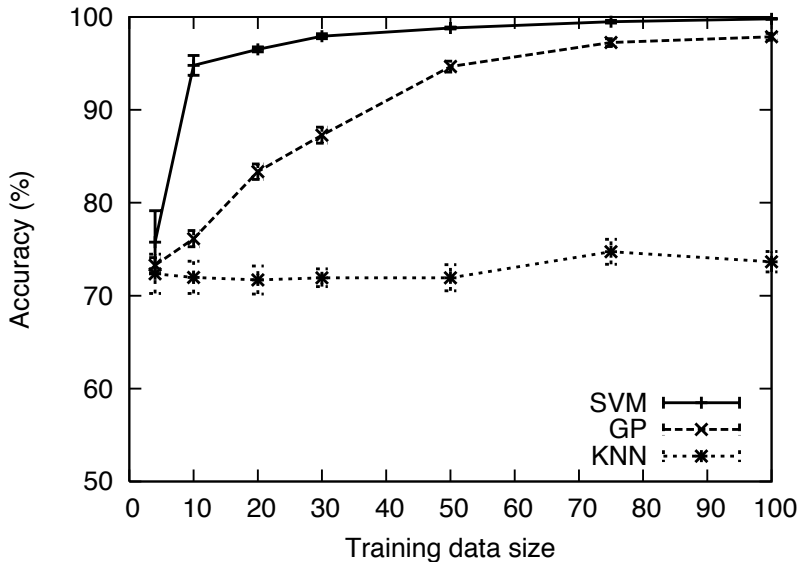


Figure 6.4: The impact of training data size on the prediction accuracy for Support Vector Machines, Gaussian Processes, and K-nearest neighbour. The data-to-signature ratio used is 25:1. For KNN the number of neighbours is tuned to the training data size as described in Section 6.2.1

-0.68 signals high confidence (above 90% accuracy).

The graph in Figure 6.6 shows the predictive likelihood of Gaussian Processes at various training data sizes. As expected, the prediction confidence also increases with training data size, in step with prediction accuracy. Overlaying the GP's accuracies from Figure 6.4 with the predictive likelihoods from Figure 6.5 suggests that for a data-to-signature ratio of 25:1 a predictive likelihood between -0.655 and -0.672 signals high confidence.

The results in this section confirm the predictive likelihood offers a practical guide as to when the classifier's decision can be overruled.

6.2.2.4 Mixed data-to-signature ratios

Section 5.3.7 in the previous chapter showed that Support Vector Machines can cope with mixed data-to-signature ratios for data sets with low mean data-to-signature ratios (such as 25:1) at minimal loss of accuracy, but are more severely affected for higher data-to-signature ratios (such as 100:1 and

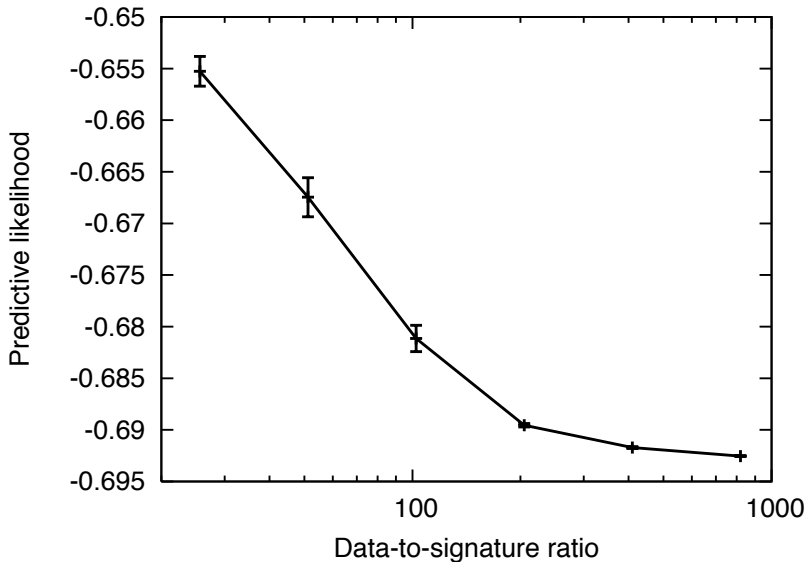


Figure 6.5: GPs predictive likelihood (confidence value) at various data-to-signature ratios. Training data size is 100.

200:1).

The graph in Figure 6.7 compares Support Vector Machines to Gaussian Processes and K-nearest neighbours at classifying data sets with fixed and mixed data-to-signature ratios. The fixed data-to-signature ratio is 25:1 while the mixed data-to-signature ratio is normally distributed with mean 25:1; standard deviation of 20 was selected as in Section 5.3.7.

The results show that the accuracy obtained with Gaussian Processes using mixed data-to-signature ratios is approximately 5% less than with fixed data-to-signature ratios. By comparison, the gap in Support Vector Machines is approximately 1%. The results for K-nearest neighbours show a drop of approximately 13% in accuracy when the data-to-signature ratio is mixed. As in the previous sections, the accuracy of the KNN algorithm at detecting worm mutations of known worms remains significantly below that of Support Vector Machines and Gaussian Processes.

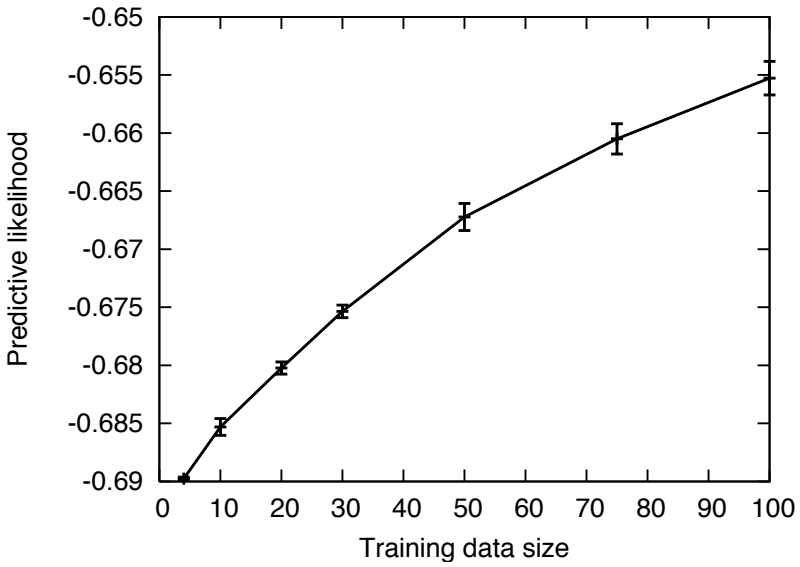


Figure 6.6: The impact of training data size on the prediction confidence for Gaussian Processes at a data-to-signature ratio of 25:1.

6.2.2.5 False alarm rates

As with the Support Vector Machines in Section 5.3.8, Receiver Operator Characteristics (ROC) graphs are used to investigate false alarm rates. Also as in Section 5.3.8, the confidence measure for Support Vector Machines is obtained by calculating the distance to the hyperplane. For K-nearest neighbours instance ranking (based on the algorithm in [106]) was used. Gaussian Processes as mentioned return the confidence measure natively.

Figure 6.8 compares the ROC curves for the Support Vector Machines, Gaussian Processes, and K-nearest neighbours classifiers. The results confirm that, as with accuracy, the KNN's area under the curve (AUC) is considerably less than both the Gaussian Processes or Support Vector Machines, meaning that KNN suffers from a considerably higher number of false alarms.

As with accuracy, Gaussian Processes rival Support Vector Machines, covering only a slightly less area under a curve.

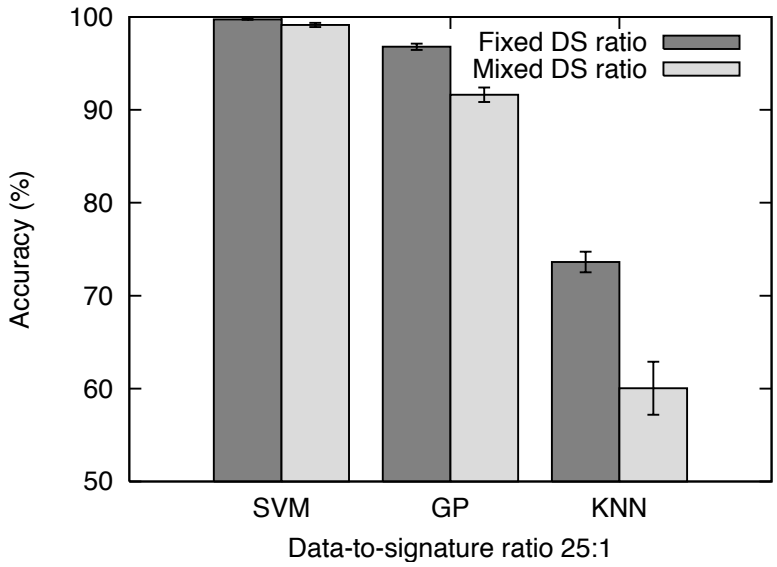


Figure 6.7: Prediction accuracy comparing SVMs, GPs, and KNN trained and tested with a single fixed data-to-signature ratio and a mixed data-to-signature ratio with comparable mean.

6.2.3 Combining classifiers

Figure 6.9 shows the prediction accuracy for classifier combinations using a majority vote to obtain a classification. Normally it would make sense to combine only an odd number of classifiers; for completeness, however, the graph also shows the prediction accuracy for all possible pair combinations, using a uniform random number generator to break ties where necessary.

From the results it can be seen that no combination beats the standalone Support Vector Machines and Gaussian Processes classifiers. A probable explanation is that Support Vector Machines and Gaussian Processes do not complement each other well and show similar predictions for the same test input data. K-nearest neighbours pulls down the average dramatically in every duo in which it participates.

However, in combination with Support Vector Machines and Gaussian Processes it appears to be outvoted most of the time, reducing the accuracy only marginally.

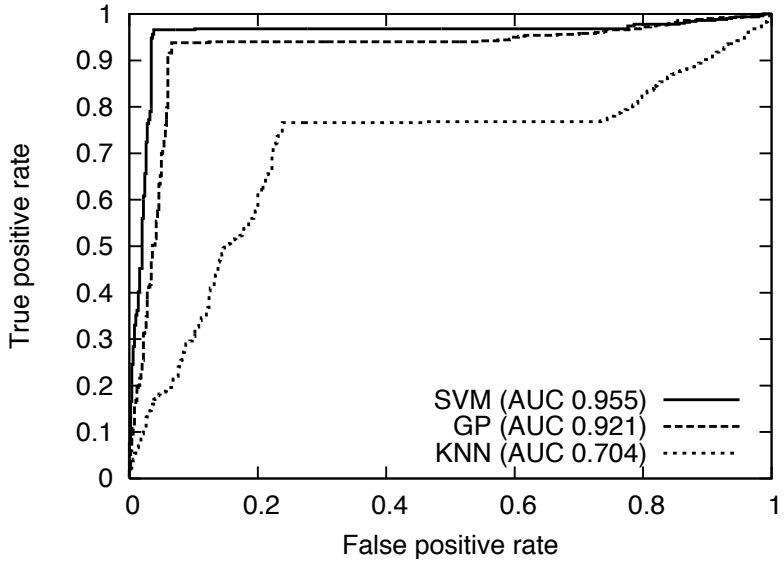


Figure 6.8: ROC curve showing the trade-off between true and false positive rates for SVMs, GPs, and KNN. Data-to-signature ratio is 100:1.

6.3 Summary

This chapter compared the performance of Support Vector Machines to that of Gaussian Processes and K-nearest neighbours for detecting worm mutations, with the following findings:

- **SVMs and GPs show similar accuracies**. Support Vector Machines and Gaussian Processes consistently showed similar accuracies, with SVMs being marginally more accurate than GPs. Both Support Vector Machines and Gaussian Processes achieve decisively higher accuracy than K-nearest neighbour.

- **Tuning the number of neighbours in KNN**. The number of neighbours in K-nearest neighbours needs to be tuned to the training data size since there is no silver bullet value that works well across all training data sizes.

- **GP confidence measures offer a practical guide**. The confidence measure returned by Gaussian Processes is low when accuracy is low,

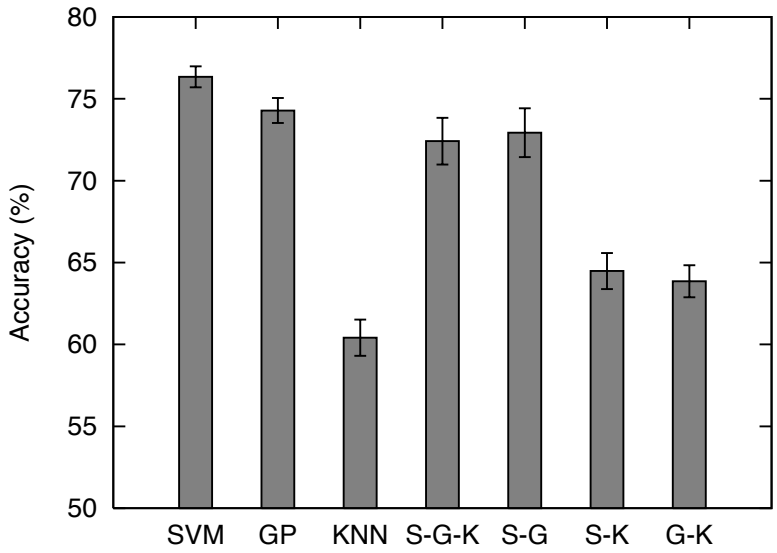


Figure 6.9: Prediction accuracy for various combinations of classifiers. Data-to-signature ratio is 200:1, since it best illustrates the trends of all data-to-signature ratios investigated.

rendering it a practical tool to further improve the accuracy of Gaussian Processes.

- **SVM and GP accuracy dependent on training data size**. Accuracy increases steadily with training data size for Support Vector Machines and Gaussian Processes, the rise being steeper for Support Vector Machines. K-nearest neighbours is tuned to training data size and accuracy hovers just above 70% for all training data sizes.

- **SVMs cope best with mixed data-to-signature ratios**. Support Vector Machines cope better with data sets of mixed data-to-signature ratios than Gaussian Processes. K-nearest neighbours, however, does not fare well with mixed data-to-signature ratios.

- **SVMs and GPs have similar false alarm rates**. Receiver operator characteristic graphs confirm that Gaussian Processes have only a marginally higher false alarm rate than Support Vector Machines, whereas K-nearest neighbours perform considerably worse.

- **Standalone SVM and GP classifiers work best**. No combination of classifiers (using majority vote) beats the standalone Support Vector Machines and Gaussian Processes classifiers.

