University of BRISTOL

DEPARTMENT OF COMPUTER SCIENCE

Virus detection with machine learning

Christopher Richardson

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering

September 2009 | CSMSC-09

Abstract

Standard virus detection relies on the use of signatures, which are a small number of bytes from a virus.

In this project we present an alternate approach to virus detection through the use of machine learning techniques, to detect viruses based on their behaviour instead.

A virtual environment was created to allow real viruses to propagate. In order to obtain realistic results, we attempted to simulate real-world computer usage on the virtual machines we used.

A virus 'observatory' was then designed to visualise virus propagation.

Windows perfmon counters were used to obtain a numeric representation of the computer's state. These counters from machines, were then visualised in a matrix format.

We applied machine learning techniques to detect the name of the virus active on a computer. We consider how to improve the accuracy of the classifiers through the use of information gain.

Finally we attempted to detect viruses after our system was only trained using 'normal' activity. This gives the system the possibility for being used to detect unknown viruses.

Acknowledgement

I would like to thank my supervisor Rafal Bogacz for his invaluable support and encouragement with this project.

I would also like to thank my family for their support, interest and for putting up with my antisocial working hours.

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Christopher Richardson, September 2009

Contents

1 Aims and objectives

The aim of the project was to develop a distributed system capable of using machine learning techniques to detect anomalous behaviour and prevent the further spread of viruses.

According to Kaspersky labs in April 2009 alone, 45190 unique instances of malware where found on their customers computers [1]. This worryingly high number is only likely to increase, especially as the malware author's incentives for writing such software is now mainly a financial one.

Traditional virus detection relies on the use of signatures, "which consist of sequences of bytes in the machine code of the virus. A good signature is one that is found in every object infected by the virus, but is unlikely to be found if the virus is not present." [5].

The problem with the use of signatures alone is that new types of viruses such as polymorphic ones, are able to mutate their code, making signature generation difficult/impossible. By utilising a machine learning approach we aim to train our system to detect activity which appears anomalous, meaning the system will be able to detect new viruses without needing to be explicitly trained of their presence.

Objectives:
The following is a list of the original objectives for the project, these are explained in more detail in subsequent sections.

- Build a virtual network to safely observe virus propagation
- Capture or download viruses
- Design a visualisation to view the spread of viruses
- Allow the virtual machines to log system & network activity (features)
- Produce a visualisation of these features in the form of a matrix
- Use formulae such as information gain to automatically select important features
- Develop a system to allow the exchange of each virtual machine's feature set with other machines, with the goal of improving each machine's ability to detect a virus.
- Rigorous testing of the system will be necessary to assess its performance.

[1] http://www.viruslist.com/en/analysis?pubid=204792060

2 Literature review

We will first begin by giving an overview of the traditional approaches to virus detection. Then we will review relevant material on using machine learning techniques in virus detection.

2.1 Traditional approaches

The umbrella term for malicious software is malware, covering software such as spyware, trojan horses and viruses. We will concentrate on viruses, specifically worms which "are network viruses, primarily replicating on networks" [10].

As previously mentioned the majority of anti-virus scanners make use of signatures to detect viruses, however this technique isn't without fault. Didier Stevens showed that by zero padding a malware script with a specific amount of zero bytes it became undetectable by all virus scanners he tested it with [9]. He made use of VirusTotal [2], a site which makes use of many anti-virus scanners to determine whether a file uploaded to it, is infected or not. Other problems with the signature based approach include the ability of a virus writer to produce many copies of a virus, by re-arranging the source code, so that there is no common signature between each copy, making the job of collecting each virus and then creating a signature for it a lot more time consuming.

Additionally the job of creating virus signatures becomes ever more difficult with the increasing number of polymorphic and metamorphic viruses. A polymorphic virus makes use of a decrypter/encrypter stub, to make the creation of a signature more difficult. This is because a different key is used each time the virus propagates to a different computer, so the body of the virus is completely different.

"When dormant, the body remains in encrypted form. When activated, the header is the first section of code to receive control from the operating system. The header decrypts the body in memory." [13]

It may be possible for the the anti-virus vendor to make a signature from the decrypter instructions, as there must be some executable portion of code left for the virus to be able to decrypt itself. Metamorphic viruses could potentially make the ability to create a signature impossible, as metamorphic viruses are able to 'evolve', effectively being able to re-code themselves. This may involve techniques such as substituting the machine code registers used for alternatives or inverting the logic of statements.

2.2 Fan patterns

An "invariant for long time periods and different scales (subnet sizes) or traffic types (protocols) is proportion between a number of internal (Fan-in) and external (Fan-out) data flows" [4].

[2] http://www.virustotal.com

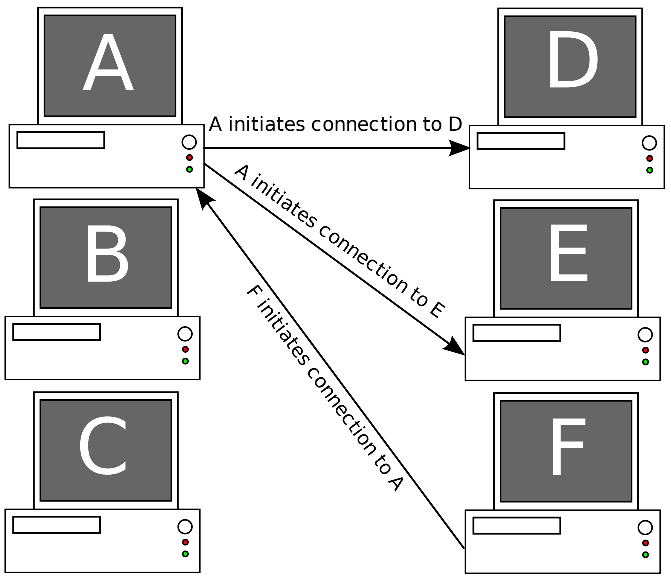

A
D
B
E
C
F
A initiates connection to D
A initiates connection to E
F initiates connection to A

Figure 1: Fan patterns

Figure 1 depicts network flow patterns between different hosts. A exhibits a network fan-out pattern, as it has initiated two connections to hosts D and E. It also has a fan-in, as it has a connection initiated by another host F. Network flows can normally only be considered for TCP based traffic as it is a stateful protocol, however we can assume UDP (User Datagram Protocol) packets to act as part of the same flow if the data is sent to the same port from the same IP address in a timely fashion.

By simply calculating the ratio of fan-out to fan-in, a value should be obtained which stays fairly constant across time. In the event of a virus attack for instance, the number of fan-out connections, will possibly dramatically increase, as is the case with the Sasser virus, which initiates 128 threads for the sole purpose of scanning for other machines. This ratio could be fed into the novel classifier as another 'feature' to determine anomalies for.

2.3 Novelty classifiers

As our system makes use of machine learning techniques rather than signatures, we need to detect abnormality in the system.

This can be accomplished by two methods:

The first being to train the system with both normal and abnormal data, meaning there are two classes. This does somewhat lengthen the training process, as it will then be necessary to train the system under both general computer usage and then also when a virus is running on the network. Another issue with this approach is that the system would need to be trained with knowledge of many viruses, which could prove infeasible, as well as making the system 'aware' of specific viruses rather than a broader generalisation of what a virus is.

The second approach is to use a novelty classifier, where only one class is required. This means the system only needs to be trained with normal system activity, allowing it to infer what activity is abnormal.

Any approach involving machine learning can never be 100% effective in preventing the propagation of viruses. However any approach which can somewhat curb the proliferation of such nefarious code must be surely beneficial. It is important to note that also even traditional approaches to virus detection using signatures can never be completely effective, as the bytes chosen as signatures from malicious files, may also appear innocently in ordinary programs.

In [12] Yeung and Chow make use of the Parzen window as a method for detecting network anomalies for their Network Intrusion Detection System. They note that the Parzen window is non-parametric and as such "provided that sufficient data [is] available, the non-parametric approach can model arbitrary distributions without being restricted to special functional forms" [12]. With virus detection it is easily possible to obtain plenty of data, to represent normal network activity.

We now describe the use of the Parzen window for novelty detection. The following formula represents the Parzen window with a Gaussian kernel for 1 dimension i.e. when only one attribute (feature) is used to describe the state of the system.

$$\hat{P}(x) = \frac{1}{n(2\pi)^{1/2}\sigma} \sum_{i=1}^{n} exp(-\frac{(x - x_i)^2}{2\sigma^2}) \tag{1}$$

Where:

- x - Value of the attribute for the state of the system being tested.
- x_i - A value of the attribute from one of the instances the system has been trained with.
- n - Number of samples in the training set
- σ - Bandwidth of the kernel, this needs to be large enough so that instances of the attribute can form clusters, which denote non-novel data.

To generalise the formula for an arbitrary number of attributes (dimensions):

$$\hat{P}(x) = \frac{1}{n(2\pi)^{d/2}\sigma^d} \sum_{i=1}^{n} exp(-\frac{\sum_{t=1}^{d}(x_t - x_{i_t})^2}{2\sigma^2}) \tag{2}$$

Where:

- x_t - Represents the value of the t^{th} attribute for the testing instance.

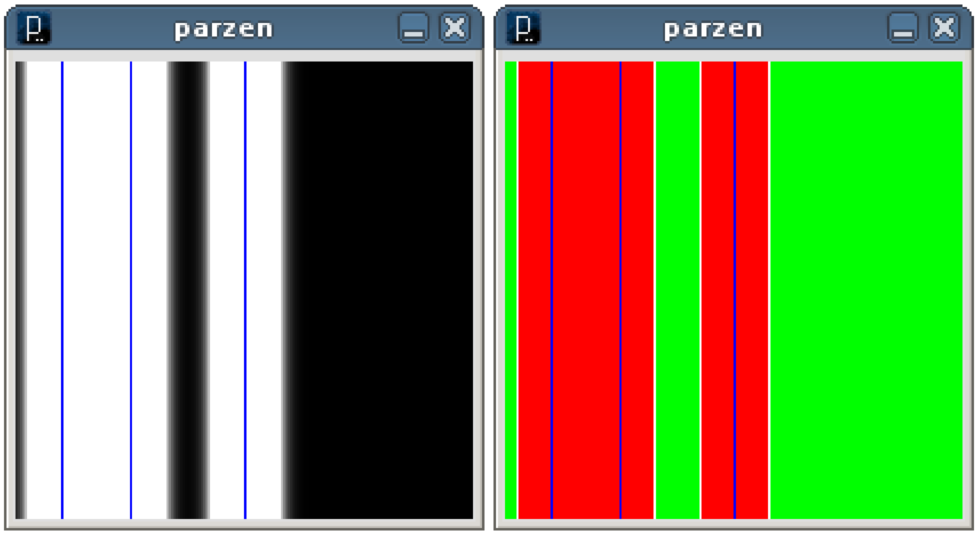


Figure 2: 1D Parzen window, with and without thresholding

- x_{i_t} - Represents t^{th} attribute of the i^{th} instance of the training data.
- d - Number of attributes describing instances.
- n - Number of instances (an instance for this purpose refers to a set of data for all available attributes recorded at a specific moment of time).