The final verdict is that Support Vector Machines and Gaussian Processes were close rivals, with Support Vector Machines achieving a few percent points higher accuracy on average. K-nearest neighbours were constantly outperformed by Support Vector Machines and Gaussian Processes. The results also underline that for the techniques to work successfully, it is important that a sufficient amount of training data is available. The next chapter explores the possibility of automatically generating such training data.

Chapter 7

Generating worm mutations

The experiments in the previous two chapters showed that machine learning techniques successfully detect worm mutations. More precisely, the results showed that, given sufficient training data, the machine learning techniques generate classifiers able to detect worm mutations.

However, as mentioned in Section 5.1.1, these experiments used synthetic worm mutations for training data due to the lack of a wide spectrum of worm mutations in the wild. For a real-life deployment of the machine learning classifiers, obtaining a sufficiently large sample of training data poses a challenge. This chapter provides the missing link: a *worm mutation generator* that automatically generates worm mutations.

This chapter first describes two approaches taken by the worm mutation generator: *structurally* generating mutations by programmatically replacing the executable payload, and *randomly* generating mutations by intercepting malicious network flows and arbitrarily changing bytes in these flows. The chapter then compares which of the two approaches yields the higher quality training data in terms of classifier accuracy.

7.1 Structurally mutated worms

The concept of the structural worm mutation generator is to take a known exploit and programatically change the executable payload to generate worm

mutations. This approach is consistent with the worm model introduced in Section 4.1, which defined worm mutations as worms sharing the same exploit code, and reflects the approaches taken by past worm mutations such as Code Red I and Code Red II.

Three real exploits were selected from the *Metasploit framework* [25], a toolkit for exploit developers and security professionals, which provides a vast collection of well documented exploits. The exploits were chosen according to three key criteria. First, they had to be fairly straightforward to adapt and so lend themselves well for creating worm mutations easily. Second, they should blend into normal traffic in the sense that files such as those used for the executable payload do not appear anomalous. And third, to be as diverse from one another as possible in terms of communication protocol, payload types and payload sizes, and so ensure that any results apply to a wide range of worms.

The three worms created – named Distcc worm, Minishare worm, and WarFTP worm after the services they exploit – are described in detail in the following sections. These worms will also serve as the source worms for the randomly mutated worms described later in Section 7.2.

7.1.1 Distcc worm

Distcc [94] is a C/C++ compiler framework for distributing builds across several Linux machines on a network. An exploit [107] allowing arbitrary execution of shell commands is made available by Metasploit.

The worm infection works in four stages:

1. Launching the exploit, which opens the door to execute arbitrary shell commands on the target machine at the same user-level as the Distcc daemon.

2. Uploading the worm's executable payload in source code form. This is achieved seamlessly by piping the source code into a temporary file on the target host with the `cat` shell command.

3. Compiling and executing the uploaded source code. This uses the `gcc` (Gnu Compiler Collection) [108] command, which is present on most Linux installations, and since Distcc is essentially a `gcc` wrapper, can be safely assumed to be present on the victim.

4. Opening a backdoor `telnet` on port 4444 with a one line Perl command, allowing arbitrary shell commands to be executed.

The executable payload would usually also include spreading logic, such as generating a set of random IP addresses to probe next, and possibly some potentially malicious behaviour, such as corrupting the infected machine's hard-drive. For this chapter's experiments it suffices to infect the host and see if the payload was executed, and so the worms do not include any spreading logic or malicious payload.

7.1.2 Minishare worm

Minishare [95] is a trimmed-down web server for Microsoft Windows designed for file sharing via a browser interface. A buffer overflow exploit [109] allowing arbitrary Windows executables to be uploaded and executed is made available by Metasploit.

The Minishare worm works in three stages:

1. The attacking host makes the executable payload available for downloading by binding to a known local port. The worm will call back on this port once it has successfully infected the host.

2. The attacking host launches the exploit code against the target. If successful, the worm then calls back on the attacking host to download the executable payload.

3. The worm on the infected host launches the downloaded payload.

Unlike the Distcc worm, the Minishare worm does not download its executable payload in source form but as a binary executable. The download callback is a simple HTTP request, which the attacking host serves with Webrick [110], a minimalist web server that ships with Ruby. As with the Distcc

worm, the executable payload injected by the generator into the Minishare worm does not contain any spreading logic.

Since Minishare is primarily intended for file sharing, the uploading of binary data will not appear anomalous. That would not be the case if Minishare were intended for use as a regular web server where the dominant type of file transferred would be text-based web pages.

7.1.3 WarFTP worm

WarFTP [111] daemon is a simple File Transfer Protocol (FTP) server for Windows. A stack-based buffer overflow vulnerability [112] is available in Metasploit that allows arbitrary Windows executables to be uploaded and launched.

The WarFTP worm works similarly to the Minishare worm:

1. The attacking host makes the executable payload available for download by binding to a known local port.

2. The attacking host launches the exploit code against the target, and on success calls back to download the payload.

3. The worm on the infected host launches the downloaded payload.

The difference between the WarFTP worm and the Minishare worm goes beyond the use of different vulnerable services. The WarFTP worm was chosen because it can carry payloads considerably larger than both Minishare and Distcc without appearing anomalous. This allows the experiments to verify whether, as would be expected, the data-to-signature ratio plays a significant part in the quality of generated classifiers. The reason the WarFTP worm does not appear anomalous is that it exploits the FTP service, and as such it would not be uncommon to observe large file transfers in everyday traffic.

7.2 Randomly mutated worms

The basic idea of the random worm mutation generator is to (i) intercept a worm en route from an attacking host to a target host, (ii) generate a set of mutations by randomly mutating the worm's byte stream, and (iii) forward the generated mutations to the original target. The next sections discuss in detail the random mutation generator's requirements and its architecture.

7.2.1 Requirements

The random worm mutation generator must:

- **Intercept worms en route from one host to the next**. In order for the system to mutate intercepted worms, it must be able to capture them in their entirety. This involves reassembling worms that may have been split into several packets, as well as reordering packets that arrive out of order.

- **Work transparently**. The generator must forward all packets so that the attacker remains unaware of its existence, effectively acting as a transparent proxy.

- **Determine whether the target was successfully infected**. After the worm's mutations have been forwarded to the worm's original target host, the generator must determine whether the target was successfully infected. Without this check the mutation can not be labelled as malicious or benign, and thus can not serve as training data.

- **Store generated mutations**. The final step after determining whether a mutation is malicious or benign is to store it accordingly. This builds up the training data set of labelled mutations that later will be fed into the training stage.

7.2.2 Architecture

The high-level architecture consists of three machines connected in series, as shown in Figure 7.1. Machine `A` is the attacker from which the original worm is launched and machine `C` is the worm's target. Machine `B` sits between machines `A` and `C` and acts as a transparent proxy that mutates worms.

A typical cycle in this architecture looks as follows:

1. Machine `A` launches the original worm at machine `C`
2. Machine `B` intercepts the worm by reassembling the packets in the flow
3. Machine `B` randomly mutates the intercepted worm
4. Machine `B` forwards the mutation to machine `C`
5. Machine `B` queries machine `C` whether the attack succeeded
6. Machine `B` stores the mutation if the attack succeeded

The worm model introduced in Section 4.1 defined a worm mutation as a worm that exploits the same vulnerability as the original worm, but executes a different payload upon successful infiltration. Thus an attack of random mutation will only succeed if the mutation alters the executable payload without affecting either the application specific data (which is required to successfully transport the worm) or the exploit (which is required to infiltrate the target). Only successful attacks will be labelled as malicious.

The random mutation generator is an attractive extension over the structural mutation generator since it offers a fully automated method to generating worm mutations. While the structural mutation requires knowledge about the worm's structure – in particular the location of its payload – the random mutation generator treats worms as black boxes and requires no such prior knowledge. The experiments in this chapter will tell whether the random mutations can match their structural counterparts in serving as high quality training data.

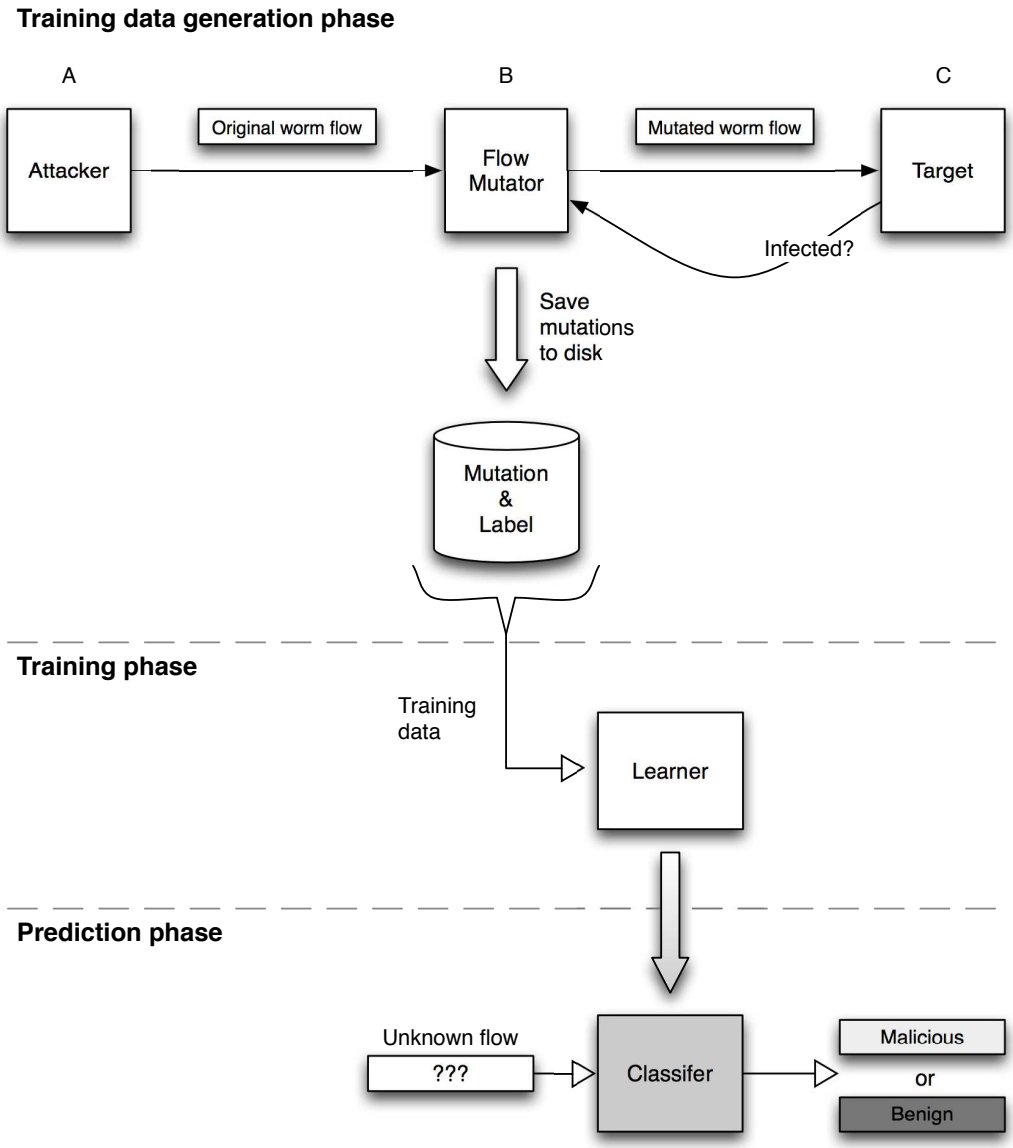


Figure 7.1: How the worm mutation generator fits into the overall worm detection architecture. The worm mutation adds a pre-learning phase that generates the training data (top) for the learning phase (middle).

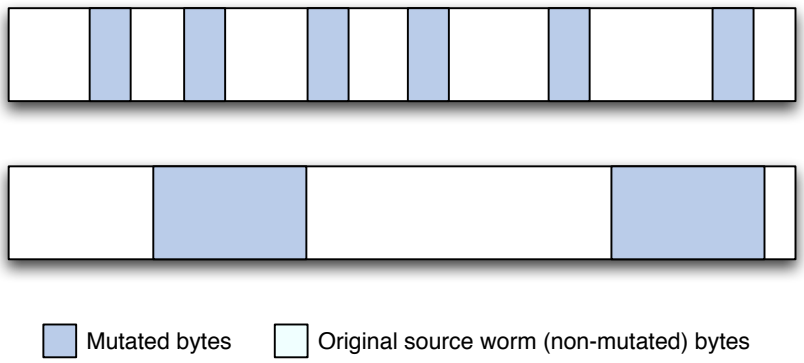


Figure 7.2: **Gauging the mutation degree**. A high mutation probability and small chunk size (top) yields fine-grained mutations. A large chunk size and low mutation probability (bottom) produces more coarse-grained mutations.

7.3 Experiment design

The experiments will test the quality of the worms generated by the worm mutation generator in two parts: first for the structurally mutated worms and second for the randomly mutated worms.

To challenge the classifier, the benign mutations closely resembled their malicious counterparts except that they were disarmed of the exploit code. In the case of the WarFTP worm, for example, the benign trace consists of an FTP file transfer of the executable payload, minus the exploit code. Similarly, for Minishare the executable payload was uploaded to the file sharing repository without launching it. For Distcc the benign traces remotely compile the source code but do not execute it.

For the random mutation generator, the experiments will test the impact of the mutation degree on classifier accuracy. Two parameters control the mutation degree: the mutation probability and the mutation chunk size (Figure 7.2). Thinking of the traffic as an array of bytes b, the mutation probability determines whether an intercepted byte b_i should be mutated. The mutation chunk size n then controls how many of the subsequent bytes $b_i, b_{i+1}, \ldots, b_{i+n-1}$ will be mutated. The experiments will investigate the impact of both the mutation probability and the mutation chunk size on clas-

sifier accuracy.

7.3.1 Isolated testing network

Due to the hazardous nature of worms the experiments were conducted in an isolated testing network. This network consists of three machines interconnected via a hub[1]: (i) a client machine from which attacks are launched, (ii) a server machine that runs the vulnerable software, and (iii) a machine hosting the forwarding proxy, which sits between the attacker and the target, intercepting traffic between the two. This setup derives directly from the architecture described in Section 7.2.2.

The machines used were AMD Athlon XP 2200 with 1 GB of physical memory. The target machine was a dual boot system running Ubuntu Linux 7.04 and Microsoft Windows XP Service Pack 1. The reason for using a dated Windows version was that it is known to contain many security holes, and as such offers an ideal breeding ground for the worm mutations. The other two machines ran Ubuntu Linux 7.04.

7.3.2 Implementation of a worm testing framework

In addition to the hardware setup described above, the experiments were driven by a custom-built distributed worm testing framework. This framework, implemented in Ruby, automates the execution of the experiments by providing a central interface from which to (i) reset the testbed, (ii) select an attack type, (iii) launch the attack, and (iv) verify whether the infection was successful.

The framework's distributed components map directly to the hardware setup in the previous section and therefore consist of an attacker, a target, and a forwarding proxy. These components run as agents on their hardware counterparts using the Distributed Ruby API [110], and are remotely coordinated by a central controller. The controller conducts the following event

[1]Note that a hub was used rather than an Ethernet switch since hubs forward traffic on all ports thereby allowing communications to be tapped.

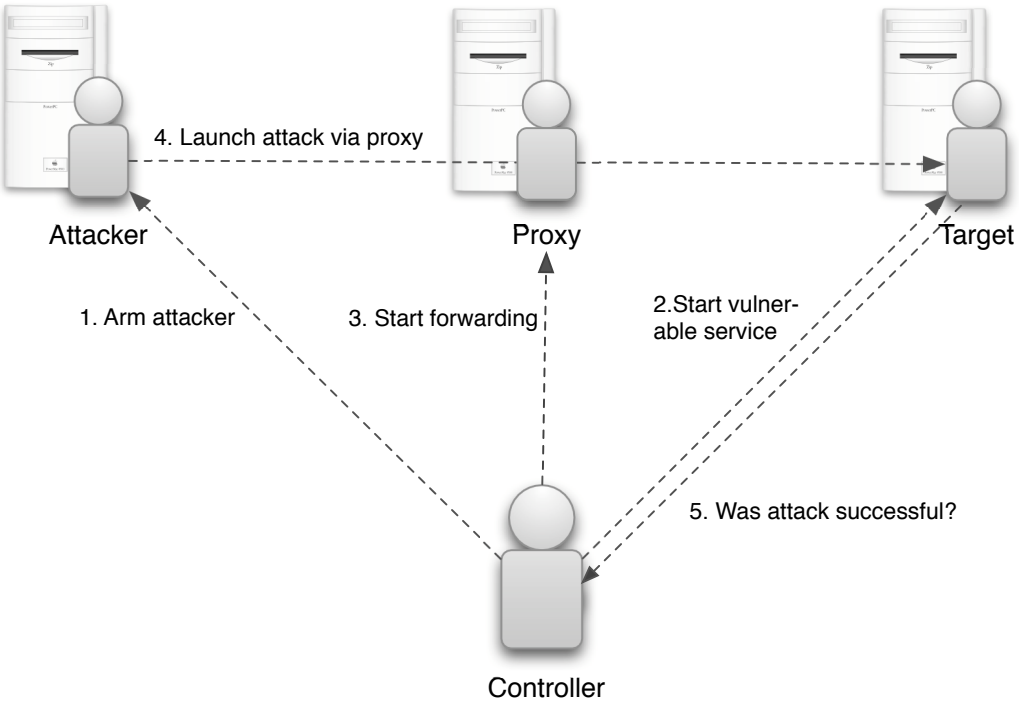


Figure 7.3: Worm testing framework. Agent interaction during an attack cycle.

sequence (Figure 7.3 shows the interaction of the agents for the first 5 steps):

1. Select a worm and arm the attacker with this worm
2. Launch the vulnerable service on the target
3. Start the forwarding proxy and recording the flow
4. Launch the attack
5. Query whether the target was successfully infected
6. Store the mutated flow under malicious or benign
7. Stop recording the flow and reset the target host

The central controller can sit on either of the three machines, or on a dedicated machine. For convenience, the experiments run the controller on the

forwarding proxy machine. Each worm attack is encapsulated in a dedicated class with the *command* design pattern [113].

For structurally mutated worms the proxy simply records the traffic flows and then transparently forwards them to the target. For randomly generated worms the proxy additionally randomly mutates bytes with a given probability as discussed in the outset of this Section.

The forwarding proxy uses sockets in a modified version of *port proxy* [114]. Sockets offer an attractive solution since they deal with the TCP flow's contents directly, freeing the mutation generator from having to reassemble flows[2]. A drawback of the sockets approach is that the ports under attack must be known in advance since they can only bind to fixed ports. While this is adequate for the controlled environment of this chapter's experiments, it poses a restriction for production use. A workaround is to accept connections from all ports, which could be implemented with an open source routing gateway package such as the *Click modular router* [115].

The querying of the target is implemented by having the worm emit its unique ID to a log file. For the Distcc worm this is easily implemented with an `echo` command in the shell command sequence that is part of the executable payload. For Minishare, which carries compiled C programs as its executable payload, writing to the log file is achieved by injecting `fprintf` statements into the programs' source code. WarFTP, which carries binary executables for which source code is not available, achieves this by wrapping the executables in another executable that emits an `fprintf` before launching the wrapped executable. The wrapper is a simple C program that embeds the executable as a hex string generated by the `xxd` Unix utility.

7.3.3 Structurally mutated worms

Key experiments from the previous two chapters are repeated using structurally mutated worms. In particular, the accuracy of Support Vector Machines, Gaussian Processes, and K-nearest neighbours at various training

[2]Flows would have to be reassembled manually if a packet capturing library such as *libpcap* [88] were used.

data sizes is compared for each of the worms.

A data set of 400 mutations was created for each worm. As in previous experiments, half of these were malicious and the other half benign. Various training data sizes were investigated by selecting an equal number of malicious and benign mutations as training data, and selecting 100 of the remaining mutations (again half malicious and half benign) as test data. Experiments were repeated 50 times to obtain averages and standard deviations.

For Distcc, mutations were created by substituting the source code file that is uploaded and executed with typical C tutorial programs. As such the Distcc worm mutations carry a small payload with an average trace size of 1.7KB at a standard deviation of 1.1. The exploit size is 0.3KB, resulting in an average data-to-signature ratio of 5.7:1

Structural mutations of the Minishare worm were created by substituting the executable payload with small binary executables. For this the set of C tutorial programs from the Distcc worm was compiled, resulting in an average trace size of 8.4KB and a standard deviation of 1.7. Given an exploit size of 2.2KB, this yields a low average data-signature-ratio of 3.8:1.

WarFTP structural mutations were created by substituting the executable payloads with exemplar Windows executables. The average trace size was 82.4KB with standard deviation of 55.2. With an exploit size of 1KB, WarFTP has the highest average data-to-signature ratio of the three worms with 82.4:1. The experiments will show how well the classifier can cope with such a high data-to-signature ratio.

7.3.4 Randomly mutated worms

These experiments compare classifiers trained with the structurally mutated worms from the previous section with classifiers trained with randomly generated worm mutations. To allow for a fair comparison, the training data for the latter consists solely of randomly generated mutations, while the test data consists of the same structurally mutated worms and benign data used in the previous section.

The experiments for the random mutation generator require a source

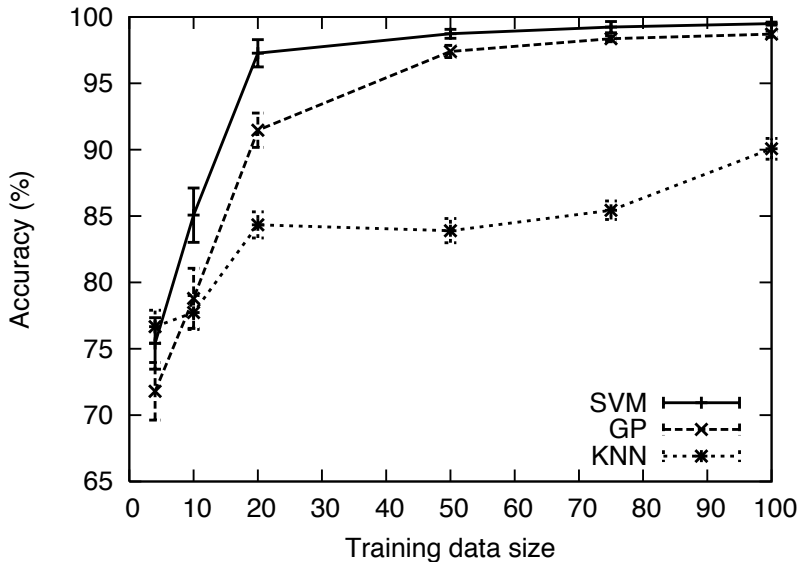


Figure 7.4: **Structurally mutated Distcc worm**. Accuracy vs. training data size for Support Vector Machines, Gaussian Processes, and K-nearest neighbours.

from which random mutations are generated. For each of the three worms the source worm was selected to be as close to the average size as possible.

7.4 Analysis

The analysis follows the structure of the experiment design. First the results of repeating key experiments from the previous two chapters on structurally mutated worm mutations are presented and discussed. The best performing machine learning technique is then used to test the random mutations and the two results are compared.

7.4.1 Effectiveness of structurally mutated worms

This section looks at the results for the Distcc, Minishare and WarFTP worm mutations in turn.

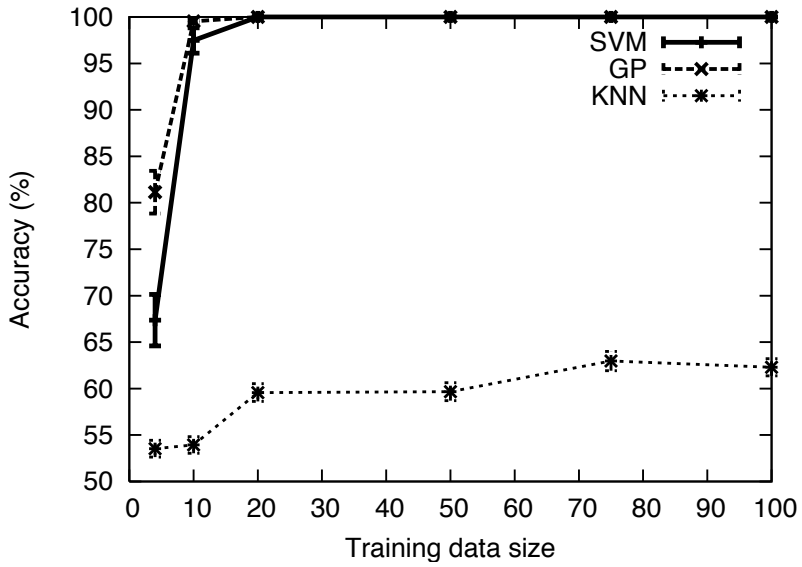


Figure 7.5: **Structurally mutated Minishare worm**. Accuracy vs. training data size for Support Vector Machines, Gaussian Processes, and K-nearest neighbours.

7.4.1.1 Distcc worm

Figure 7.4 compares the accuracy of Support Vector Machines, Gaussian Processes, and K-nearest neighbours on the Distcc worm. The graph shows that the accuracy increases for Support Vector Machines and Gaussian Processes up to a training data size of 50, after which the accuracy levels off. For training data sizes below 20, K-nearest neighbours lags not too far behind Support Vector Machines and Gaussian Processes, while showing considerably less accuracy for higher training data sizes.

As in the previous chapter, Support Vector Machines and Gaussian Processes achieve similar high accuracy for training data sizes above 50, peaking at 98% accuracy. For training data sizes below 50, Support Vector Machines perform noticeably better than Gaussian Processes.

7.4.1.2 Minishare worm

Figure 7.5 compares Support Vector Machines, Gaussian Processes, and K-nearest neighbours on structural Minishare mutations. As for the Distcc

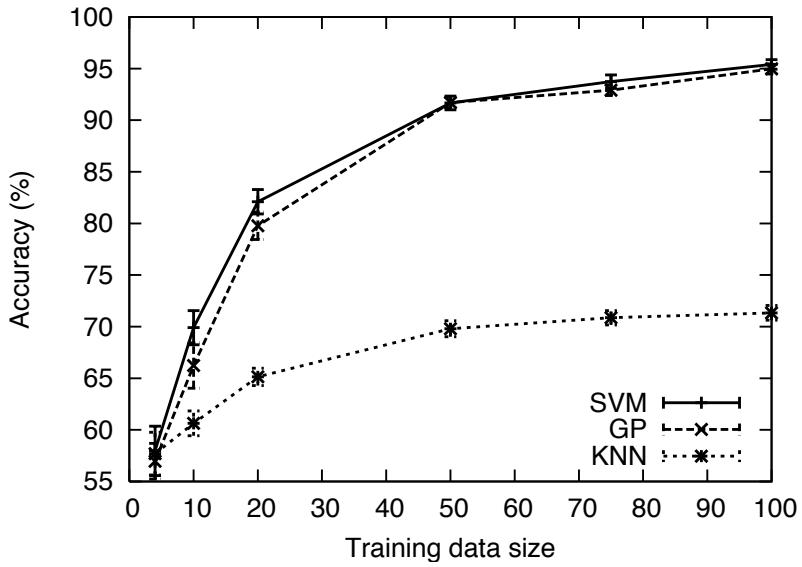


Figure 7.6: **Structurally mutated WarFTP worm**. Accuracy vs. training data size for Support Vector Machines, Gaussian Processes, and K-nearest neighbours.

mutations, classification accuracy increases with training data size for Support Vector Machines and Gaussian Processes.

The rise in accuracy is considerably steeper for Support Vector Machines and Gaussian Processes than with Distcc, climbing to near 100% accuracy for training data sizes as low as 10. The steep rise suggests that the training stage is able to detect a pattern sooner than for Distcc, the most likely explanation being the lower data-to-signature ratio.

As with Distcc, Support Vector Machines and Gaussian Processes perform similarly well except for very small training data sizes. K-nearest neighbours performs worse altogether, hitting a maximum of just over 60% accuracy.

7.4.1.3 WarFTP worm

Figure 7.6 compares Support Vector Machines, Gaussian Processes, and K-nearest neighbours for structural WarFTP mutations. Compared to the Distcc and Minishare results, the climb in accuracy for increasing training data size is more gradual, indicating that the larger data-to-signature ratio

makes it more difficult to extract a distinguishing pattern.

Of the three worms, WarFTP achieves the lowest maximum accuracy for Support Vector Machines and Gaussian Processes, peaking at around 95%. Again the most probable explanation is the high data-to-signature ratio. The accuracy for K-nearest neighbours is considerably worse, consistently tailing SVMs and GPs by around 25% from training data sizes larger than 50.

As with Distcc and Minishare, Support Vector Machines and Gaussian Processes perform comparably.

7.4.1.4 Discussion

In summary, the results of the structural worm mutation experiments showed that:

- As with synthetic worms, Support Vector Machines and Gaussian Processes show higher classification accuracy than K-nearest neighbours
- Classification accuracy depends on the data-to-signature ratio, with smaller data-to-signature ratios leading to higher accuracy

The results in this section have set the bar for the randomly mutated worms investigated in the next section. The results obtained for the three worms used in this section will now be compared directly to results obtained using training data sets generated by the random worm mutation generator.