To test the performance of the Parzen window I created an implementation of it in Java using Processing, a multimedia library.

Figure 2 depicts a 1D implementation of it. The values of x_i are 20, 50 and 100, these are represented by the thin blue lines. The width of the dialog is 200 pixels, representing values 0 to 199. "The density estimator p(x) obtained from the training set give us a quantitative measure of the degree of novelty for each new example" [2], this is what is plotted for all x values 0 - 199. The intensity of this value is plotted in the left dialog in white (black represents zero). The Gaussian kernel gives the white blob a feathered edge.

The right dialog depicts the output of the classifier with a threshold applied. The red blobs represent areas of non-novel data, whereas the green background denotes the area classified as novel.

Figure 3 depicts a 2D implementation of the classifier. I created a number of instances of x,y coordinates shown as blue pixels to train the classifier with.

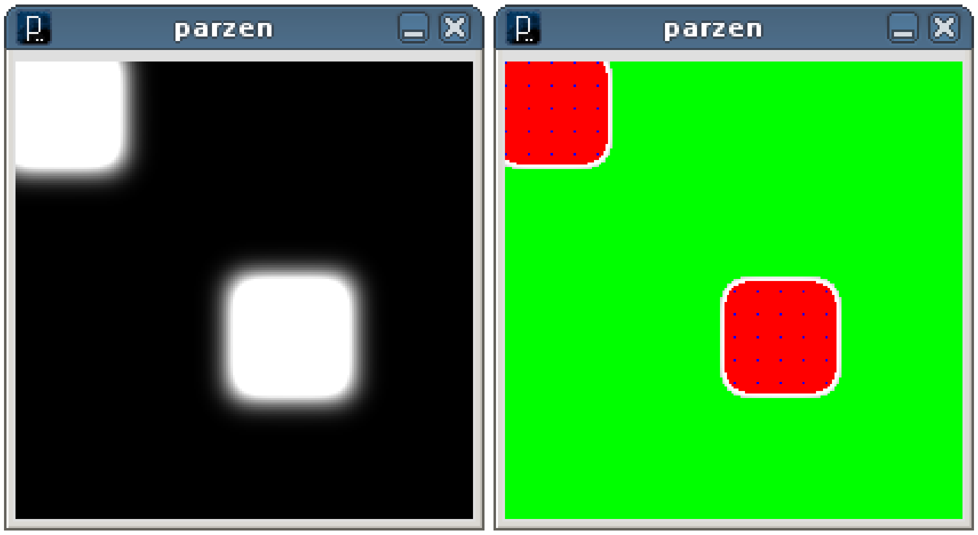


Figure 3: 2D Parzen window, with and without thresholding

2.4 Naive bayes

The simplest version of naive bayes for discrete data is:

$$c = \arg_{C_i} \max P(c_i) \prod_j P(a_j|c_i) \tag{3}$$

Or:

$$c = \arg_{C_i} \max \log P(c_i) \sum_j log P(a_j|c_i) \tag{4}$$

(Formulae obtained from the Introduction to Machine Learning course slides)

Where:

- $P(c_i)$ - Represents the probability that data is in that class
- a_j - Represents a particular attribute
- c_i - Represents a particular class

Equation 4 is preferred as it ensures that the total value is kept to a minimum and also it doesn't matter if one probability is zero.

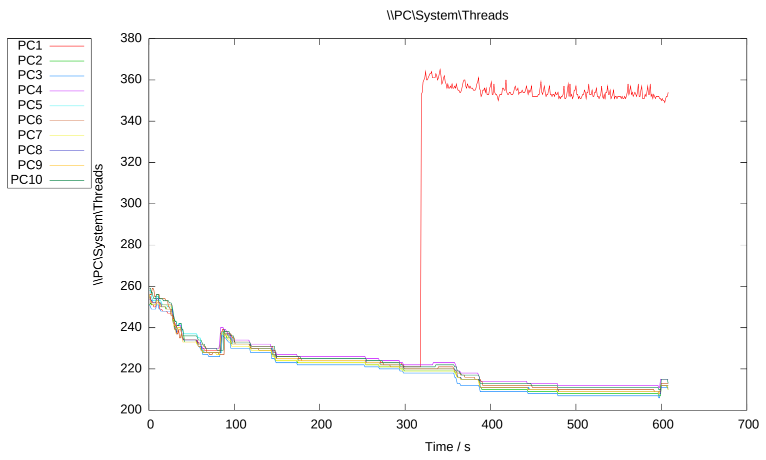
2.5 System patterns

Figure 4(a) depicts the total number of system threads on 10 different virtual machines. After approximately 5 minutes the Sasser virus was initiated on PC1, the number of threads can be seen to dramatically increase for this PC. Under normal usage such a sharp increase in the number of system threads would be unusual but not impossible. More investigation would be needed to determine how useful this feature is.

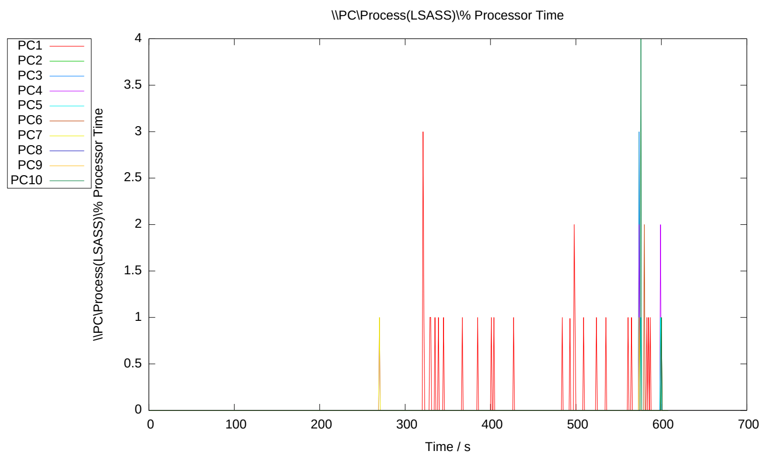
Figure 4(b) represents the processor usage of the LSASS (Local Security Authority Subsystem Service) a system service which handles the authentication of users. As the virus was first used to infect PC1 the many red spikes represent the virus compromising LSASS.

Figure 4(c) represents the "number of logon attempts for local, across the network, and service account authentication that occurred since the server was last started up" [1]. This number gradually increases for PC1 as it tries to infect other machines on the network.

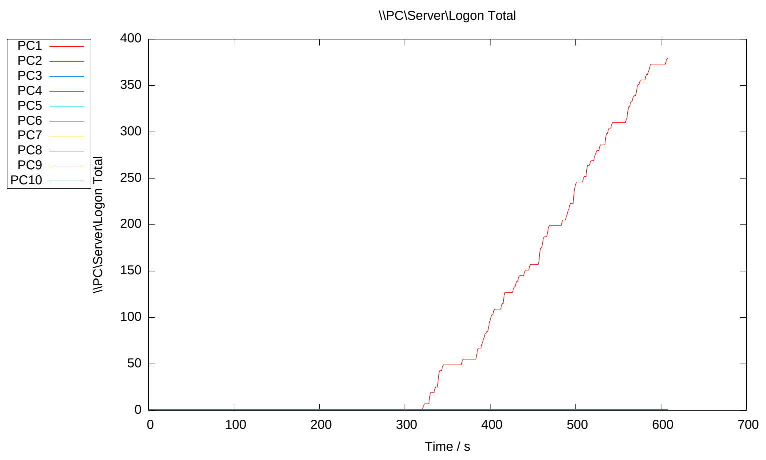
Each of these features may be useful. Later we quantify how effective different features appear to be, i.e. ones which improve our system's ability to detect viruses the most, whilst still avoiding false positives.



(a) System thread count under Sasser virus (from BSc dissertation [7])



(b) LSASS processor time under Sasser virus (from BSc dissertation [7])



(c) Logon total under Sasser virus (from BSc dissertation [7])

2.6 Choosing features

In order for our system to automatically choose features which are the most important in detecting a virus, it was necessary to compare how significant each feature is in the detection of a virus, one potential way in which this can be accomplished is through the use of the ‘information gain’ formula.

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \tag{5}$$

Where:

- S - represents the training set
- A - represents the attribute that we are comparing
- $|S|$ - represents the number of elements in the training set
- $|S_v|$ - represents the number of elements where the attribute has a particular value.

$$Entropy(S) = \sum_i -p_i log_2 p_i \tag{6}$$

The calculation of information gain is illustrated below with a brief calculation:

Number of TCP resets	**Before/After Infection**
0	before
0	before
0	after
1	after

As we are trying to find the information gain from a signal before virus infection to after virus infection, it is necessary to calculate the entropy of the before/after data.

Entropy(Training set) = -0.5 log2 0.5 - 0.5 log2 0.5 = **1**

Where v = 0:
$|Sv| = 3$
$|S| = 4$
$Entropy(S_0) = -2/3 \; log2 \; 2/3 \; - \; 1/3 \; log2 \; 1/3 = \mathbf{0.92}$

3/4 * 0.92 = 0.69

Where v=1:
$|Sv| = 1$
$|S| = 4$
$Entropy(S_1) = -0/1 \; log2 \; 0/1 \; - \; 1/1 \; log2 \; 1/1 = \mathbf{0}$

Entropy has to be zero, as all elements are the same
1/4 * 0 = 0

Therefore:
1 - (0.69 + 0) = **0.31** bits of extra information are needed to represent the transformation of ‘before’ to ‘after’.

It is likely that the most important features will have a large number of bits of information gain, due to the signal significantly altering during virus activity. This is verified later on through the use of experiments.

The following is an example of calculating the information gain using real world data obtained from the Sasser virus.

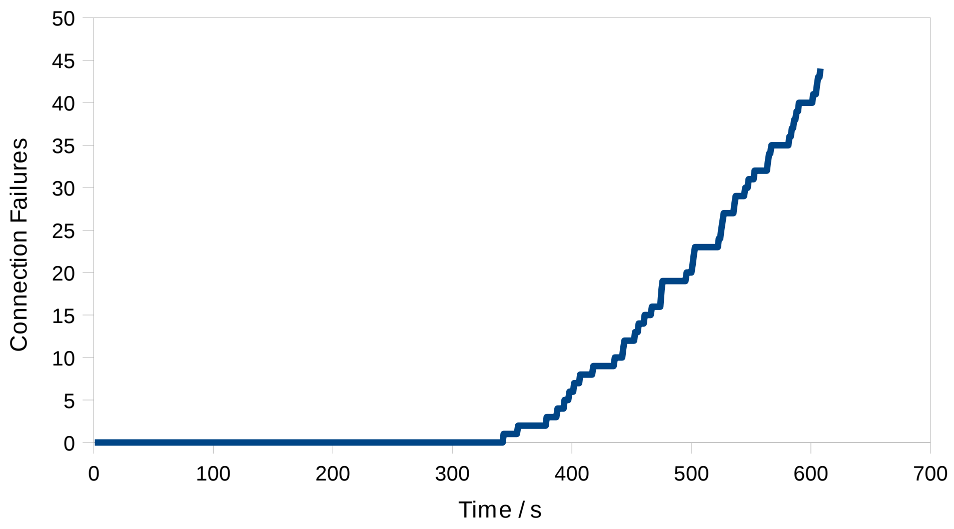


Figure 4: Graph representing the number of TCP connection failures with the Sasser virus. The virtual machine is infected at 300s. (graphed based on data obtained from my BSc dissertation[7])

Figure 4 depicts 10 minutes worth of activity from a virtual machine. The first 5 minutes (300 seconds) depict the number of TCP connection failures, when no virus was running. The remaining 5 minutes depict the number of failures when the virus is running. There are number of reasons why the number of TCP connection failures would increase when the virus is running, the most likely being, that as it attempts to scan for other computers on the network it will attempt connection to non-existent machines, leading to the increase of connection failures.

In order to calculate the information gain for this example we wrote a Java program to process values of TCP connection failures before and after the infection. Figure 5 depicts a histogram of the before and after data. The calculated value of information gain was **0.70** bits.

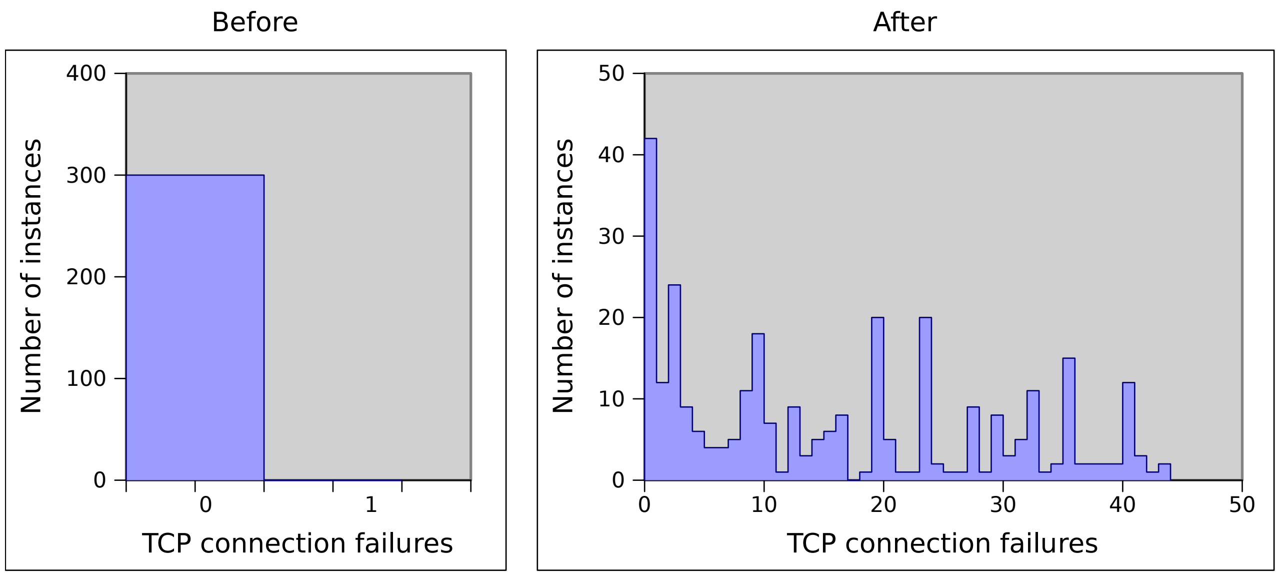


Figure 5: Histograms of the number of samples of TCP connection failure values (again generated based on data gathered during my BSc dissertation)

2.7 Classifying malware by behaviour

In [8] they make use of nepenthes [3] which acts as a honeypot to catch malware.

They then passed the malware into CWSandbox [4] a tool used to analyse the behavior of malware. The behaviour logged includes activity such as "changes to the Windows registry, e.g., creation or modification of registry keys".

They pre-process the textual report logged by CWSandbox and then feed the processed textual data into a SVM (Support Vector Machine).

They explain that with their system "on average 88% of the provided testing binaries are correctly assigned to malware families."

2.8 ROC curves

Simply counting the number of times virus-like activity is successfully detected by classifiers, when different viruses are released onto the virtual testbed isn't alone enough to quantify the accuracy of our system.

There are different types of errors which need to be taken into account.

- A false positive in this case will denote a situation where the system believes it has detected a virus, when there was none.

[3]http://nepenthes.carnivore.it/
[4]http://www.cwsandbox.org/

- A false negative denotes the case when the system informs us the system is clean, when in fact it is infected.

The worst type of error would be the false negative, as the user would mistakenly think their system is clean when in fact it is not. It is also important that the system is able to minimise the number of false positives, as a high number of these makes the system unusable. Therefore in order to test the accuracy of the system at detecting viruses, it is necessary to use ROC curves, to plot the fraction of true positives against false positives. ROC curves can be used to visually depict how accurate a particular classifier is. The area under curve (AUC) gives a quantitative measure for the accuracy, 1 represents a perfect classifier and 0.5 represents random performance.

2.9 Progressive Susceptible Infectious Detected Removed (PSIDR)

In [11] they describe their PSIDR model of virus propagation and cleanup, based on epidemiological models.

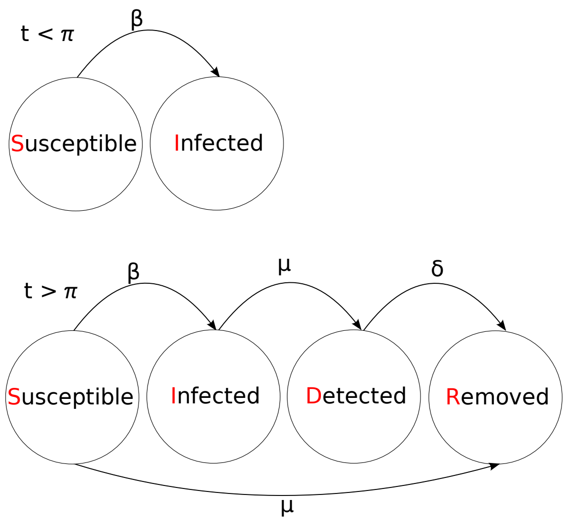


Figure 6: PSIDR epidemiological model (based on diagram in [11])

In their model a machine on a network is in one of four states. S, means the machine is susceptible to infection. I, means the machine is currently infected. D, means the virus has been detected, but not yet removed. R, means the threat of the particular virus has been removed. Initially machines are all susceptible and there is no signature available to disinfect the virus, as denoted by t $< \pi$ (the time before signature release).

The following variables represent the rate of progression of the machines to another state:

- β - Represents the rate of infection of the virus.

- μ - Rate of distribution of the signature
- δ - Cleanup rate
- $t < \pi$ - Time before signature release
- $t > \pi$ - Time after signature release

Figure 6 depicts when a machine is allowed to progress from one state to another.

The aim of the model is to depict the effects of a virus infection on a large number of machines and how quickly it can be cleaned up. Due to the simplicity of the model it can't take into account different network topologies.

Figure 7 depicts frames from an animation of the PSIDR model I generated by writing a Java program. Each pixel represents a different machine in the 'virtual' network.

The colours denote each stage in PSIDR:

- Green - **S**usceptible
- Red - **I**nfected
- Black - **D**etected
- White - **R**emoved

The PSIDR model shows how important it is that a signature is released as soon as possible. With conventional anti-virus systems a signature has to be created, this can lead to a considerable delay before the virus is detected on consumer's computers. The anti-virus company must first obtain a copy of the virus and then generate a signature, which is likely mainly an automatic process (unfortunately information on how these signatures are actually generated in reality is sparse, due to being a trade secret). Most anti-virus programs use a pull mechanism for signature updates, which means the anti-virus program itself must download signatures at a specific interval, likely daily. This gives the virus free reign to spread until a signature has been released and downloaded. This can be seen in figure 8, where the number of infected nodes dramatically decreases once the signature has started to be downloaded.

3 Design

3.1 Overview of the system

The system consists of three main components:

The system uses a virtual network to allow the propagation of viruses within a closed and secure environment (See section 3.2). This makes the system a lot safer as there is no risk of viruses escaping onto an important network. It also gives us more flexibility in choosing