7.4.2 Effectiveness of randomly mutated worms

Are the randomly mutated worm mutations suitable as training data for the machine learning classifiers? This section attempts to answer this question by comparing classifiers trained with randomly mutated worms to those trained with structurally mutated worms from the previous section. Based on the results of the previous section, the results in this section will base its comparison on Support Vector Machines.

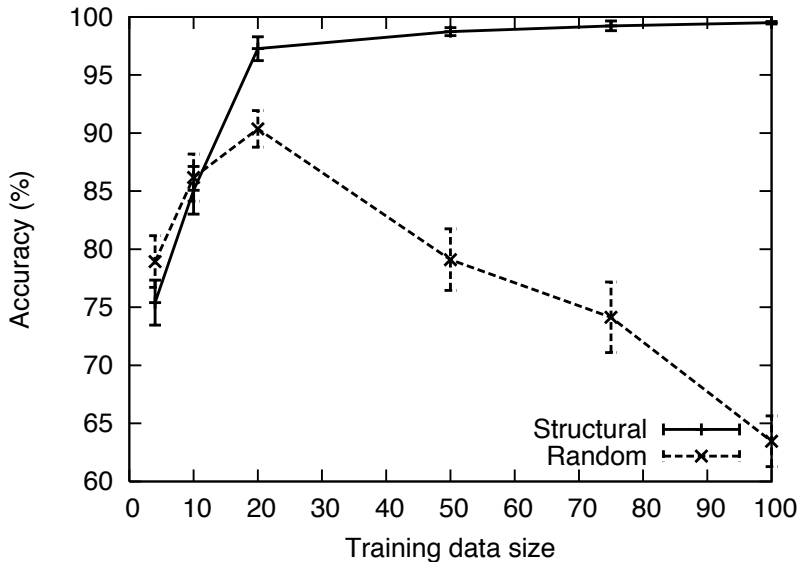


Figure 7.7: **Randomly mutated Distcc worm**. Prediction Accuracy vs training data size for classifiers trained of the randomly mutated Distcc worm. Mutation probability was 0.005 with a chunk size of 300 bytes.

7.4.2.1 Distcc worm

Figure 7.7 compares the prediction accuracy of the structural mutations with random mutations of the Distcc worm. The results are promising, and show that classifiers trained with random mutations match that of structural mutations up to training data sizes of 10, topping at 90% accuracy for training data size 20.

For training data sizes greater than 20 accuracy drops, most likely due to overfitting. Recall that the random mutations are generated from the same source worm, and for larger training data sizes the training phase could falsely pick up parts of the source code as being part of the exploit code, especially if those parts have not been mutated by the random byte mutator.

7.4.2.2 Minishare worm

The results of the Minishare worm show a different trend to Distcc, but are equally encouraging. Accuracy increases steadily for the random mutations

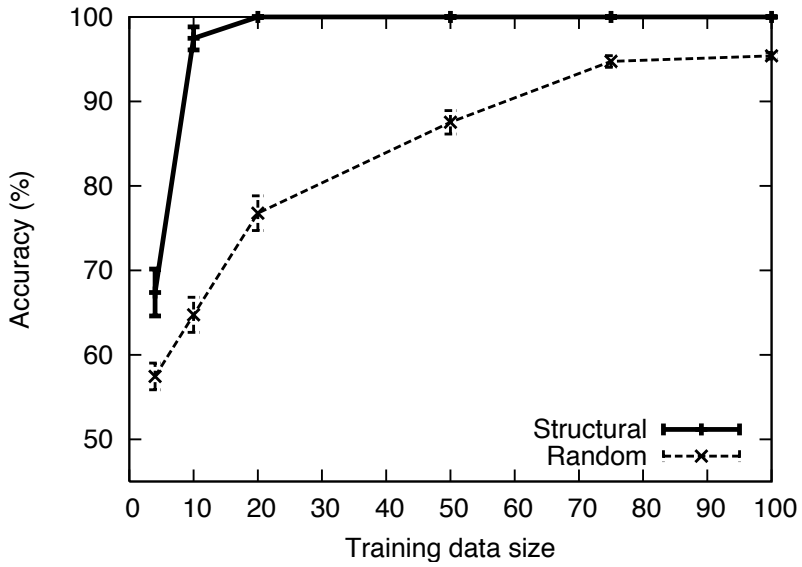


Figure 7.8: **Randomly mutated Minishare worm**. Prediction Accuracy vs training data size for classifiers trained with generated Minishare worm mutations. Mutation probability was 0.005 with a chunk size of 250 bytes.

with increasing training data size, but less gradually than the structural mutations, suggesting that the training stage requires a larger spectrum of random mutations than structural mutations to detect a common pattern.

The classifier reaches an encouraging maximum accuracy of 95% for random mutations, just 5% below that of structural mutations.

7.4.2.3 WarFTP worm

Figure 7.9 shows the results for the WarFTP worm. Out of the three worms investigated, this worm proved the most difficult for the classifiers, achieving the lowest accuracy for classifiers trained with random mutations of just 55% accuracy for smaller training sizes.

The most likely explanation for this poor result is both the high data-to-signature ratio (82.4:1) and the large standard deviation of the average payload size (standard deviation 55.2 for an average payload size of 82.4KB).

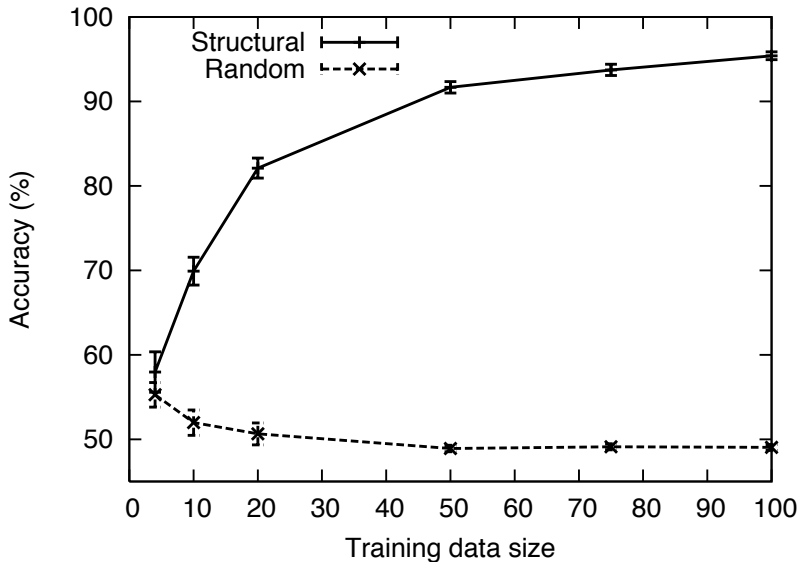


Figure 7.9: **Randomly mutated WarFTP worms**. Prediction Accuracy vs training data size for classifiers trained with generated WarFTP worm mutations. Mutation probability was 0.0001 with a chunk size of 1000 bytes.

7.4.2.4 Varying the mutation degree

The previous experiments kept the mutation degree fixed for each worm. Figure 7.10 analyses the impact of varying the number of bytes mutated, showing that accuracy increases as more bytes are mutated. Similarly Figure 7.11 analyses the impact of varying the mutation probability, showing that accuracy increases with higher mutation probability.

Combined, the results suggest that a higher mutation degree leads to a better trained classifier, as expected. As more bytes are mutated, the training phase is able to narrow down the common pattern onto the exploit code.

7.4.2.5 Discussion

The results of the randomly mutated worms were encouraging. Especially the Minishare experiments have shown that the random worm mutation generator is suitable to generate training data for machine learning classifiers, albeit with less overall accuracy than for structural mutations

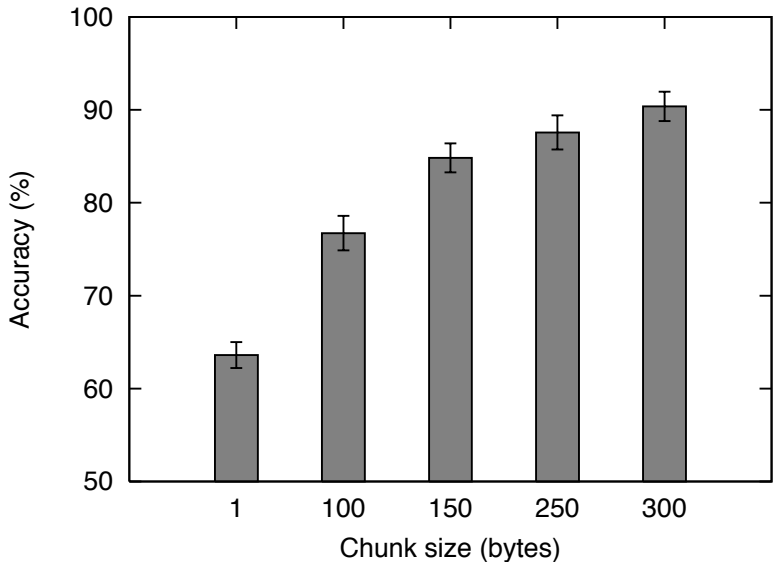


Figure 7.10: Impact of mutation chunk size (with a fixed probability of 0.005, and training data size of 20) on the prediction accuracy of randomly mutated Distcc worms.

Unusual about the Distcc results was that accuracy dropped for larger training sizes. Of the three worms, Distcc was the only one with a payload in source code form rather than as a binary executable, hinting that the classifier is prone to overfit purely-text based random mutations.

The results of the WarFTP worm were, however, less encouraging: with an accuracy of 55%, the classifier was little more than guessing the predictions. The poor results could be the result of the high data-to-signature ratio (82.4:1). By comparison, Distcc has a data-to-signature ratio of 5.7:1 but achieves 90% accuracy, while Minishare achieves up to 95% accuracy with a data-to-signature ratio of 3.8:1.

Gauging the mutation degree showed that accuracy improves as more bytes are mutated. However, the trade-off here is time. For example, with a mutation probability of 0.01 and a chunk size of 100 bytes, Distcc yielded a malicious mutation every 25 mutation cycles. Given that a random mutation cycle in the worm testing framework takes around 9 seconds to complete, the total time to build a training data size of, say, 100 malicious mutations is just over 6 hours.

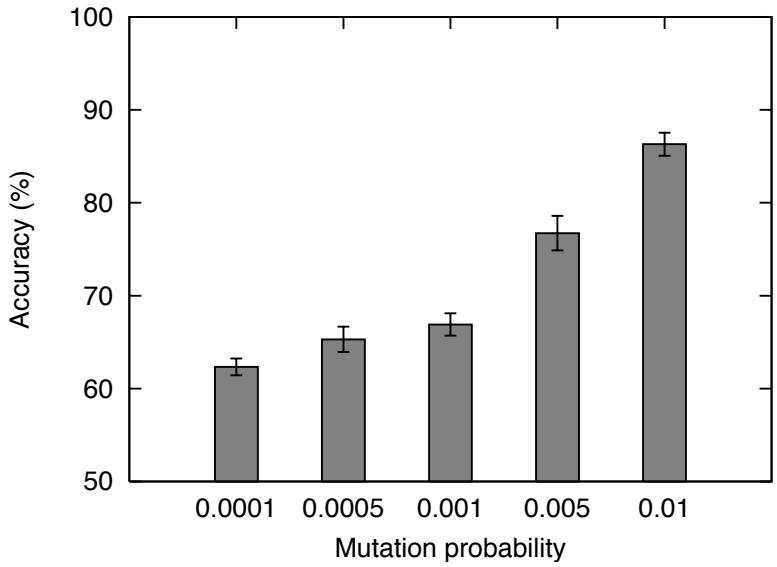


Figure 7.11: Impact of mutation probability (with a fixed chunk size of 100 bytes, and training data size of 20) on the prediction accuracy of randomly mutated Distcc worms.

7.5 Summary

This chapter flagged the need to automatically generate worm mutations to serve as training data for machine learning classifiers due to the lack of worm mutations in the wild. In particular, this chapter investigated:

- **How to generate training data**. Described the design, implementation and deployment of a worm mutation generator to handle this automation in two ways: structurally mutating worms by programatically replacing executable payloads, and randomly mutating worms by altering their byte streams arbitrarily.

- **Effectiveness of generated structural mutations**. Evaluated the structural mutation generator of the machine learning algorithms (Support Vector Machines, Gaussian Processes, and K-nearest neighbours) by repeating the key experiments of the previous chapter. The results showed that the quality of the generated structural mutations are sufficiently high to serve as training data.

- **Effectiveness of generated random mutations**. Compared random mutations to structural mutations using Support Vector Machines. The results showed that for Distcc and Minishare the quality of the random mutations is also sufficiently high to serve as training data, with the added bonus that they require no prior knowledge about the worm.

To conclude, this chapter has shown that using the worm mutation generator is a viable option for generating training data.

Chapter 8

Conclusion

8.1 Discussion

The worm problem affects us all. Today's news is filled with stories about spam flooding the Internet, phishing sites relieving people of their savings, Trojans harvesting sensitive information for identity theft, and worms cloaking as benign network traffic to sneak past firewalls and wreaking havoc.

Many network intrusion detection systems [43, 42, 14, 47, 53, 54] have been written to protect both private users and large organisations, but no matter how frequently the software is updated, it always seems to lag one step behind worm authors. This dissertation has the ambitious goal of not just levelling the playing field but to turn it upside down – to leverage the power of machine learning to automatically detect mutations of known worms.

Detecting worms has often been likened to finding a needle in a haystack. Machine learning [56, 57, 58], and in a larger context artificial intelligence, have long been hailed for their successes in recognising patterns [15, 66, 67], and it is this power that this dissertation attempts to harness.

Conceptually, by feeding machine learning techniques a training set, machine learning techniques are able to find distinguishing features in the training data that enables them to classify unknown data. The approach of this dissertation was to build an intrusion detection system that is fed both worm (malicious) traffic and normal (benign) traffic as training data, so that it can

automatically filter worm mutations out of everyday network traffic.

The past years have witnessed rapid growth and advances in the field of machine learning, and so one of the key decisions of this dissertation was which machine learning algorithm to use. Of all machine learning techniques, Support Vector Machines [15] have risen to prominence since the 1990's. The popularity of Support Vector Machines is not unfounded: they have successfully been applied to a large number of real-world pattern recognition problems including categorisation [66], image classification [67], and handwritten character recognition [15].