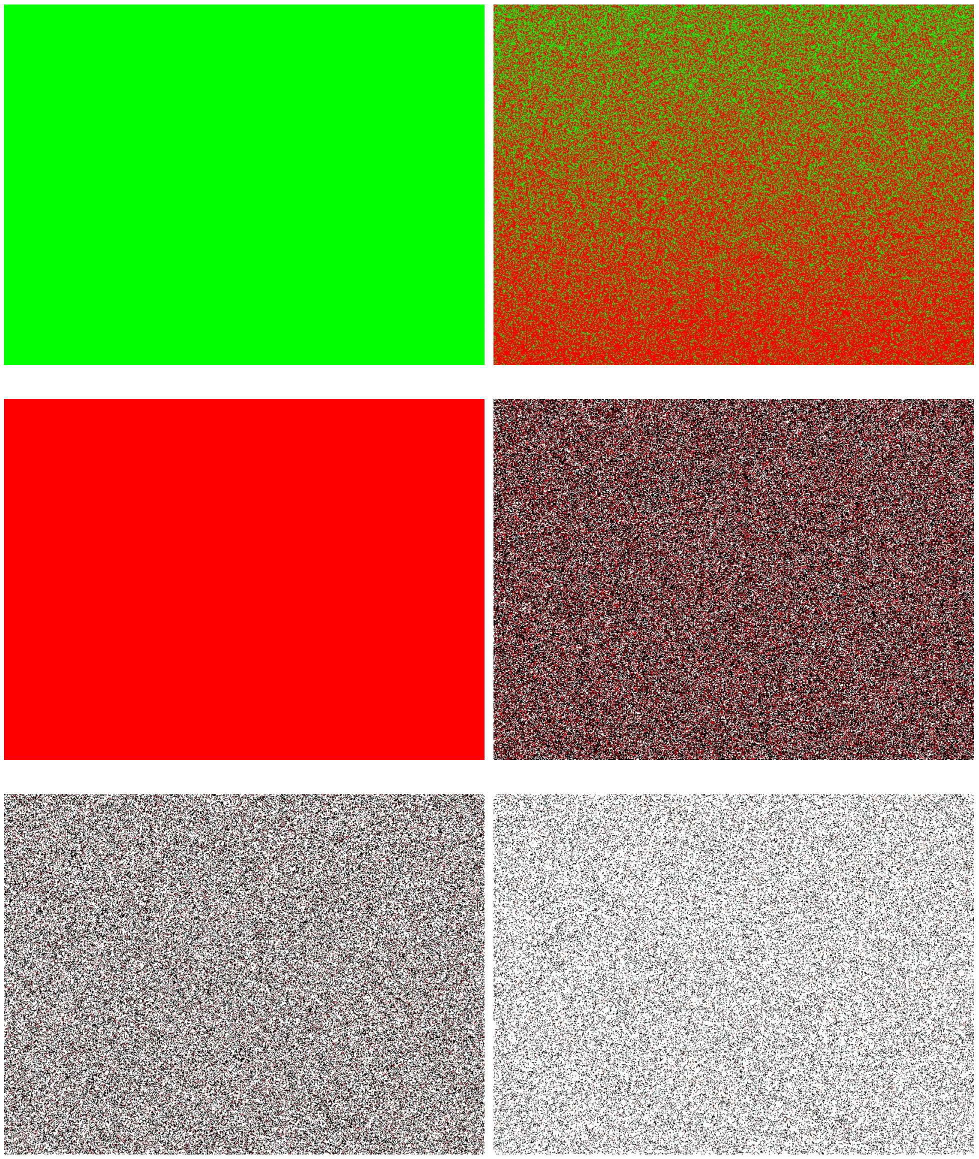


Figure 7: Visualisation of the PSIDR model

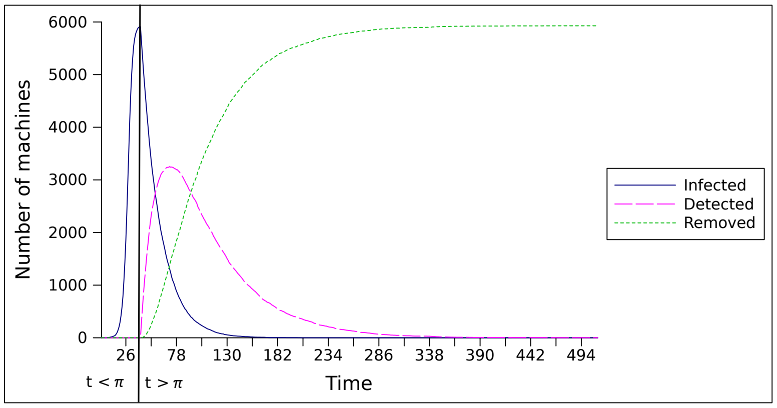


Figure 8: We generated this graph using 5929 'virtual machines'. $\beta = 0.4$, $\mu = 0.05$ and $\delta = 0.02$. (a similar graph can be seen in [11])

the topology of the network, as, for instance virtual switches can easily be added, which is much less expensive than using physical hardware. Additionally by using virtual machines (VMs) and a virtual network, the VMs can easily be 'cleaned' of virus infections.

The system then selects and computes various system features which may indicate potentially anomalous system activity (See section 3.3). The features are selected based on their ability to detect anomalous activity e.g. how much their value changes from when the system is under normal activity to when a virus is introduced.

Once features which appear to be able to indicate virus activity have been found they are combined with machine learning techniques, to train the system to produce an output indicating whether there is activity which indicates the presence of a virus on the system. (See section 3.4)

Figure 9 depicts the virtual environment which acts as a secure containment for the viruses. The arrows depict the virtual network connections between the nodes (the topology shown is arbitrary). Each virtual machine has the ability to log features, which will then be fed into machine learning classifiers in order to train the system what should be classified as normal activity.

3.2 Virtual network

In order to examine viruses it was necessary to make use of a virtual machine (VM). As the majority of viruses are windows based, Windows was the OS of choice for the VMs, specifically Windows XP as it requires less computational power than Windows Vista, meaning more VMs can be run on our physical machine.

There are many different virtual machines available, some of the most popular being:

- KVM

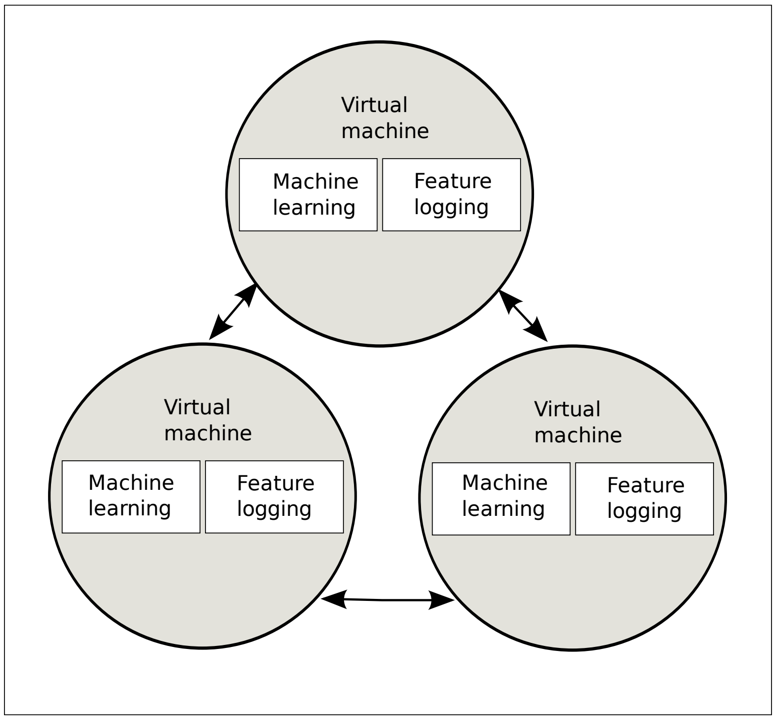


Figure 9: System overview

- Vmware Server
- Virtualbox
- Xen

Most VM software provides the ability to treat the virtual hard-disk as immutable, this means that although the viruses appear to modify the hard-disk when the system VM is running, no changes are ever actually committed to the disk image.

To complete this objective a script was designed to invoke a specified number of virtual machines and initiate virus execution on a particular VM. It is important that communication with the VMs doesn't take place using sockets, as this control data may potentially contaminate the network data we capture for analysis. A virtual serial port was instead used to control the VMs.

Most VM packages support a simple one switch network. More complex networks can be formulated using software from the Virtual Distributed Ethernet (VDE) project, which allows the creation of a virtual network with a custom topology.

Virtual NetManager allows the creation of VDE networks using a GUI, figure 10 depicts a simple network topology designed with this.

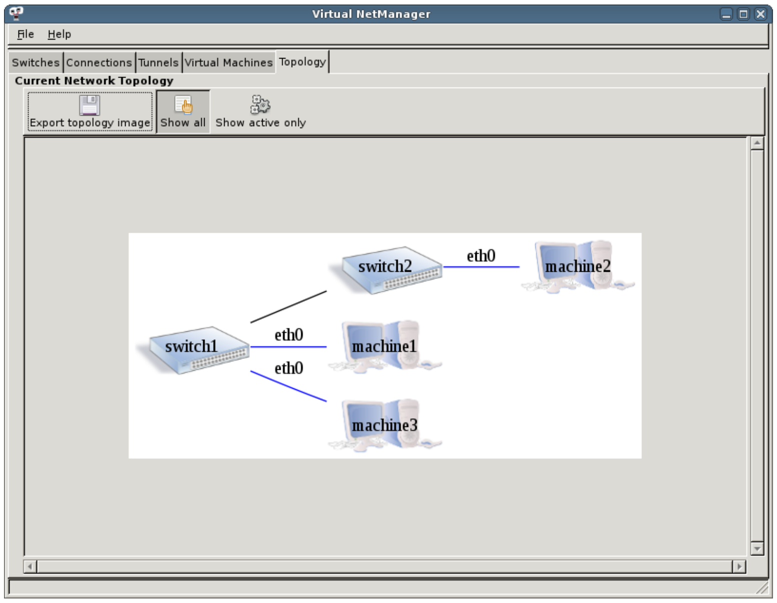


Figure 10: Network manager program

3.3 Features

In order to apply any machine learning techniques to detect viruses it was necessary to design features to provide relevant information.

3.3.1 Local data logging

It is necessary to log data which will be useful in the detection of viruses. Windows provides access to many different system counters through the Performance Counters API [5]. As this provides such a rich source of information it was necessary to devise an approach to automatically select the most appropriate counters.

3.3.2 Feature matrix

Features can visualised on a 2D matrix, where each pixel denotes a different feature. This makes it possible to view a vast amount of data simultaneously. A real-time view of this data enables users to easily see which features are most relevant in virus detection. This could therefore allow users to select features both manually, as well as automatically using an algorithm. It would be interesting to see whether a human picks features similar to the ones chosen by an algorithm.

[5] http://msdn.microsoft.com/en-us/library/aa373078(VS.85).aspx

3.4 Machine learning techniques

The naive bayes classifier will be used. As well as our novelty classifier using the Parzen window. Additionally the One Class Classifier will be used [6].

3.5 Obtaining viruses

We have a large number of viruses obtained from VX heavens [7]. Another small objective was developing a system to capture viruses for use in experiments, directly from the Internet. See 4.4 for more information on the implementation of this.

3.6 Novel detection in a collaborative system

As our virus detection system consists of many machines which will all be logging features, one possible way to judge an anomaly, would be to train the Parzen window classifier with samples from all machines except the machine trying to determine whether it exhibits anomalous behaviour. The system's own instances of features could then be used as test data for the Parzen window, to see if its state appears anomalous in relation to the state of all other machines, assuming that the majority of machines are not infected with a virus.

In [6] they mention that "distributed viral response mechanisms require some degree of trust between the automated agents", this is so important as a system's own security is then dependant on others. In the model they provide, they make use of agents running on PCs, where agents exchange information relating to the number of infected nodes on the network, so that the system can quantify the threat of a particular virus.

4 Implementation

4.1 Virtual network

A virtual environment was created to observe the effects of virus infection safely.

In order to create a virtual network, virtual machines running Windows XP were created.

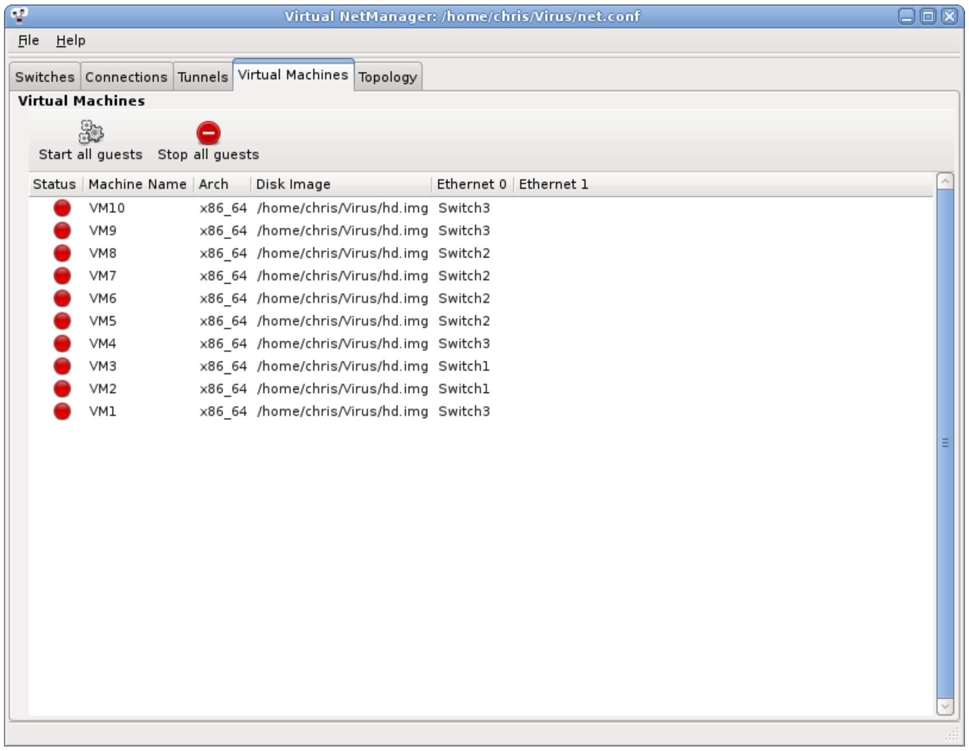
The virtual machine software KVM was chosen because of its integration with the Virtual NetManager, software which can create a virtual network.

Figure 11 depicts the program used to create the virtual network. It allows the creation of switches on a virtual network and easily allows switches to be connected together and virtual machines to be connected to the switches.

We created a number of BASH (the default Linux shell) scripts to handle the launch of the VMs and dealing with the data they produce.

[6]OneClassClassifier - http://www.cs.waikato.ac.nz/~kah18/occ/

[7]http://vx.netlux.org/



Status	Machine Name	Arch	Disk Image	Ethernet 0	Ethernet 1
●	VM10	x86_64	/home/chris/Virus/hd.img	Switch3	
●	VM9	x86_64	/home/chris/Virus/hd.img	Switch3	
●	VM8	x86_64	/home/chris/Virus/hd.img	Switch2	
●	VM7	x86_64	/home/chris/Virus/hd.img	Switch2	
●	VM6	x86_64	/home/chris/Virus/hd.img	Switch2	
●	VM5	x86_64	/home/chris/Virus/hd.img	Switch2	
●	VM4	x86_64	/home/chris/Virus/hd.img	Switch3	
●	VM3	x86_64	/home/chris/Virus/hd.img	Switch1	
●	VM2	x86_64	/home/chris/Virus/hd.img	Switch1	
●	VM1	x86_64	/home/chris/Virus/hd.img	Switch3	

Figure 11: Virtual NetManager

The main script which handles this is **runVirtualNetwork**. It accepts one argument, the name of the virus to be executed on the first VM (the virus is only executed on one machine and is then allowed to propagate from that machine). The virus specified must exist in a disk image we have of many viruses. The virus is copied onto a disk image (share.img), each VM has access to (it appears as a standard windows drive to them).

The script also prompts the user to specify the time before the virus is executed and the time the experiment should continue after, this information is saved to a text file in the share.img image, so that our windows software (protector.exe) can determine when the virus should be executed.

The Virtual NetManager program has to be used via its graphical interface. We modified this program so it could instead be used via the command line, so that we could start the virtual network and the VMs with no user intervention.

The experiment ends when protector.exe on the first VM writes TERMINATENOW, to its virtual serial port. The virtual serial ports write their data to files on our physical machine.

When our BASH script **endCheck** finds this line of text, it calls the **stopVirtualNetwork** script. This script copies all the perfmon data written by each VM as well as the information detailing which viruses were executed on each machine, into a folder created for this experiment (named after the the virus and the time the experiment was performed).

A single DHCP (Dynamic Host Configuration Protocol) server was used to allocate IP addresses to the machines. As DHCP makes use of broadcast messages, it doesn't matter which switch it is connected to, as the broadcast messages are propagated across each switch and are received by each virtual machine.

The dhcp3 daemon was installed on my actual PC and configured, to give IP addresses in the range 10.0.0.* .

The following shows a relevant snippet from the configuration file for the DHCP server:

```
ddns-update-style none;
option domain-name "example.org";
option domain-name-servers ns1.example.org, ns2.example.org;
default-lease-time 600;
max-lease-time 7200;
log-facility local7;

subnet 10.0.0.0 netmask 255.255.255.0 {
  range 10.0.0.2 10.0.0.254;
}
```

As the DHCP server is running on the bare-metal (this means on the actual processor rather than in a virtual machine), it was necessary to create a virtual network interface, for it to interact with the virtual machines.

```
tunctl -t tap0
/etc/init.d/dhcp3-server restart
```

For the experiments where we wanted the virtual machines to simulate real-world activity, such as downloading webpages, we needed to provided Internet access to the VMs. This needed to be provided in such a way, as to prevent viruses escaping into the wild, whilst still allowing access to webpages and SSH (Secure Shell).

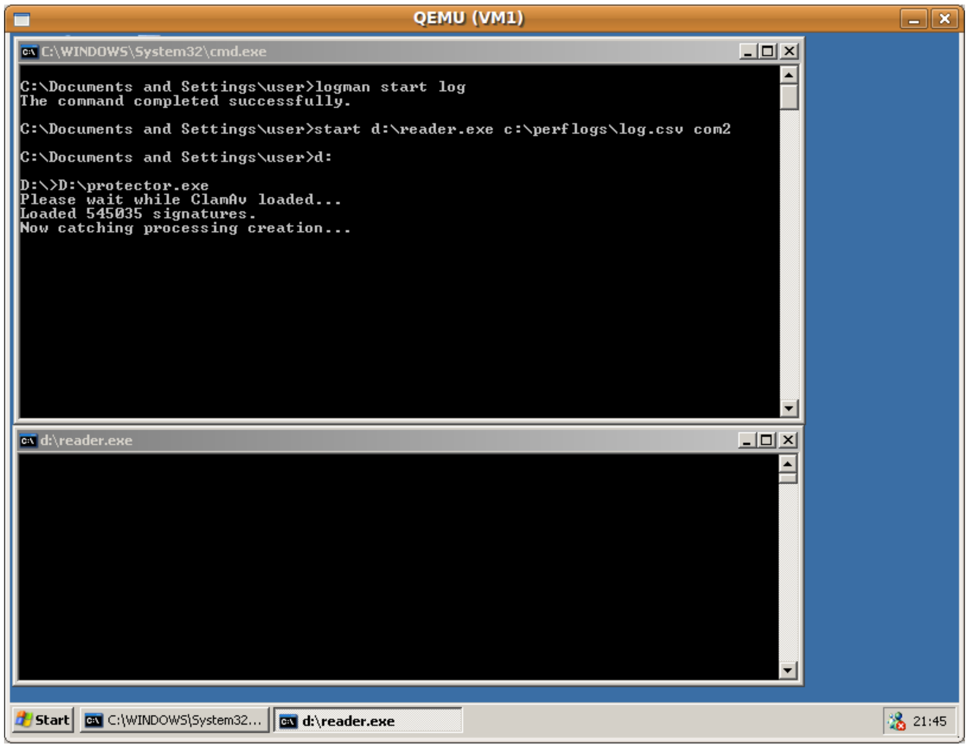
To accomplish this we used iptables, which is a program to allow control over Linux's firewall, to allow us to forward packets from the VMs out onto the Internet. Using iptables we were able to allow packets out only on the following ports: DNS (Domain Name System), HTTP (used for webpage transfer) and SSH (used in this case to upload files).

4.2 Virus observatory

For the virus observatory, it was necessary to see the propagation of a virus to different machines. To do this a standard anti-virus program can't be used, as they will quarantine the virus upon detection meaning we would be unable to see the virus spread.

The open source anti-virus scanner ClamAV was chosen, as it uses a simple dynamic link library, which can be utilised to scan a file and determine if there is a virus. ClamAV itself doesn't support on-access scanning (where a file is scanned before a process is allowed to execute), to provide this functionality, I created a program based on http://www.codeproject.com/KB/system/soviet_protector.aspx using Visual Studio in C.

This source code consists of a kernel mode driver, which monitors process execution, and a user mode program, which is used to deny/allow execution of a program. This code was modified so that when a new program was about to be run, it would first be scanned using ClamAV. Information on new process executions, was then written to a serial port, this includes the name of the virus if one is found by ClamAV.



```
C:\Documents and Settings\user>logman start log
The command completed successfully.

C:\Documents and Settings\user>start d:\reader.exe c:\perflogs\log.csv com2

C:\Documents and Settings\user>d:

D:\>D:\protector.exe
Please wait while ClamAv loaded...
Loaded 545035 signatures.
Now catching processing creation...
```

Figure 12: Virtual machine

This information is also used by our machine learning programs, in order to generate the class for the system activity in the training sets. As ClamAV tells us which virus is active on the computer and we know the time the virus was found, we are able to determine what class each instance of perfmon data should be given (as each instance of perfmon data also contains a time it was recorded).

Figure 12 depicts a virtual machine, it shows it running two programs we made. The reader.exe program logs the system counters to a virtual serial port, so that we are able to perform machine learning after the experiment has finished. The protector.exe program logs information on the processes which are executed on the machine and the name of the virus they are infected with, if any.

The observatory itself, consists of a graph, which represents the switches and machines on the network. This is created based on the topology of the network designed using the Virtual NetManager program. Figure 13 depicts the observatory running the Nanspy virus.

The observatory was written using Java, making use of JFreeChart in order to produce a chart depicting the number of viruses at a specific time. The JUNG library was used to