With such a good track record of detecting patterns, can Support Vector Machines find that proverbial needle in the haystack? Answering this question was the task of the first part of this dissertation, with encouraging results: the classifier reached 99% accuracy for the most common worm data-to-signature ratio of 25:1[1]. Even more encouraging was that Support Vector Machines are resilient to a fair degree of distortion, a popular technique with worm authors to cloak their worms. The results also suggest that Support Vector Machines could detect polymorphic worms [116], which bypass conventional intrusion detection systems by mutating themselves at each network hop.

The solid results for Support Vector Machines in detecting worms raised additional questions addressed by this dissertation: how do other machine learning techniques fare in detecting worm mutations? Can the results of Support Vector Machines be bettered by other popular machine learning techniques? The second part of this dissertation was devoted to answering these questions by comparing Support Vector Machines with Gaussian Processes [16] and K-nearest neighbours [17].

Gaussian Processes have a theoretical edge over Support Vector Machines in that they return a confidence measure with their classification, which a secondary classifier (for example a human) could leverage to potentially

[1]Interestingly, worm authors strive for a highly compact executable payload for faster spreading, effectively reducing the data-to-signature. The results show that the smaller the data-to-signature ratio, the higher the accuracy of the classifier. Conversely, a less skilled worm author who produces larger code might make it more difficult on the classifier.

reduce the number of false alarms. K-nearest neighbours, being a very simple and intuitive algorithm, offered another angle on how good the results of Support Vector Machines really were.

Support Vector Machines and Gaussian Processes were close rivals, with Support Vector Machines achieving a few percent points higher accuracy than Gaussian Processes, on average. While Support Vector Machines and Gaussian Processes proved to be a close call, K-nearest neighbours was outperformed by both Support Vector Machines and Gaussian Processes, and that despite the number of neighbours being tuned to the training data size.

Comparing Support Vector Machines, Gaussian Processes and, to a lesser extent also K-nearest neighbours, underlined that training data size is pivotal to the success of detecting worm mutations. This poses a challenge for deploying such classifiers in the real-world: the number of worm mutation traces in the wild is limited, potentially starving such a classifier of its key resource needed to detect worms.

The mission of the third and final part of this dissertation, then, was to find a way to secure the vital training data that fuels the classifiers. The result was a worm mutation generator that automatically generates worm mutations in two ways: (i) structurally mutating a source worm by replacing its executable payload, and, (ii) randomly mutating chunks of its trace. The results showed that the quality of the generated structural worm mutations were of high quality, suitable to serve as training data. The random worm mutations showed encouraging results as well, generating good quality training data for the two of the three worms investigated, with the important added bonus that they require no prior knowledge of the worm aside from being in possession of a known, lethal trace of the original worm.

This dissertation achieved its ambitious goal of putting network intrusion detection systems one step ahead of worm authors by automatically detecting worm mutations. It has laid the groundwork to yield a powerful weapon against the worm threat: a network intrusion detection system armed with a Support Vector Machine classifier, fed by the worm mutation generator with training data. This system should be powerful and accurate enough to au-

tomatically detect mutations of worms with little to no human intervention. This dissertation has laid a solid foundation, and the next chapter will look at how this solid foundation can be further extended.

8.2 Contributions

This work contributes to the defence against computer worms in the following ways:

- **Feasibility of Support Vector Machines in detecting worm mutations**. A thorough investigation into the feasibility of using machine learning based pattern recognition techniques to detect worm mutations was conducted. In particular, the optimal configuration of Support Vector Machines, including choice of kernel, n-gram extraction size, and training data sizes was investigated. The resilience to signature mutations and signature corruption, as well as the ability to cope with mixed data-to-signature ratios, was also investigated.

- **Comparison of alternative machine learning techniques in detecting worm mutations**. Support Vector Machines were compared to two alternative machine learning techniques, Gaussian Processes and K-nearest neighbours, with regards to how effectively they detect worm mutations. The results demonstrated that Support Vector Machines show slightly higher accuracy than Gaussian Processes, while K-nearest neighbours perform considerably worse. Gaussian Processes return a confidence value with their predictions, and this confidence was shown to offer a practical guide when the classifier is uncertain.

- **Automatically generating training data**. A worm mutation generator was developed in order to overcome the problem of limited availability of training data in the wild. The results confirmed the encouraging results from the previous chapters, as well as that training data for the machine leaning techniques can be generated using structural and, to a lesser degree, random mutation strategies.

Chapter 9

Future work

This dissertation has successfully shown that machine learning techniques are suitable for detecting worm mutations. Yet this is just the beginning. The machine learning techniques explored in this dissertation must now prove themselves in the real world, and this opens some interesting and exciting opportunities for future work.

This chapter looks into some of these opportunities: defending against sophisticated worm attack strategies, designing a custom kernel, alternative feature representation, real-time detection, online training, cascading classifiers, and scaling the infrastructure.

9.1 Sophisticated worm attack strategies

Worm authors often go to great lengths to maximise the potency and lethality of their worms. This has led to highly sophisticated attack strategies, which begs the question how robust the approach proposed in this work is to such strategies. This section describes some of these attack strategies and discusses how the work of this dissertation could be made more robust against such attacks.

9.1.1 Polymorphic blending attack

As mentioned in Section 2.1.2, *polymorphic worms* are worms that mutate themselves in such a way that their signature changes at every hop when traversing a network. Anomaly detection systems, by their nature of searching for anomalies rather than intrusions, lend themselves well to the detection of polymorphic worms.

An attack strategy that takes polymorphic worms to the next level are *polymorphic blending attacks* [117]. These attacks essentially consist of polymorphic worms that not only mutate from hop to hop, but also mutate in such a way that they blend in with normal traffic. Polymorphic blending attacks have been shown to successfully evade byte frequency based anomaly detection systems by morphing in such a way that their extracted features blend with normal traffic [117].

Since the approach advocated by this work uses n-gram based feature extraction, it is possible that it would be bypassed by polymorphic blending attacks. Consequently it would be interesting to investigate how the mechanism proposed in this work copes with polymorphic blending attacks, and to ascertain whether alternative feature representations such as those discussed in Section 9.3 improve the performance.

9.1.2 Zero day vulnerabilities

Vulnerabilities are typically discovered months before worms that exploit them are released. It is possible, however, that a worm author discovers a previously undisclosed vulnerability and releases a worm on the same day. Such worms are known as *zero day worms*. This dissertation focuses on detecting mutations of known worms and as such will not be capable of detecting worms on which it has not been trained, whether they are zero day worms or not.

However, classifiers in this dissertation are built using training data that contains both benign and malicious flows, and the classifiers learn to distinguish between the two classes. Using a confidence measure, such as provided by Gaussian Processes, it may be possible to flag new and unknown flows for

further investigation. This mechanism could be used to defend against zero day exploits.

9.1.3 Poisoning of benign traffic

A form of attack that has been shown to fool automatic signature generators, such as Polygraph [48], is to poison benign traffic by deliberately injecting well-crafted noise [118]. This poisoning can be accomplished by sending out an instance of benign traffic for every instance of malicious traffic. For example, for every attack of a vulnerable service, a worm could also perform a benign request to that service, which includes the exploit code, with the only difference that the exploit will not be executed.

By poisoning the benign traffic, it becomes increasingly difficult to distinguish between malicious and benign flows. The problem is that signature generators are at their most vulnerable in the training phase. The same is true for machine learning based classifiers: if given mislabelled training data they will not be able to build a good classifier.

This dissertation builds on the assumption that training data is correctly labelled and has not been deliberately poisoned. It would be interesting to investigate how susceptible machine learning is to poisoned traffic, and, if found to be highly susceptible, how the classifiers could be made more robust. A way to carry out this investigation is to feed the training stage with deliberately mislabelled data, for example labelling 5% of the benign data as malicious.

9.2 Designing a custom kernel

As mentioned in Section 3.2.3, a kernel conceptually measures the similarity between two data points in a given feature space. Kernels such as the RBF kernel deal with non-linearly separable data by implicitly mapping the data to a higher dimensional feature space. As such, the accuracy of the classifier depends on how suitable the feature space can represent the data at hand.

For cases where the feature space cannot readily represent the data, the

focus of recent work was to build a custom kernel [119, 71, 120]. Indeed, the string kernel [71] used in this dissertation is a custom kernel originally designed to categorise text documents, with good results that closely rivalled the linear and RBF kernels but did not match their classification times (Section 5.3.2.6).

Left for future work is the development of a custom, more discriminatory kernel based on the model of the structural and random byte mutation generators. This could take into account the degree of mutation, data-to-signature ratios and training data size, as well as cater natively for split, jumbled and corrupted signatures. A guide to constructing kernels is given in [56].

9.3 Alternative feature representation

As discussed in Section 4.4, this dissertation uses n-grams to extract features for the linear and RBF kernels. The findings of Section 5.3.3.1 suggest that bi-grams offer superior performance over tri-grams at only marginally less accuracy. This dissertation side-stepped values $n > 3$ because the number of features and hence memory requirements grow exponentially with n. However, higher values of n could improve the classifier's accuracy, and investigating how accuracy is affected by larger n will be insightful.

To be practically feasible, higher values of n must be approximated. One approximation technique are 2_v-grams suggested by Perdisci et al [52] as an improvement over PayL [50], two network intrusion detection systems outlined in Section 2.2.2. 2_v-grams approximate higher numbers of n by using a sliding window of size $v + 2$ when traversing the data. The idea is to extract information about higher n-grams ($n > 2$) by measuring only the occurrence frequency of byte pairs that are v bytes apart in the flow.

An advantage of 2_v-grams is that they work well with sophisticated attack strategies such as polymorphic blending attacks, where normal bi-grams perform poorly [52]. On the downside, however, 2_v-grams are still prone to the curse of dimensionality, although Perdisci et al [52] suggest to reduce the feature space by a feature clustering algorithm. Note that 2_v-grams are a

generalisation of n-grams; for $v = 0$ the technique degenerates to standard bi-grams.

Another alternative approximation technique is Content-based Payload Partitioning as used by Autograph [47] and Earlybird [14], both discussed in Section 2.2.2. This technique divides the payload into variable length blocks by computing Rabin fingerprints [121] over a sliding window, with the sliding window stopping when a predefined Rabin fingerprint is matched.

Rabin fingerprints can be efficiently computed over sliding windows [121], and have been shown to be robust to byte insertions, deletions and replacements [47] – common cloaking techniques by worm authors. However, Rabin fingerprints sometimes generate very short blocks, yielding unspecific features and consequently a high number of false alarms. At the other extreme, Rabin fingerprints may match the entire flow payload, leading to long signatures that are unsuitable for detecting worms. The workaround suggested in Autograph [47] is to impose maximum and minimum block lengths.

9.4 Real-time detection

In this dissertation, the machine learning classifiers enjoyed the comforts of a lab environment where the focus was accuracy rather than speed, but to compete in the real world they need to meet strict real-time requirements. This section outlines how real-time performance can be improved by (a) optimising flow reassembly, (b) partial flow classification, (c) dedicated hardware, and (d) reducing the memory footprint.

9.4.1 Optimising flow reassembly

The approach taken by this work is to detect worms in reassembled flows since this ensures that worms can be scanned in their entirety, even if they are divided into several, potentially reordered, packets. The machine learning classifier thus sits at the top of the network stack, freed from worrying about packets and seeing the network traffic simply as a continuous byte stream that it can match to its training data.

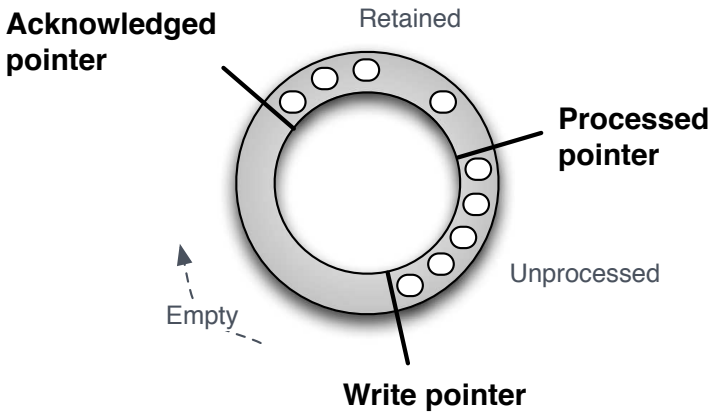


Figure 9.1: Real-time flow reassembly using a circular buffer. The circular buffer keeps track of two regions by way of three pointers: (i) a sequence of unprocessed packets, and (ii) packets that the flow analyser has requested to retain. Keeping the retained packets in the buffer as long as possible avoids spilling them to the heap.

However, this may break strict real-time requirements if deployed in high speed locations with immense traffic volumes, such as border gateways in university campuses and large corporations. There is significant overhead involved in copying data packets between physical memories, such as from the network card to main memory, as well as copying memory between kernel and user level. Typically, when dealing with line-rate classification, dedicated hardware enables working off the network interface's memory directly, rather than resorting to copying packets into main memory.

The impact of real-time flow reassembly and possible ways of achieving it on high speed network links was presented in [90]. Real-time flow reassembly in [90] is accomplished through the use of special-purpose network capturing interfaces and a circular buffer, several hundred megabytes large, as shown in Figure 9.1. The system fills the buffer and in parallel reassembles the flows directly from the buffer, thereby avoiding expensive memory copies.

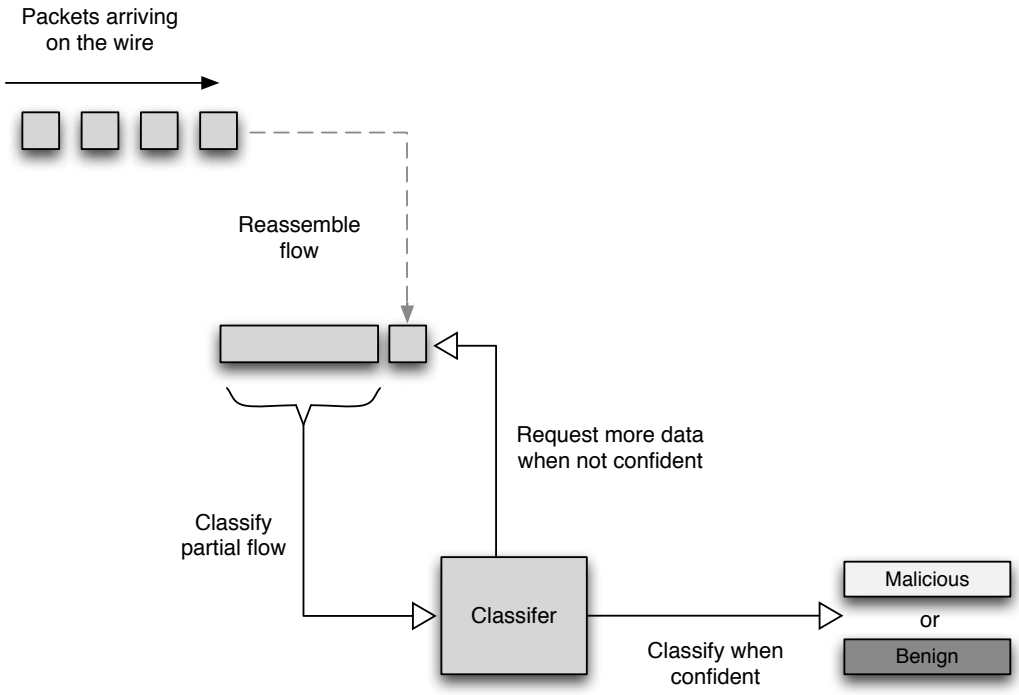


Figure 9.2: Partial flow classification

9.4.2 Partial flow classification

Besides possibly breaking real-time requirements, another disadvantage of scanning fully reassembled flows is that the worm's attack cycle must be completed, allowing the entire flow to be reassembled, before the machine learning classifiers begin their detection. By the time a malicious flow has been reassembled, passed to the classifier and categorised by the classifier, the worm may have already infiltrated the host.