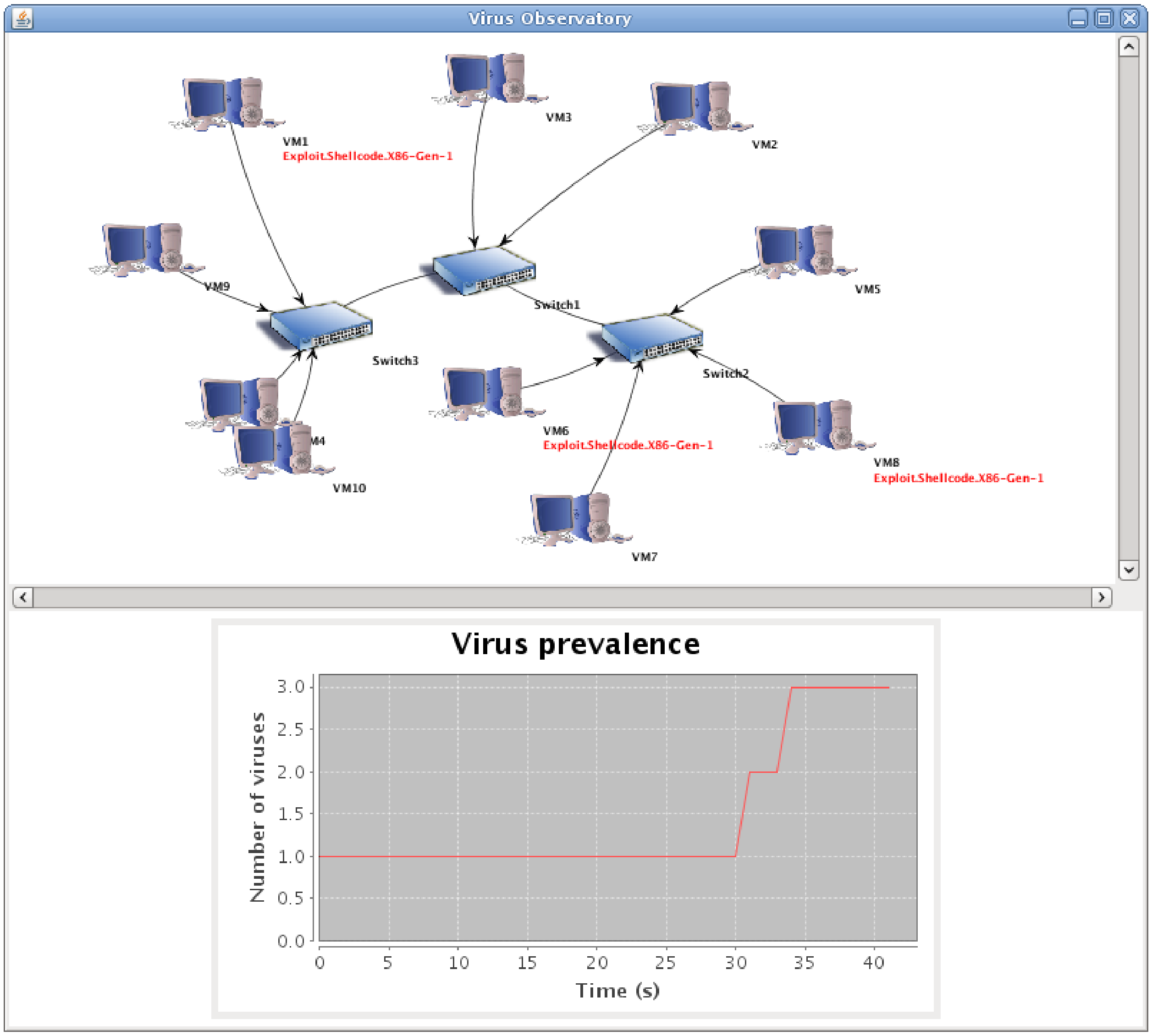


Figure 13: Virus Observatory

enable the creation of a graph of the network topology.

The observatory can be used live, when the virus is actually running on the virtual machines. It can also be used after the experiment has finished, to allow you to replay the effect of the virus propagation, it does this in real-time, simulating the same propagation times.

A thread is created for each virtual machine, to monitor the output from its serial port. If a virus is detected, the name of the virus is displayed next to the machine it was found on.

4.3 Feature visualisation

Figure 14 depicts a visualisation of many different system counters on six virtual machines, represented in a matrix fashion. Again this program was developed using Java.

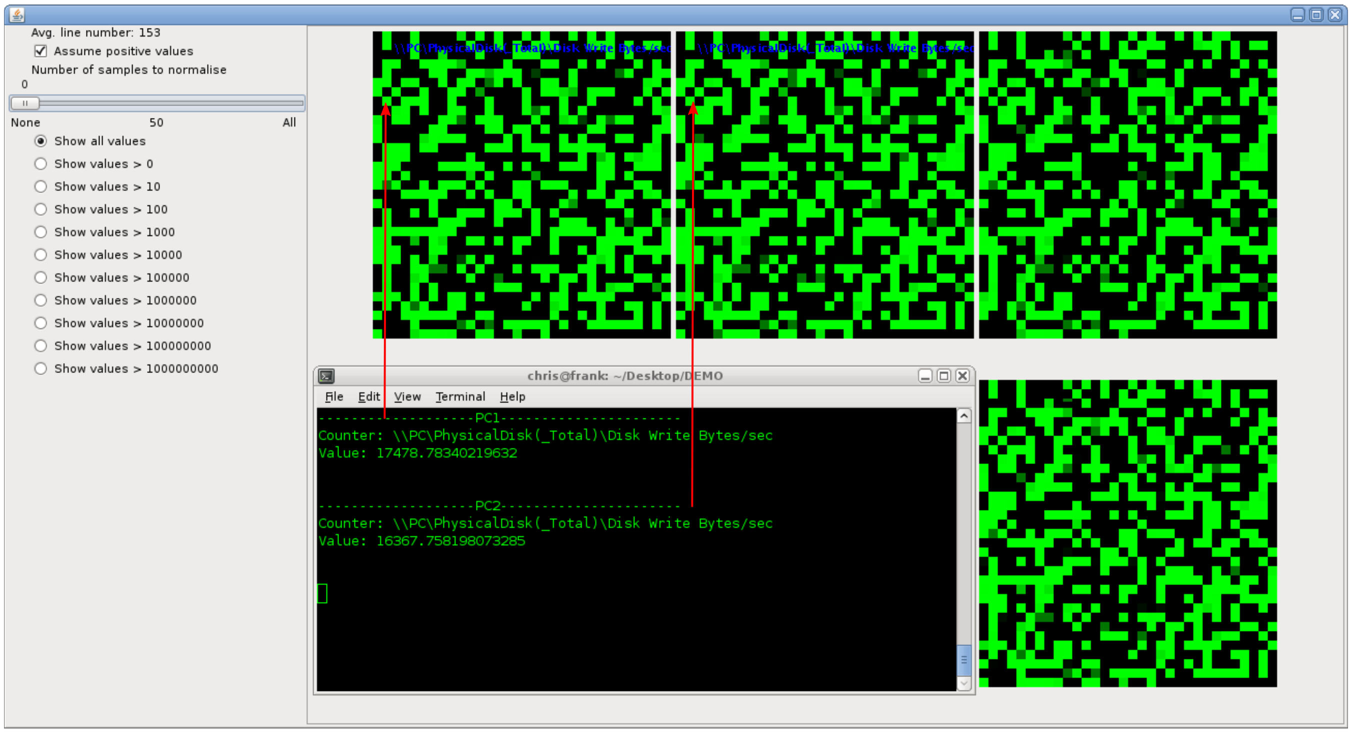
Each square in the matrix represents a different system counter. In order to compare values from different counters against each other, to see how they change, the system counter's values can be normalised.

A slider component (which allows the user to drag a node to a desired value) is used to vary the number of samples used in the normalisation process.

As the visualisation depicts real-time activity from each system, normalising with all samples will result in a slow-down of performance of the program as it runs which is why normalizing with around 25 samples more useful.

After the normalisation stage the value is passed into a sigmoid function, which generates values between 0 to 1. This is necessary as the values need to be mapped on to a range of 256 different colours (different shades of green).

Figure 14 depicts an example where no normalisation has been used. There is therefore less subtly in the values produced, the values are mostly completely green or black. This makes it fairly easy to see patterns between different machines.


Avg. line number: 153

[✓] Assume positive values

Number of samples to normalise

0

None 50 All

[● Show all values]
○ Show values > 0
○ Show values > 10
○ Show values > 100
○ Show values > 1000
○ Show values > 10000
○ Show values > 100000
○ Show values > 1000000
○ Show values > 10000000
○ Show values > 100000000
○ Show values > 1000000000

```
------------------PC1--------------------
Counter: \\PC\PhysicalDisk(_Total)\Disk Write Bytes/sec
Value: 17478.78340219632

------------------PC2--------------------
Counter: \\PC\PhysicalDisk(_Total)\Disk Write Bytes/sec
Value: 16367.758198073285
```

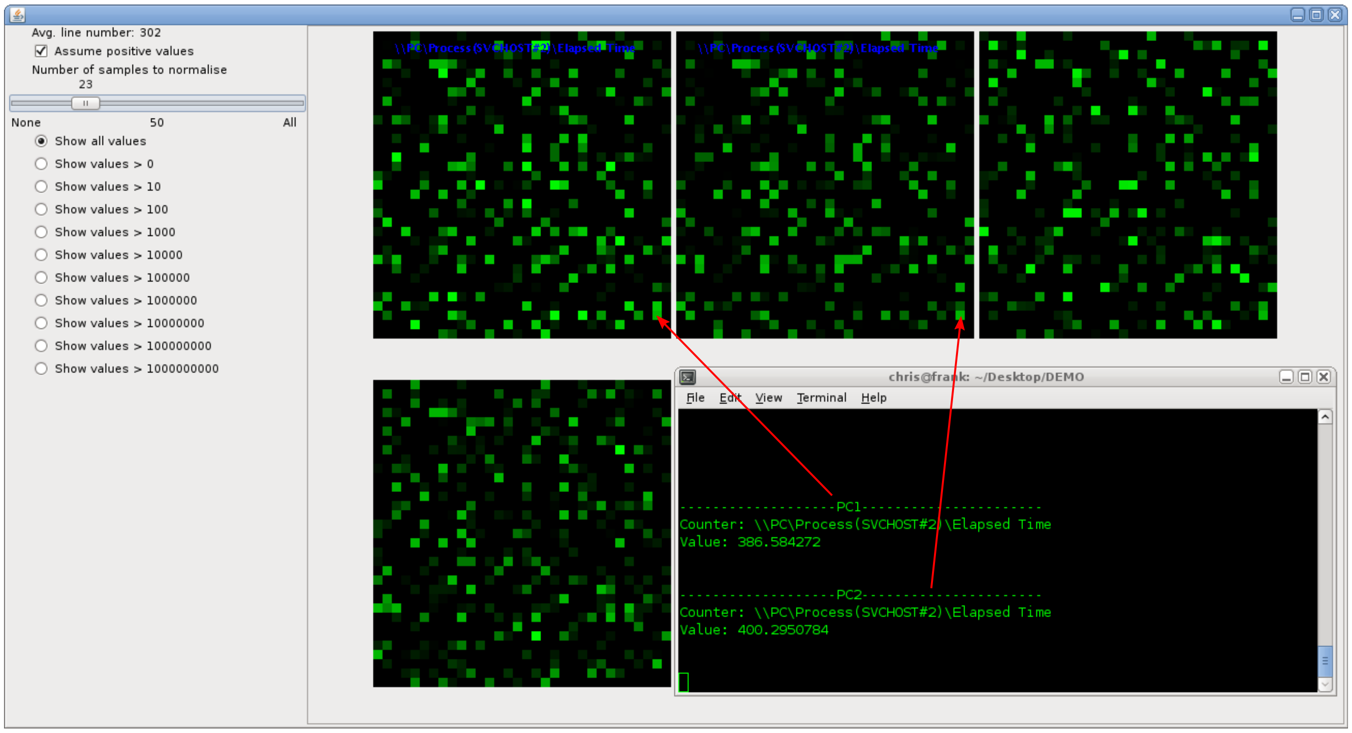
\\PC\PhysicalDisk(_Total)\Disk Write Bytes/sec
\\PC\PhysicalDisk(_Total)\Disk Write Bytes/sec
chris@frank: ~/Desktop/DEMO
File Edit View Terminal Help

Figure 14: Visualisation of features from 6 VMs depicting no normalisation

For example in figure 14 you can see that the values of the same counter on different machines would be reasonably similar just by looking at the matrix. After clicking on the counter on each machine to display the actual counter value, this is validated.