A logical solution is to initiate classification when the first packets arrive on the wire, rather than waiting for the flow to be reassembled in its entirety. Figure 9.2 suggests how such on-the-fly classification could work in practice. The idea is to continuously classify flows while they are being reassembled and to use the confidence measure (as provided by Gaussian Processes) to determine whether the information seen so far is sufficient to label the flow as malicious or benign.

On-the-fly classification offers challenging opportunities for future work.

What is the optimal confidence threshold? What is the average number of packets required to make a sufficiently accurate classification? And, most importantly, can partial classification accuracy match that of full flow classification? Exploring such questions will be an interesting addition to this work.

9.4.3 Dedicated hardware

A further possibility worthy of investigation is to improve real-time performance by running the classifier (in whole or in part) on dedicated hardware. To be effective, the dedicated hardware should sit as close as possible to the network hardware, such as border routers and gateways, preferably even share the same physical memory so that network flows can be examined instantly.

The dedicated hardware could take the form of a custom-designed integrated circuit (IC) chip, which offers the maximum possible performance. On the downside, a classifier fully hardwired into a circuit will be hard to upgrade, tune and debug. Greater flexibility, albeit at a slight performance cost, could be gained by using configurable hardware such as that provided by NetFPGA [122].

9.4.4 Reducing the memory footprint

As mentioned in Section 4.4.2, even with bi-gram extraction a feature vector consists of $65,536$ entries, possibly straining the available memory resources for a large number of flows in a real-time environment. Future work could investigate reducing the memory footprint, for example using the following methods:

- **Feature selection**. This involves selectively removing features from the feature space and repeating the experiments to see if the accuracy can be maintained without the removed features. Libsvm ships with a tool that performs this selective feature removal [123].

- **Sparse matrices**. Instead of storing all entries of the feature vector in an array, only the non-zero items could be stored in a sparse representation. Libsvm offers the option to store feature vectors as dense or sparse matrices [98].

9.5 Online training

The machine learning classifiers used in this work separated training and classification not just by function but also by time: training is done offline and the results fed to the classifier for online use. As it stands, the only way to update the classifier is to train a new classifier offline in the background and hotswapping it in when ready.

Retraining the classifier online offers the advantage of being able to continuously improve classifiers while they are in operation, potentially improving their accuracy. The basic approach is to keep training classifiers incrementally with fresh training data, as for example shown in [124]. Incremental learning raises a number of intriguing questions for future work:

- Should the fresh training data be added when new training data is available, or should the training data be collected and bundled before updating the classifier? If bundled, what bundle size will yield the best speed-accuracy trade-off?

- Should fresh training data (bundled or not) be added as soon as it is available, or only when the system is under light load? Is it possible to add the training data without risking that worms slip past during the update?

- How does legacy data affect the classifier's speed and accuracy? Can legacy data be phased out as fresh training data is phased in?

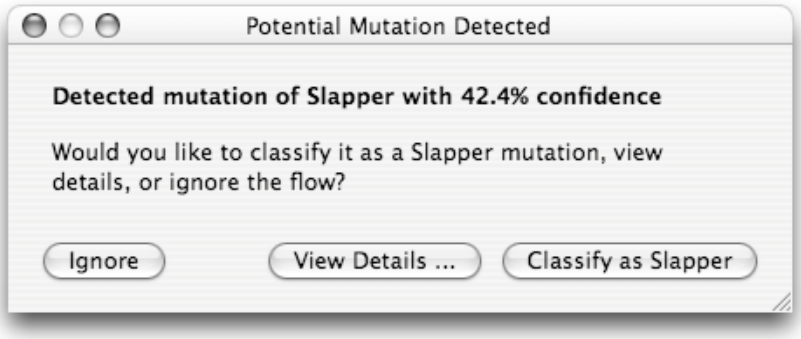


Figure 9.3: Manual analysis in uncertain cases.

9.6 Cascading classifiers

As a stand-alone classifier, Support Vector Machines have consistently performed more accurately than Gaussian Processes throughout this dissertation. At the same time, the confidence measure returned by Gaussian Processes has been shown to be a valuable tool since it offers another angle on the classification – it can be flagged for further processing when the confidence is low.

It would thus be interesting to investigate how accurately a chain of classifiers performs with Gaussian Processes as the primary classifier, handing off to a secondary classifier in cases of uncertainty. What are possible candidates for the secondary classifier? Two possible options are a) to involve a human operator to manually analyse the prediction, and b) to hand off further analysis to a refined machine learning classifier that is slower but more accurate.

9.6.1 Manual analysis

A possibility is to transfer control to a manual analysis stage where a human operator examines the flow. The manual analysis stage could be used both as a last resort if all else fails, and as a reinforcement mechanism for the machine learning algorithms in the early stages when the training data is still fresh.

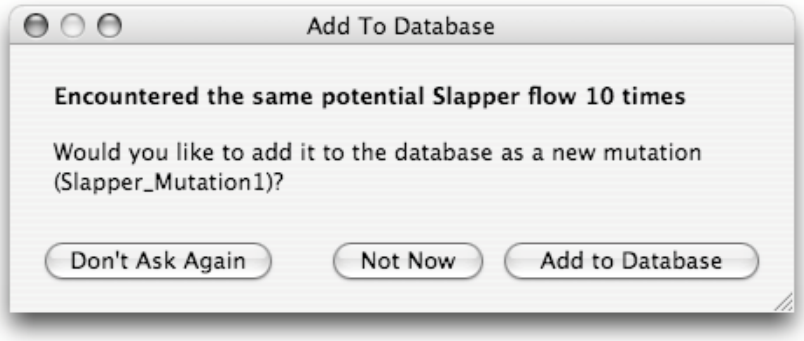


Figure 9.4: Adding new training data in manual analysis.

How might an interface for a human operator look? When the classifier is uncertain about it's classification, the human analysis stage is notified and prompts the operator as in Figure 9.3. The operator can then decide whether to accept the classification, ignore it or view details. The details of the worm mutation could be displayed in `tcpdump` format augmented with flow statistics, such as byte frequency distributions, for the human operator to make an informed decision.

The operator's input affects the accuracy of future predictions. If the operator rejects the classification, then the system will be less confident upon the next similar encounter. If the operator accepts the classification, the system will be more confident.

After asking the operator the same question a number of times (say, 10 times), the system can prompt the operator as in Figure 9.4. The operator can then decide whether the system should continue to prompt him, perform the action automatically, or add the discovered mutation to the database. Adding the signature to the database triggers the machine learning technique to retrain its classifiers.

9.6.2 Refined machine learning analysis

Rather than introducing a human operator into the loop, another option is to use an automated, more refined second machine learning classifier. This

classifier is slower – based on the understanding that only a fraction of the predictions by the Gaussian Processes will require further attention – but must be more accurate than the primary Gaussian Processes classifier.

Which machine learning classifier meets these requirements for the secondary classifier? One possibility is to equip the secondary classifier with a special-purpose kernel for detecting worm mutations, as outlined in Section 9.2. Another possibility is to use a different feature representation such as higher n-gram values, as discussed in Section 9.3.

9.7 Scalability

The approach taken by this dissertation is to build a binary classifier for a single worm that determines whether a flow is a mutation of that worm. This implies that a classifier must be built for each known worm. A question left for future work is whether this system scales well with a large number of worms, and if not, how the system can be made more scalable.

A potential option to increase scalability is to build multi-class classifiers that can distinguish between more than two classes (worms). Gaussian Processes, and K-nearest neighbours support multi-class classification natively, while Support Vector Machines have to resort to workarounds such as *one-versus-rest* [56]. It would be insightful to repeat the experiments in this dissertation on popular SVM extensions for multi-class classification, such as those compared in [100].

Another approach to scale the system is to build multiple binary classifiers (that is, one for each worm) and distribute them over to multiple machines, for example, one per classifier. There are a number of pitfalls when distributing to multiple machines, including but not restricted to unnecessary and time-consuming copying of data between machines. Considerable care must be taken that only necessary information flows between machines.

A lighter approach to distribute the system is to physically factor out the training to dedicated machines that build classifiers and upload these to the classification machines when ready.

9.8 Summary

This chapter explored opportunities for future work:

- **Defending against sophisticated worm attack strategies**. Worm authors often go to great lengths to ensure their worms evade detection, for example by building polymorphic worms that mutate themselves at each network hop, or by poisoning benign traffic with fake worms. Future work could investigate how robust the machine learning classifier is to such sophisticated worm attack strategies.

- **Designing a custom kernel**. A kernel measures the similarity between two data points in a given feature space, and as such the accuracy of the classifier depends on how well the feature space can represent the data at hand. Left for future work is the development of a custom, more discriminatory kernel based on the model of the mutation generators. Future work can investigate whether such a custom kernel can further improve the accuracy of this dissertation's classifier.

- **Alternative feature representation**. This work investigated unigrams, bi-grams and tri-grams to extract features for the linear and RBF kernels, but higher value n-grams could improve the classifier's accuracy further. Since the number of features grows exponentially with n, this requires approximation techniques such as 2_v-grams or Content-based Payload Partitioning.

- **Real-time detection**. Future work can improve real-time performance by a) optimising flow reassembly by directly accessing the network interface's physical memory, b) classifying worms with only partially reassembled flows, and c) placing the classifier on dedicated hardware such as custom-built integrated circuit boards or reconfigurable hardware.

- **Online training**. This work's machine learning classifiers perform training offline, meaning that when new training data arrives a new classifier has to be built and hotswapped with the old classifier. Future

work could investigate whether training the classifier online can further improve its accuracy. A possible approach is incremental learning, where the classifier is fed with new training data during live operation.

- **Cascading classifiers**. While the standalone Support Vector Machines has consistently performed more accurately than Gaussian Processes, it would still be interesting to see how effectively Gaussian Processes and its confidence value work in a chain of classifiers. Possible candidates for the secondary classifier handling uncertain cases are a human operator performing manual analysis, and a refined machine learning classifier that is slower but more accurate.

- **Scalability**. A task open for future work is to investigate the scalability of this work's implementation for large numbers of worms. One approach is to upgrade the binary classifier to a multi-class classifier. Another approach is to distribute the system, for example by separating the training and classification stages onto two machines. A step further is to distribute the system onto one machine per classifier.

Appendix A

Formulas of Support Vector Machines

In its basic, linear form, Support Vector Machines is a hyperplane that maximises the distance to the support vectors in a training data set. The distance of vector x to the hyperplane is given by:

$$u = \vec{w} \cdot \vec{x} - b \tag{A.1}$$

where w is the normal vector to the hyperplane. The separating hyperplane is defined by $u = 0$, and the nearest points line on the planes which $u = \pm 1$. Thus the margin m is

$$m = \frac{1}{||w||^2} \tag{A.2}$$

The problem of maximising the margin can be stated as an optimisation problem [63]

$$\min \frac{1}{2}||\vec{w}||^2 \text{ subject to } y_i(\vec{w} \cdot \vec{x} - b) \geq 1, \forall i \tag{A.3}$$

where x_i is the i^{th} training data item and $y_i \in \{1, -1\}$ its corresponding label. By applying a Lagrangian the optimisation problem can be converted into a Quadratic Programming problem where the objective function Ψ depends

solely on the Lagrange multiplier α:

$$\min_{\vec{\alpha}} \Psi(\vec{\alpha}) = \min_{\vec{\alpha}} \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j (\vec{x_i} \cdot \vec{x_j}) \alpha_i \alpha_j - \sum_{i=1}^{N} \alpha_i \tag{A.4}$$

where N is the number of training data items. The Lagrangian is constrained by the inequality

$$\alpha_i \geq 0, \forall i \tag{A.5}$$

and the linear equality

$$\sum_{i=1}^{N} y_i \alpha_i = 0 \tag{A.6}$$

The normal vector $\vec{w}$ and the value of b can be calculated once the the Lagrange multipliers have been found:

$$\vec{w} = \sum_{i=1}^{N} y_i \alpha_i \vec{x_i}, \quad b = \vec{w} \cdot \vec{x_k} - y_k \text{ for some } \alpha_k > 0 \tag{A.7}$$

For non-linearly separable data sets there will be no separating hyperplane, yielding an infinite solution in the above formula. Cortes and Vapnik [15] modified the optimization problem in Equation A.3 to allow (but penalize) cases when no correct margin can be reached:

$$\min_{\vec{w},b,\vec{\xi}} \frac{1}{2} ||\vec{w}||^2 + C \sum_{i=1}^{N} \xi_i \quad \text{subject to } y_i(\vec{w} \cdot \vec{x} - b) \geq 1 - \xi_i, \forall i \tag{A.8}$$

where ξ_i are slack variables that take into account margin errors, and the C parameter is the softness of the margin – it trades off margin width with the number of margin errors. Transforming the optimisation problem into dual form changes the constraint in Equation A.4 into a box constraint:

$$0 \leq \alpha_i \leq C, \forall i \tag{A.9}$$

Appendix B

Receiver Operator Characteristics Graphs

Receiver operator characteristics (ROC) [96] curve analysis presents a way of quantifying the trade-off between the detection rate (true positives) and the false alarm rate (false positives). ROC curves have their roots in signal detection and medical decision-making, and have recently become a popular way of analysing machine learning classifiers. This chapter presents an overview of ROC curve analysis, following [125].

B.1 ROC graphs

ROC graphs are a two-dimensional depiction of the accuracy of a signal detector plotting the true positive (y-axis) rate against the false positive rate (x-axis) respectively as shown in Figure B.1. The true positive rate is calculated by dividing the number of true positives by the total number of positives, and the false positive rate is calculated by dividing the number of false positives by the total number of negatives.