Figure 15 depicts an example were normalisation has been used. The values appear to change more noticeably with normalisation.

Like the virus observatory, the feature visualisation can either run from live data or from pre-recorded data.


\\PC\Process(SVCHOST#2)\Elapsed Time
\\PC\Process(SVCHOST#2)\Elapsed Time
chris@frank: ~/Desktop/DEMO
File Edit View Terminal Help

Avg. line number: 302

[✓] Assume positive values

Number of samples to normalise
23

None 50 All

[● Show all values]
Show values > 0
Show values > 10
Show values > 100
Show values > 1000
Show values > 10000
Show values > 100000
Show values > 1000000
Show values > 10000000
Show values > 100000000
Show values > 1000000000

```
-------------------PC1----------------------
Counter: \\PC\Process(SVCHOST#2)\Elapsed Time
Value: 386.584272

-------------------PC2----------------------
Counter: \\PC\Process(SVCHOST#2)\Elapsed Time
Value: 400.2950784
```

Figure 15: Visualisation of features from 6 VMs using normalisation

4.4 Collecting viruses

In order to collect more viruses, a virus 'honeypot' was setup. We used Amazon's web services to accomplish this. Amazon's ec2 service, enables us to set up a virtual server and easily shut it down whenever we need to.

On this virtual server we ran our WindowsXP virtual machine. The virtual machine was given plenty of memory (800 megabytes).

The VM has a virtual interface tap0, through which its packets are sent or received. Additionally all packets flowing through this interface were captured using tcpdump to a file, so that the infection vector of viruses could be observed.

The following shows how the VM was initialised:

```
#with vnc
nohup qemu -vnc :0 -net nic -net tap -daemonize -hda `pwd`/winxp.img -m 800

# ip of host VM
route add -host 192.168.1.10 dev tap0

# need to give tap ip
ifconfig tap0 192.168.1.11

# must run firewall stuff AFTER we setup routes
./fire-all

# setup local forward for remote desktop
# so, that i can tunnel in via ssh, to rdesktop VM

nohup tcpdump -i tap0 -nnvvXSs 65535 -w packets_`date +%d_%m_%Y`.pcap 2>/dev/null &
```

The Amazon ec2 instance, needs to forward packets addressed to its open ports to the the VM; to do this a special iptables script was written.

The only ports which aren't forwarded are port 22 and 5900 (as well as ports less than 22, as there aren't any open ports in this range on the WindowsXP VM). Port 22 is used by SSH, a secure shell, used to communicate with the ec2 instance to control it.

Port 5900 provides VNC access to the WindowsXP VM, we didn't want this port open to the general public as it would give people who found it, complete access to the VM.

So port 5900 was left closed, to control the VM we needed to set up an SSH tunnel to the VNC port, meaning it could only be accessed through an encrypted tunnel via port 22.

The following is the iptables script used:

```
iptables -F
iptables -X
iptables -t nat -F
iptables -t nat -X
iptables -t mangle -F
iptables -t mangle -X
modprobe ip_conntrack
modprobe ip_conntrack_ftp

# Setting default filter policy
iptables -P INPUT ACCEPT
iptables -P OUTPUT ACCEPT

echo 1 >  /proc/sys/net/ipv4/ip_forward

/sbin/iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
/sbin/iptables -A FORWARD -i eth0 -o tap0 -m state
        --state RELATED,ESTABLISHED -j ACCEPT
/sbin/iptables -A FORWARD -i tap0 -o eth0 -j ACCEPT

# most
iptables -t nat -A PREROUTING -p tcp --dport 23:5899 -j DNAT
        --to-destination 192.168.1.10

iptables -A FORWARD -s 192.168.1.10 -p tcp --dport 23:5899 -j ACCEPT

iptables -t nat -A PREROUTING -p udp --dport 23:5899 -j DNAT
        --to-destination 192.168.1.10
iptables -A FORWARD -s 192.168.1.10 -p udp --dport 23:5899 -j ACCEPT
iptables -t nat -A PREROUTING -p tcp --dport 5901:65535 -j DNAT
        --to-destination 192.168.1.10
iptables -A FORWARD -s 192.168.1.10 -p tcp --dport 5901:65535 -j ACCEPT

iptables -t nat -A PREROUTING -p udp --dport 5901:65535 -j DNAT
        --to-destination 192.168.1.10
iptables -A FORWARD -s 192.168.1.10 -p udp --dport 5901:65535 -j ACCEPT
```

Using nmap to scan the windows VM through a terminal on the ec2 instance, I found the following ports were open.

```
root@ip-10-226-231-244:~/winxp# nmap 192.168.1.10

Starting Nmap 4.62 ( http://nmap.org ) at 2009-08-29 13:22 UTC
Interesting ports on ip-192-168-1-10.eu-west-1.compute.internal (192.168.1.10):
Not shown: 1710 closed ports
PORT     STATE SERVICE
135/tcp  open  msrpc
139/tcp  open  netbios-ssn
445/tcp  open  microsoft-ds
1025/tcp open  NFS-or-IIS
3389/tcp open  ms-term-serv
MAC Address: 52:54:00:12:34:56 (QEMU Virtual NIC)

Nmap done: 1 IP address (1 host up) scanned in 0.998 seconds
```

These same ports were also visible externally to the Internet, to allow viruses to infect the machine.

We left the windows VM running for 20 hours in order to give the machine enough time to be able to catch viruses.

We were able to examine the network captures I obtained using Snort, a program which is used to detect network intrusions using rules.

The following shows the alerts generated by Snort, from the tcpdump files we generated:

```
[**] [1:1444:3] E3[rb] TFTP GET from external source [**]
[**] [1:2000355:4] E4[rb] ET POLICY IRC authorization message [**]
[**] [1:2000427:9] E3[rb] ET POLICY PE EXE Install Windows file download [**]
[**] [1:2002750:10] ET POLICY Reserved IP Space Traffic - Bogon Nets 2 [**]
[**] [1:2003:12] SQL Worm propagation attempt [**]
[**] [1:2004:11] SQL Worm propagation attempt OUTBOUND [**]
[**] [1:2008120:1] E3[rb] ET POLICY Outbound TFTP Read Request [**]
[**] [1:2050:14] SQL version overflow attempt [**]
[**] [1:22000032:6] E2[rb] BLEEDING-EDGE EXPLOIT LSA exploit [**]
[**] [1:22466:7] E2[rb] NETBIOS SMB-DS IPC$ unicode share access [**]
[**] [1:2404011:1142] E4[rb] ET DROP Known Bot C&C Server Traffic (group 12)  [**]
[**] [1:2404013:1142] E4[rb] ET DROP Known Bot C&C Server Traffic (group 14)  [**]
[**] [1:2508:13] NETBIOS DCERPC NCACN-IP-TCP lsass DsRolerUpgradeDownlevelServer
    overflow attempt [**]
[**] [1:292000032:99] E2[rb] BotHunter EXPLOIT LSA exploit [**]
[**] [1:299913:1] E2[rb] SHELLCODE x86 0x90 unicode NOOP [**]
[**] [1:3001441:1] E3[rb] TFTP GET .exe from external source [**]
[**] [1:31000004:99] E3[rb] BotHunter Scrip-based Windows egg download .exe [**]
[**] [1:3397:8] NETBIOS DCERPC NCACN-IP-TCP ISystemActivator RemoteCreateInstance
    attempt [**]
[**] [1:3409:7] NETBIOS DCERPC NCACN-IP-TCP IActivation remoteactivation overflow
    attempt [**]
[**] [1:5001684:99] E3[rb] BotHunter Malware Windows executable (PE) sent from
    remote host [**]
```

```
[**] [1:52000032:6] E5[rb] BLEEDING-EDGE EXPLOIT LSA exploit [**]
[**] [1:52466:7] E5[rb] NETBIOS SMB-DS IPC$ unicode share access [**]
[**] [1:592000032:99] E5[rb] BotHunter EXPLOIT LSA exploit [**]
[**] [1:599913:1] E5[rb] SHELLCODE x86 0x90 unicode NOOP [**]
[**] [1:7209:10] NETBIOS DCERPC NCACN-IP-TCP srvsvc NetrPathCanonicalize overflow
    attempt [**]
```

Snort makes use of simple rules in order to detect malicious activity, they are similar to regular expressions.

The following snippet depicts one of the Snort rules which detected malicious activity:

```
mal1.rules:alert tcp $EXTERNAL_NET !20 -> $HOME_NET any (msg:"E3[rb] BotHunter
    Malware Windows executable (PE) sent from remote host";  content: "MZ"; content:
     "PE|00 00|"; within:250; flow: established; sid:5001684; rev:99;)
```

The rule looks for the byte sequence "MZ" and "PE|00 00|" in the content of each packet and will generate an alert provided the sequences are within 250 bytes of each other.

In order to extract the viruses from the VM, I first killed the VM. The viruses were contained inside the VM's disk image. To extract them, I mounted the disk image, on the ec2 instance and used the FPROT [8] anti-virus scanner to find the viruses.

According to FPROT for linux the following viruses were found:

```
[Found downloader] <W32/Bloop.A.gen!Eldorado (generic, not disinfectable)>
[Found security risk] <W32/Backdoor2.FEHI (exact)>
[Found trojan] <W32/Trojan5.DCW (exact)>
```

Unfortunately when the viruses were tested in the virus observatory they didn't spread. There are a number of possible reasons for this. The virus may use techniques to determine if it is inside a VM, for instance checking the MAC address of network adapters, to see if the address is owned by a virtual machine software company.

The virus may also refuse to spread if there is no active Internet connection, a common way in which they can detect this is by attempting to resolve a popular domain name such as microsoft.com, if this fails the virus will refuse to spread.

The reason viruses employ such techniques is purely to make it as hard as possible for researchers to understand how it functions.

4.5 Activity generator

In order to generate system activity that would reasonably accurately represent real world computer usage, we designed a program which would spawn various programs such as Internet Explorer, paint, calculator and upload files at random intervals. These programs have Internet access to allow them to actually download webpages or upload files via SFTP (a secure file transfer protocol).

It is important this program should be run throughout the whole experiment, as it is important when the virus is launched, normal computer usage continues as well.

[8] http://www.f-prot.com/

If this activity generator was only used before the virus execution, the virus may be artificially more easily detected by the machine learning classifiers because the virus would be one of the main programs running on the operating system.

4.6 Weka

Weka was chosen as the basis of the machine learning part of the virus detection system due to the fact it provides a large number of machine learning classifiers as well making it easy to write your own classifier to integrate within their framework.

In order for Weka to be able to load the Perfmon data it first needed to be converted into Weka's arff format. To do this a Java program was written (Tool.java) this program processes data obtained from each virtual machine. One file from a VM contains the perfmon data, the other file includes information on which processes where executed and whether they were infected by a virus.