These two rates change in relation to another. That is, when the true positive rate is high, the false positive rate will be low, and vice versa. Naturally this means that these two rates can be equal somewhere in the middle.

The basic idea of ROC graphs is to provide a visualisation of the trade-off

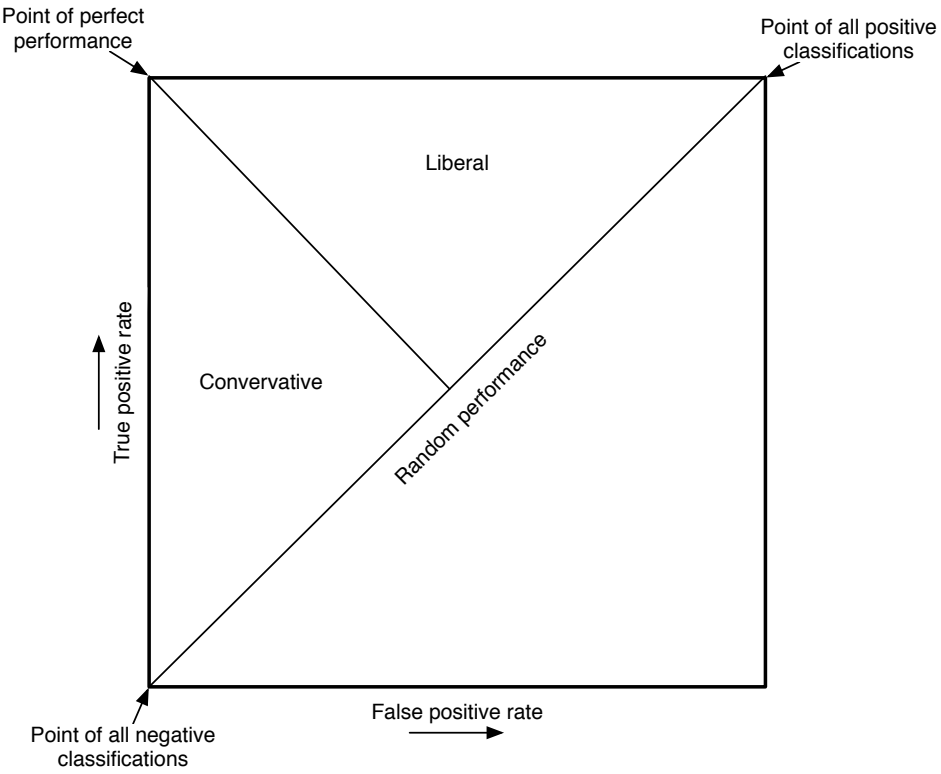
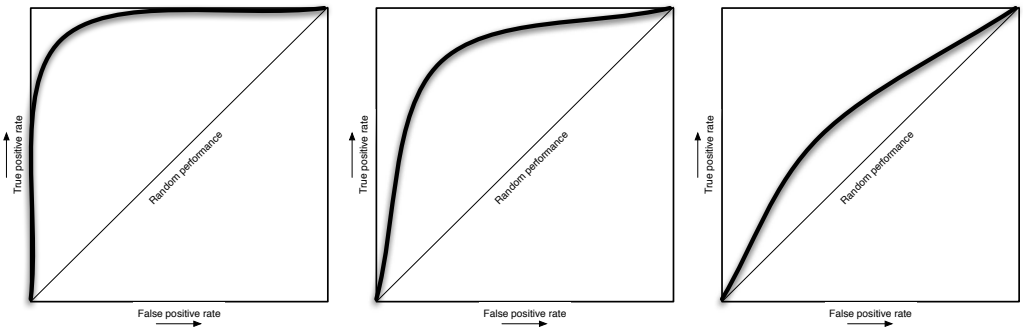


Figure B.1: Receiver operator characteristics (ROC) graph space showing (i) the line of random performance, (ii) liberal and conservative regions, (iii) the point of perfect performance, and (iv) the points of all positive and all negative classifications.

between the true and false positive rates, and hence an understanding of the accuracy of a classifier.

There are several general areas of interest in a ROC graph, as illustrated in Figure B.1. First, the diagonal dividing line that connects the bottom left to the top right corner represents random performance (guessing). Points that lie above this line are better than random and points below this line are worse than random. In theory, for classifiers there should be no points below this dividing line because classifiers that perform worse than random could be mirrored into the top half by simply inverting their outputs.

The top half of this diagonal line can be further split in half with a perpendicular line. The left half of this new division shows conservative (higher true positive than false positive rates) classifiers, whereas the right



(a) Almost perfect performance (b) Good performance (c) Poor performance

Figure B.2: Receiver operator characteristics curves for various performances.

half shows liberal (lower true positive than false positive rates) classifiers.

Classifiers in the conservative region make fewer false positive decisions, with the extreme point being the bottom left corner which indicates a classifier that classifies all instances as negative. This means, on the one hand, that it will not produce any false positives, but on the other hand, will not produce true positives either.

In the liberal region classifiers exhibit a good true positive rate but commit a significant amount of false positives. Again, with the extreme case being the top right corner, which represents a classifier that marks everything as true.

Finally, the point in the top left corner represents a perfect classifier. This classifier has a 100% true positive rate and a 0% false positive rate. This point can also be used as a reference point whereby other points on the ROC graph can be ranked by their distance to it.

B.2 ROC curves

The true advantage of ROC graphs does not come from single point interpretation, but from the ability to characterise a classifier's performance model as a curve. Figures B.2(a), B.2(b), and B.2(c), show the curves for an almost

perfect classifiers, a good classifier, and a poor classifier respectively.

To aid the interpretation of ROC curves, it is important to understand how they are constructed. Obtaining a single point in the ROC space from a classifier is straightforward by simply calculating the true and false positive rates for a classification. But how can a single run like this be transformed into a curve?

Curves are obtained from classifiers that attach a probability or ranking to each prediction. For each possible probability[1] or ranking a point is plotted in the ROC space. These points joined together form the ROC curve. As with the ROC graph itself, the left half of the curve represents the classifier's performance under high (conservative) decision thresholds and the left represents the classifiers performance under low (liberal) decision thresholds.

Curves can also be obtained for classifiers that do not yield a probability or ranking with each prediction. One method is to obtain a confidence value in an ad-hoc manner, such as calculating the distance to a SVM's separating hyperplane. Another way of obtaining an estimate is to sort the test set's individual classifications by their confidence values and then iterating over these values, computing true and false positive rates for all classifications up to and including the current value.

B.3 Area under curve

A single metric that can be obtained from these curves is the total area under the curve (AUC) [97]. Classifiers with larger areas perform better, on average, than classifiers with lower areas. Nevertheless, it is still possible for classifiers with lower AUC's to perform better than classifiers with higher AUC's for some regions of the graph.

[1]The default decision or probability threshold of classifiers is typically 0.5.

Glossary

Anomaly detection In intrusion detection, anomaly detection systems are equipped with a model of *normal* traffic. The idea is to detect intrusions by searching for traffic that does not correspond with this model.

Botnet A network of infected hosts that can be remote controlled to perform potentially malicious tasks.

Data-to-signature ratio The amount of data (padding) relative to the size of the signature.

Decision surface see separating hyperplane.

Distributions (mathematics): generalisation of functions and probability distributions

False negative Erroneously classifying something as negative, for example, erroneously classifying malicious data as benign.

False positive Erroneously classifying something as positive, for example, erroneously classifying benign data as malicious.

Flow see TCP/IP flow.

Gaussian Processes A machine learning technique that yields similar results to Support Vector Machines, with the addition of returning a confidence value as part of its result.

Hyperplane See separating hyperplane.

Intrusion detection systems Intrusion detection systems are special purpose monitoring systems that attempt to identify break-ins.

K-nearest Neighbours A simple machine learning technique that surprisingly often performs well.

Kernel Conceptually, kernels equip machine learning techniques such as Support Vector Machines, with the ability to map non-linearly separable data points into different dimensions where they are linearly separable.

Linear kernel The standard SVM kernel that tries to find a dividing hyperplane by calculating the dot product on pattern vectors in the original feature space.

Machine learning A sub-discipline of artificial intelligence and involves developing algorithms that allow computers to learn.

Misuse detection In intrusion detection, misuse detection systems are equipped with models of known intrusions (or signatures). These models, known as signatures, are used to identify intrusions by looking for matches in the network traffic.

N-gram extraction A feature extraction technique counts occurrences of all character combinations of size n, typically by linearly scanning the data.

Network intrusion detection systems Network intrusion detection systems, are specially designed to focus on network related intrusions. These systems are typically deployed at network gateways in organizations, allowing them to act as filters for any incoming traffic.

Perceptron A type of artificial neural network that builds linear classifiers.

Predictive likelihood The confidence measure in Gaussian Processes, which is the sum (or in some cases the average) logarithms of all predictive posteriors.

Radial basis function (RBF) kernel A kernel that applies a Gaussian function to the pattern vectors, implicitly taking them to a higher space.

Separating hyperplane A line (2-dimensions), or hyperplane (higher dimensions) that divides the training data into disjoint groups. Equipped with a separating hyperplane, a classifier can then label a given test data point based on its position relative to the hyperplane.

Signature Effectively a fingerprint that can be used to uniquely identify intrusions, such as worms. In its simplest form, it consists of a string of characters (or bytes).

String kernel A kernel, originally developed for categorising text documents, that maps the input strings into the feature space generated by all sub-sequences of a given size, where it applies the inner product.

Supervised learning Learning with labelled training data.

Support Vector Machines A machine learning technique known to perform particularly well at pattern recognition tasks such as text categorisation and hand-written digit recognition.

Support vectors In Support Vector Machines, support vectors are the key data points close to the hyperplane that, if removed, would change the location of the hyperplane.

TCP/IP flow Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) packet streams, where a stream is identified by the source Internet Protocol (IP) address, source port number, destination IP address, and destination port number.

True negative Correctly classifying something as negative, for example, classifying benign data as benign.

True positive Correctly classifying something as positive, for example, classifying malicious data as malicious.

Unsupervised learning Learning with unlabelled training data, such as clustering.

Worm Worms are malicious programs that spread themselves to hosts on the Internet by exploiting vulnerabilities in software applications.

Bibliography

[1] J. F. Shoch and J. A. Hupp, "The worm programs–early experience with a distributed computation," *Communications of the ACM*, vol. 25, no. 3, pp. 172–180, 1982.

[2] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham, "A taxonomy of computer worms," in *WORM '03: Proceedings of the 2003 ACM workshop on Rapid malcode*, pp. 11–18, ACM, 2003.

[3] E. Spafford, "The internet worm program: an analysis," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 1, pp. 17–57, 1989.

[4] D. Kienzle and M. Elder, "Recent worms: a survey and trends," *in WORM 03: Proceedings of the 2003 ACM Workshop on Rapid Malcode*, pp. 1–10, 2003.

[5] H. Berghel, "The code red worm," *Communications of the ACM*, vol. 44, no. 12, pp. 15–19, 2001.

[6] D. Moore, C. Shannon, and K. C. Claffy, "Code-red: a case study on the spread and victims of an internet worm," in *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, (New York, NY, USA), pp. 273–284, ACM Press, 2002.

[7] eEye Digital Security, "Microsoft internet information services remote buffer overflow (system level access)," *http://research.eeye.com/html/advisories/published/AD20010618.html*, Last visited: December 1st, 2007.

[8] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the Slammer Worm," *IEEE Security and Privacy*, vol. 1, no. 4, pp. 33–39, 2003.

[9] N. Weaver, "Warhol Worms: The Potential for Very Fast Internet Plagues," *University of California Berkeley, February*, 2002.

[10] S. Staniford, V. Paxson, and N. Weaver, "How to Own the Internet in Your Spare Time," *Proceedings of the 11th USENIX Security Symposium*, vol. 8, pp. 149–167, 2002.

[11] B. Mukherjee, L. Heberlein, and K. Levitt, "Network intrusion detection," *Network, IEEE*, vol. 8, no. 3, pp. 26–41, 1994.

[12] A. Lazarevic, L. Ertöz, V. Kumar, A. Ozgur, and J. Srivastava, "A comparative study of anomaly detection schemes in network intrusion detection," in *Proceedings of the Third SIAM International Conference on Data Mining*, 2003.

[13] S. Kumar and E. H. Spafford, "A Pattern Matching Model for Misuse Intrusion Detection," in *Proceedings of the 17th National Computer Security Conference*, pp. 11–21, 1994.

[14] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated Worm Fingerprinting," *OSDI '04: 6th Symposium on Operating Systems Design and Implementation*, 2004.

[15] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[16] C. Williams and D. Barber, "Bayesian classification with Gaussian processes," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 20, no. 12, pp. 1342–1351, 1998.

[17] T. Cover and P. Hart, "Nearest neighbor pattern classification," *Information Theory, IEEE Transactions on*, vol. 13, no. 1, pp. 21–27, 1967.

[18] O. Sharma, M. Girolami, and J. Sventek, "Detecting worm variants using machine learning," in *CoNEXT '07: Proceedings of the 2007 ACM CoNEXT conference*, (New York, NY, USA), pp. 1–12, ACM, 2007.

[19] J. Brunner, *The Shockwave Rider*. United Kingdom: Harper & Row, 1975.

[20] T. Aslam, I. Krsul, and E. H. Spafford, "Use of a taxonomy of security faults," in *Proc. 19th NIST-NCSC National Information Systems Security Conference*, pp. 551–560, 1996.

[21] A. Bazaz and J. D. Arthur, "Towards a taxonomy of vulnerabilities," *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pp. 163a–163a, Jan. 2007.

[22] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade," *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, pp. 227–237, 2003.

[23] F. Cohen, "Computer viruses: theory and experiments," *Computer Security*, vol. 6, no. 1, pp. 22–35, 1987.

[24] K. Lan, A. Hussain, and D. Dutta, "The effect of malicious traffic on the network," *in PAM '03: Proceedings of the Passive and Active Measurement Conference*, 2003.

[25] J. C. Foster, *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Syngress Publishing, 2007.

[26] D. Ahmad, "The rising threat of vulnerabilities due to integer errors," *IEEE Security and Privacy*, vol. 1, no. 4, pp. 77–82, 2003.

[27] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers," in *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, (Berkeley, CA, USA), pp. 16–16, USENIX Association, 2001.

[28] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a board range of memory error exploits," in *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, (Berkeley, CA, USA), pp. 8–8, USENIX Association, 2003.

[29] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, (Berkeley, CA, USA), pp. 17–17, USENIX Association, 2005.

[30] K. Anagnostakis, M. Greenwald, S. Ioannidis, A. Keromytis, and D. Li, "A cooperative immunization system for an untrusting internet," *Networks, 2003. ICON2003. The 11th IEEE International Conference on*, pp. 403–408, 28 Sept.-1 Oct. 2003.

[31] I. Arce and E. Levy, "An analysis of the Slapper worm," *Security & Privacy Magazine, IEEE*, vol. 1, no. 1, pp. 82–87, 2003.

[32] J. Viega, M. Messier, and P. Chandra, *Network Security with OpenSSL*. O'Reilly, 2002.

[33] R. T. Fielding and G. Kaiser, "The apache http server project," *IEEE Internet Computing*, vol. 1, no. 4, pp. 88–90, 1997.

[34] G. Schaffer, "Worms and viruses and botnets, oh my! rational responses to emerging internet threats," *Security & Privacy, IEEE*, vol. 4, no. 3, pp. 52–58, May-June 2006.

[35] F. Lau, S. Rubin, M. Smith, and L. Trajkovic, "Distributed denial of service attacks," *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, vol. 3, pp. 2275–2280 vol.3, 2000.

[36] D. Wagner and B. Schneier, "Analysis of the ssl 3.0 protocol," in *WOEC'96: Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce*, (Berkeley, CA, USA), pp. 4–4, USENIX Association, 1996.

[37] C. Shannon and D. Moore, "The spread of the witty worm," *IEEE Security and Privacy*, vol. 2, no. 4, pp. 46–50, 2004.

[38] M. Bailey, E. Cooke, F. Jahanian, D. Watson, and J. Nazario, "The blaster worm: Then and now," *IEEE Security and Privacy*, vol. 3, no. 4, pp. 26–31, 2005.

[39] Microsoft Corporation, "Microsoft Security Bulletin MS03-026. Buffer Overrun In RPC Interface Could Allow Code Execution (823980).," *Microsoft TechNet*, July 16, 2003.

[40] D. E. Denning, "An intrusion-detection model," *IEEE Trans. Software Engineering*, vol. 13, no. 2, pp. 222–232, 1987.

[41] H. Debar, M. Dacier, and A. Wespi, "Towards a taxonomy of intrusion-detection systems," *Computer Networks*, vol. 31, no. 9, pp. 805–822, 1999.

[42] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer Networks (Amsterdam, Netherlands: 1999)*, vol. 31, no. 23–24, pp. 2435–2463, 1999.

[43] M. Roesch, "Snort - lightweight intrusion detection for networks," in *LISA '99: Proceedings of the 13th USENIX conference on System administration*, (Berkeley, CA, USA), pp. 229–238, USENIX Association, 1999.

[44] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, (New York, NY, USA), pp. 93–102, ACM, 2006.

[45] X. Jiang and D. Xu, "Profiling self-propagating worms via behavioral footprinting," in *WORM '06: Proceedings of the 4th ACM workshop on Recurring malcode*, (New York, NY, USA), pp. 17–24, ACM, 2006.

[46] T. H. Ptacek and T. N. Newsham, "Insertion, evasion, and denial of service: Eluding network intrusion detection," tech. rep., Secure Networks, Inc., Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, 1998.

[47] H. Kim and B. Karp, "Autograph: Toward Automated, Distributed Worm Signature Detection," in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[48] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," *IEEE Symposium on Security and Privacy*, 2005.

[49] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," *String Processing and Information Retrieval. SPIRE 2000. Proceedings. Seventh International Symposium on*, vol. 00, p. 39, 2000.

[50] K. Wang and S. Stolfo, "Anomalous payload-based network intrusion detection," tech. rep., Columbia University, 2004.

[51] K. Wang, G. F. Cretu, and S. J. Stolfo, "Anomalous payload-based worm detection and signature generation.," in *RAID* (A. Valdes and D. Zamboni, eds.), vol. 3858 of *Lecture Notes in Computer Science*, pp. 227–246, Springer, 2005.

[52] R. Perdisci, G. Gu, and W. Lee, "Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems," in *ICDM '06: Proceedings of the Sixth International Conference on Data Mining*, (Washington, DC, USA), pp. 488–498, IEEE Computer Society, 2006.

[53] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: end-to-end containment of internet worms," *Proceedings of the twentieth ACM symposium on Operating systems principles*, pp. 133–147, 2005.

[54] C. Kreibich and J. Crowcroft, "Honeycomb: creating intrusion detection signatures using honeypots," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 1, pp. 51–56, 2004.

[55] N. Provos, "A virtual honeypot framework," *Proceedings of the 13th USENIX Security Symposium, August 2004*, 2004.

[56] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[57] A. K. Jain, R. P. W. Duin, and J. Mao, "Statistical pattern recognition: A review," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 22, no. 1, pp. 4–37, 2000.

[58] S. R. Michalski, G. J. Carbonell, and M. T. Mitchell, eds., *Machine learning an artificial intelligence approach volume II*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986.

[59] R. Brunelli and T. Poggio, "Face recognition: features versus templates," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 15, pp. 1042–1052, Oct 1993.

[60] F. Constantinou and P. Mavrommatis, "Identifying known and unknown peer-to-peer traffic," *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*, pp. 93–102, 0-0 2006.

[61] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, (New York, NY, USA), pp. 144–152, ACM, 1992.

[62] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. New York, NY, USA: Cambridge University Press, 2000.

[63] C. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition," *Data Mining and Knowledge Discovery*, vol. 2, no. 2, pp. 121–167, 1998.

[64] J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*. New York, NY, USA: Cambridge University Press, 2004.

[65] K.-R. Muller, S. Mika, G. Ratsch, K. Tsuda, and B. Scholkopf, "An introduction to kernel-based learning algorithms," *Neural Networks, IEEE Transactions on*, vol. 12, no. 2, pp. 181–201, Mar 2001.

[66] T. Joachims, "Text categorization with suport vector machines: Learning with many relevant features," in *ECML '98: Proceedings of the 10th European Conference on Machine Learning*, (London, UK), pp. 137–142, Springer-Verlag, 1998.

[67] O. Chapelle, P. Haffner, and V. Vapnik, "Support vector machines for histogram-based image classification," *Neural Networks, IEEE Transactions on*, vol. 10, no. 5, pp. 1055–1064, 1999.

[68] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and Computing*, vol. 14, no. 3, pp. 199–222, 2004.

[69] R. Collobert and S. Bengio, "Links between perceptrons, mlps and svms," in *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, (New York, NY, USA), p. 23, ACM, 2004.

[70] A. Ganapathiraju, J. Hamaker, and J. Picone, "Applications of support vector machines to speech recognition," *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]*, vol. 52, pp. 2348–2355, Aug. 2004.

[71] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins, "Text classification using string kernels," *The Journal of Machine Learning Research*, vol. 2, pp. 419–444, 2002.

[72] K. Rieck and P. Laskov, "Detecting unknown network attacks using language models," *Proc. DIMVA*, pp. 74–90, 2006.

[73] C. W. Hsu, C. C. Chang, and C. J. Lin, "A practical guide to support vector classification," tech. rep., Taipei, 2003.

[74] S. S. Keerthi and C.-J. Lin, "Asymptotic behaviors of support vector machines with gaussian kernel," *Neural Computation*, vol. 15, no. 7, pp. 1667–1689, 2003.

[75] H. T. Lin and C. J. Lin, "A study on sigmoid kernels for svm and the training of non-psd kernels by smo-type methods," technical report, Department of Computer Science, National Taiwan University, March 2003.

[76] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[77] D. Mackay, "Gaussian processes - a replacement for supervised neural networks," *Lecture notes for a tutorial at Neural Information Processing Systems, NIPS 1997*, 1997.

[78] J. C. Platt, "Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods," in *Advances in Large Margin Classifiers*, pp. 61–74, MIT Press, 1999.

[79] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *Journal of the ACM*, vol. 45, no. 6, pp. 891–923, 1998.

[80] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals Eugen.*, vol. 7, pp. 179–188, 1936.

[81] I. Gokcen and J. Peng, "Comparing linear discriminant analysis and support vector machines," in *ADVIS '02: Proceedings of the Second In-*

ternational Conference on Advances in Information Systems, (London, UK), pp. 104–113, Springer-Verlag, 2002.

[82] Y. Lee, Y. Lin, and G. Wahba, "Multicategory support vector machines, theory, and application to the classification of microarray data and satellite radiance data," Tech. Rep. TECHNICAL REPORT NO. 1064r, Department of Statistics, University of Wisconsin, May 14, 2003.

[83] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and Regression Trees*. Chapman & Hall/CRC, January 1984.

[84] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Netw.*, vol. 2, no. 5, pp. 359–366, 1989.

[85] S. Mukkamala, G. Janoski, and A. Sung, "Intrusion detection using neural networks and support vector machines," *Neural Networks, 2002. IJCNN '02. Proceedings of the 2002 International Joint Conference on*, vol. 2, pp. 1702–1707, 2002.

[86] D. Program, "Transmission Control Protocol, RFC 793," tech. rep., DARPA Internet Program, 1981.

[87] J. Postel, "User datagram protocol," *Internet Engineering Note IEN-88*, 1979.

[88] Lawrence Berkeley Laborartory, "The libpcap project," *http://www.tcpdump.org/pcap.htm*, Last visited: November 10th, 2007.

[89] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[90] J. Paisley and J. Sventek, "Real-time detection of grid bulk transfer traffic," *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pp. 66–72, 0-0 2006.

[91] A. Dunkels and O. Schmidt, "Protothreads – Lightweight Stackless Threads in C," Tech. Rep. T2005:05, Swedish Institute of Computer Science, 2005.

[92] M. Roberts, "Local-order-estimating Markovian analysis for noiseless source coding and authorship identification," tech. rep., UCRL-53310, Lawrence Livermore National Lab., CA (USA), 1982.

[93] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[94] J. Hall, "Distributed computing with distcc," *Linux Journal*, vol. 2007, no. 163, p. 4, 2007.

[95] "Minishare," *http://minishare.sourceforge.net/*, Last visited: January 21st, 2008.

[96] M. Zweig and G. Campbell, "Receiver-operating characteristic (ROC) plots: a fundamental evaluation tool in clinical medicine [published erratum appears in Clin Chem 1993 Aug; 39 (8): 1589]," *Clinical Chemistry*, vol. 39, no. 4, pp. 561–577, 1993.

[97] A. Bradley, "Use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

[98] C.-C. Chang and C.-J. Lin, *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`. Last visited: March 21st, 2008.

[99] C. Kruengkrai, V. Sornlertlamvanich, and H. Isahara, "Language, Script, and Encoding Identification with String Kernel Classifiers," in *Proc. Conference on Knowledge, Information and Creativity Support Systems*, 2006.

[100] C.-W. Hsu and C.-J. Lin, "A comparison of methods for multiclass support vector machines," *Neural Networks, IEEE Transactions on*, vol. 13, no. 2, pp. 415–425, Mar 2002.

[101] M. Girolami and S. Rogers, "Variational bayesian multinomial probit regression with gaussian process priors," *Neural Computation*, vol. 18, no. 8, pp. 1790–1817, 2006.

[102] B. Gough, ed., *GNU Scientific Library Reference Manual - Second Edition*. Network Theory Ltd., 2003.

[103] T. G. Dietterich, "Machine-learning research: Four current directions," *The AI Magazine*, vol. 18, no. 4, pp. 97–136, 1998.

[104] R. P. W. Duin and D. M. J. Tax, "Experiments with classifier combining rules," in *MCS '00: Proceedings of the First International Workshop on Multiple Classifier Systems*, (London, UK), pp. 16–29, Springer-Verlag, 2000.

[105] S. Džeroski and B. Ženko, "Is combining classifiers with stacking better than selecting the best one?," *Mach. Learn.*, vol. 54, no. 3, pp. 255–273, 2004.

[106] T. Fawcett, "Roc graphs: Notes and practical considerations for data mining researchers," Tech. Rep. HPL-2003-4, HP Labs, 2003.

[107] "Distcc security notes," *http://distcc.samba.org/security.html*, Last visited: May 5th, 2008.

[108] R. Stallman, *Using the GNU Compiler Collection*. Free Software Foundation, Inc., Cambridge, Massachusetts, 2003.

[109] SecurityTracker, "Minishare buffer overlow in processing long urls lets remote users execute arbitrary code," *http://securitytracker.com/alerts/2004/Nov/1012106.html*, Last visited: January 21st, 2008.

[110] D. Thomas and A. Hunt, *Programming Ruby: the pragmatic programmer's guide*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[111] J. Aase, "Warftp daemon," *http://www.warftp.org*, Last visited: January 21st, 2008.

[112] Security Focus, "Warftp username stack-based-overflow vulnerability:," *http://www.securityfocus.com/bid/22944*, Last visited: November 29th, 2007.

[113] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Indianapolis, USA: Addison-Wesley Publishing Company, 1995.

[114] Accordata, "Port proxy v0.95," *http://www.accordata.de/downloads/port-proxy/index.html*, Last visited: July 10th, 2008.

[115] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[116] O. Kolesnikov and W. Lee, "Advanced polymorphic worms: Evading ids by blending in with normal traffic," Tech. Rep. GIT-CC-04-15, Georgia Tech, 2004.

[117] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee, "Polymorphic blending attacks," in *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, (Berkeley, CA, USA), pp. 17–17, USENIX Association, 2006.

[118] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif, "Misleading-worm signature generators using deliberate noise injection," in *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, (Washington, DC, USA), pp. 17–31, IEEE Computer Society, 2006.

[119] Q. Sun and G. DeJong, "Feature kernel functions: Improving svms using high-level knowledge," *Conference on Computer Vision and Pattern Recognition*, vol. 2, pp. 177–183, 2005.

[120] D. Zhang and W. S. Lee, "Question classification using support vector machines," in *SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, (New York, NY, USA), pp. 26–32, ACM, 2003.

[121] M. Rabin, "Fingerprinting by random polynomials," Tech. Rep. TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[122] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "NetFPGA – An Open Platform for Gigabit-Rate Network Switching and Routing," *IEEE International Conference on Microelectronic Systems Education*, vol. 0, pp. 160–161, 2007.

[123] Y. Chen and C. Lin, "Combining svms with various feature selection strategies., in feature extraction, foundations and applications," in *Taiwan University*, Springer-Verlag, 2005.

[124] G. Cauwenberghs and T. Poggio, "Incremental and decremental support vector machine learning," in *Advances in Neural Information Processing Systems*, pp. 409–415, 2000.

[125] L. Hamel, "Model assessment with ROC curves," *The Encyclopedia of Data Warehousing and Mining, Second Edition*, 2009.