The following arff file shows a simplified version of one of the arff files our program generates:

```
@relation virus
@attribute      arp_frames_per_min      numeric
@attribute      class                   {yes,no}
@data
1,      no
2,      no
1,      no
1,      no
3,      no
20,     yes
14,     yes
9,      yes
```

In this example, the class represents whether the system was infected, when that data was recorded.

Tool.java generates two arff files for each machine, provided a virus was detected on the machine. The first arff file has a name in the form weka*.arff, in it each instance has two possible class values, yes or no, indicating whether there was a virus present when the instance was recorded (just like in the above example).

The second arff file for a machine has a name in the form weka*-vrec.arff, it has the name of viruses or 'no' (indicating normal activity) for its class values. This arff file is used by machine learning techniques to detect specific viruses, rather than just virus activity.

The arff files generated have around 1070 attributes. The attributes were examined to ensure that any attributes which may aid the detection of the virus by relying on experimental procedures were removed. This task is performed by the Experiment class, after the training and test instances have been loaded from the arff files.

Attribute names containing the words "LogicalDisk" or "PhysicalDisk" were removed, as they monitor the amount of free disk space on a drive. As the virus is launched 15 or 30 minutes after the experiment has started, the amount of free disk space will decrease during this time, as Perfmon counters are being logged to disk, taking up space. The machine learning classifier could possibly train itself to detect a virus based on the amount of free disk space rather than real virus activity, we therefore removed this possibility.

As the main aim of the project is to investigate the application of machine learning techniques to virus detection we now consider how the machine learning package Weka can be utilised for this goal.

With Weka different classifiers can easily by explored. Weka is able to measure the performance of each classifier in terms of the number of correctly classified instances.

One particularly important classifier Weka provides is the naive bayes classifier. This would need to be trained with activity before and after the system is infected with a virus.

4.6.1 Novelty classifier

Our novelty classifier makes use of the 'parzen window' as explained in the literature review. It has one customisable parameter, the sigma value, which alters the Gaussian kernel to include more or less data inside a cluster.

The novelty classifier deletes all instances in the training set that use the 'yes' class (the data representing activity when the virus was present), as the the novelty classifier only requires 'normal' system activity.

The data from the parzen window has a threshold applied to it. If the value is more than zero the instance is classified as being 'normal' otherwise it is classified as 'anomalous'.

4.6.2 Deleting lowest ranking attributes

This WekaDeleteAttribute class is used to rank attributes according to a user specified ranking class (this is generally information gain). This class accepts a number representing the number of attributes to keep, these attributes to keep come from the highest ranked ones.

This class is used with the training set, the most important attributes can then be kept. The same attributes in the test set are then also kept and all other attributes deleted.

Surprisingly by deleting attributes the performance of the classifiers can actually be improved. This is explained in more detail later on.

4.6.3 Experiment

This general class is used to accept training & test instances and a specific classifier. The classifier is then trained and tested. The number of correctly classified instances can then be easily found from this class. It also generates a ROC curve for each experiment to see visually the performance of the classifier.

I modified Weka's ROC curve generator so that it wouldn't plot cross-hairs on the line, as it made it difficult to understand. Also I made it save a png image of the ROC curve to a file relating to the name of the experiment.

I discussed the Weka ROC curve generation algorithm on the Weka mailing list and found that when a large number of attributes were used with the naive bayes algorithm the ROC curve would appear very jagged. I managed to fix this by drammatically reducing the number of attributes used by deleting low ranking attributes.

4.6.4 Launch experiments

Our WekaTool class is used to spawn experiments. The experiments then generate ROC (Receiver operating characteristic) curves along with information relating to how accurate the classifier is.

Latex code is generated automatically to a tex file, relating to the experiment name, to provide information on the performance of the experiment.

The genNoveltyGraph function is used to generate a graph using GNUplot [9], which trains our novelty classifier for a range of sigma values and then tests it. This allows us to see visually which sigma values give the best percentage of correctly classified instances.

4.7 Windows setup

The virtual machines run a simple Windows batch script upon start-up:

```
logman start log

start d:\activesim.exe
start d:\reader.exe c:\perflogs\log.csv com2

d:
d:\protector.exe
```

The script initiates the logging of system counters.

It spawns reader.exe our program to dump the output generated by windows perfmon, to the VMs virtual serial port.

Protector.exe is also run at the end. As previously mentioned this is based on [10]. Each VM determines whether it is the master VM (the one which launches the virus and ends the experiment), by comparing its MAC address to a hard coded value of the master's MAC address. The ClamAV library is used to scan new processes for infection. We write the name of the process executed along with the name of the virus if there was one, to a virtual serial port.

[9] http://www.gnuplot.info/
[10] http://www.codeproject.com/KB/system/soviet_protector.aspx

5 Analysis

5.1 Overview

All tests were performed using a simple network topology consisting of 10 virtual machines and 3 switches. We used a PC with the Intel Q6600 quad-core processor [11] along with 4GB of RAM to run these. Admittedly 10 machines and 3 switches not a vast number of devices but for the purpose of these experiments, we believe it to be adequate. Running more would have also resulted in degraded performance for the VMs.

The purpose of the experiments were to allow the infection of machines whilst giving us access to various system information along with details on which machines were actually infected.

We performed a number of different scenarios to assess the performance of machine learning techniques under different conditions.

5.1.1 Scenario 1

During the first 15 minutes of each experiment all virtual machines were idle except for activity such as obtaining IP addresses and running our logging programs.

The virus was then executed on the first VM (the machine we dedicated as master, as it runs the virus and ends the experiment). The machines were then left for another 15 minutes to give the virus chance to propagate.

We ran an experiment for each of the viruses we obtained: Net-Worm.Win32.Nanspy.b, Net-Worm.Win32.Padobot.f and Net-Worm.Win32.Sasser.gen.

We then ran these experiments again, so that we would have data to be able to test our machine learning classifiers with.

We could have avoided generating test data, if we made use of cross-validation. This technique usually requires ten folds.

The order of the instances in the training set are first randomized. Then the training set is split into ten 'folds' or partitions. The first fold is used as the test set and the rest used as the training set, the machine learning classier is then trained and tested using this data. Then the second fold is used as the test set and the rest as the training set, this process continues to the tenth fold (see figure 16).

The percentage of correctly classified instances from each of these tests can then be averaged. The problem with 10 fold cross-validation is that the average percentage of correctly classified instances is generally higher than when testing on real test data, as the same training data is essentially being used many times. Cross-fold validation is more useful were it is very difficult to perform many experiments. As our system allows us to easily perform multiple experiments, we decided against using 10 fold cross-validation.

[11] http://www.intel.com/products/processor/core2quad/specifications.htm

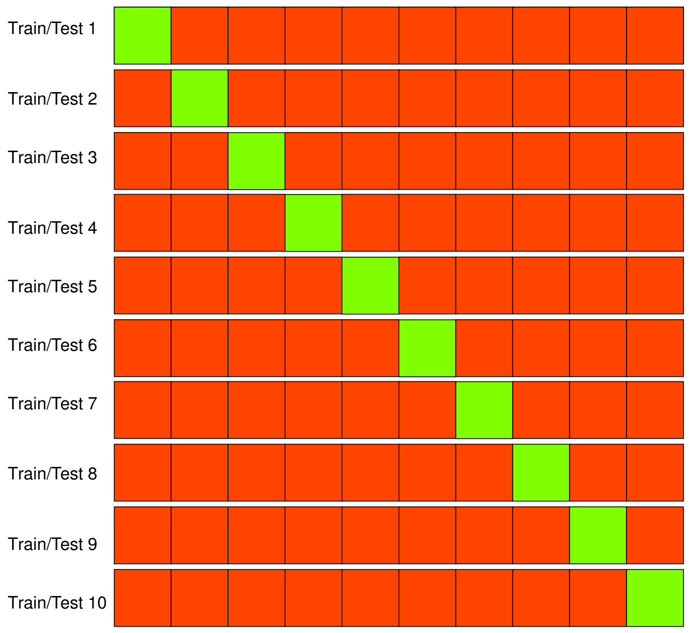


Figure 16: Ten fold cross-validation. The green squares represent the test set and the red squares, the training set

5.1.2 Scenario 2

In this scenario we performed the 3 experiments in Scenario 1 again, with the virus again being executed after 15 minutes and then waiting a further 15 minutes before the experiments ended.

We then performed the 3 experiments again to generate data which would be used for testing the classifiers.

During all these experiments though we made use of simulated real-world activity using our activity generator, to perform actions such as loading random webpages through Internet Explorer, uploading a file and opening various programs.

5.1.3 Scenario 3

We performed experiments with each of the 3 virues were the virus was executed on the first VM after 30 minutes rather than 15 minutes and then gave the experiment another 30 minutes to allow the virus to propagate.

All this experiment data was then used as a test set. With one set of three experiments from Scenario 1 being used as the training set.

5.2 Attribute Selection

In order to select important attributes, information gain can be used as a ranking method to determine useful attributes from a training set. As information gain requires discrete values, Weka performs discretization on instances to convert numeric to nominal values, see [3].

The following table shows the top ten attributes ordered by information gain for the Net-Worm.Win32.Padobot.f virus (when using absolute perfmon values and using simulated real-world activity):

Attribute	**Information gain**
\\PC\Objects\Threads	0.9922
\\PC\Process(SVCHOST#3)\Working Set	0.9917
\\PC\Process(SVCHOST#3)\Working Set Peak	0.9917
\\PC\Terminal Services Session(Console)\Handle Count	0.9900
\\PC\Process(System)\Handle Count	0.9897
\\PC\Process(Total)\Handle Count	0.9894
\\PC\Process(Total)\Thread Count	0.9891
\\PC\System\Threads	0.9891
\\PC\Terminal Services Session(Console)\Thread Count	0.9891
\\PC\Objects\Events	0.9890

The above table shows attributes which all have very similar information gain values, the problem with this, is there is not much difference to distinguish each attribute. The values also appear to be unreasonably high, this is likely due to the fact that absolute values from counters are being used.

The highest ranked attribute in this list is \\PC\Objects\Threads. The description given about this counter by perfmon is "the number of threads in the computer at the time of data collection". It makes sense that this could be the most useful attribute when using absolute values, as when the virus is executed there will by at least one more thread active, than during the period of normal activity.

We found a solution to the problem of absolute values not proving satisfactory by using gradients instead. Figure 17 depicts how we apply a gradient to the perfmon instances, by taking two values from a counter around 20 seconds apart, then using this simple formulae:

$$gradient = \frac{(\text{current counter value} - \text{previous counter value})}{interval} \tag{7}$$

The following table depicts information gain of instances were a gradient had been applied (again using simulated real-world activity):

Attribute	**Information gain**
\\PC\TCP\Connections Active	0.9763
\\PC\TCP\Connection Failures	0.9747
\\PC\Network Interface(Realtek RTL8139...)\Packets Received Non Unicast/sec	0.6682
\\PC\TCP\Segments Retransmitted/sec	0.6669
\\PC\Network Interface(Realtek RTL8139...)\Packets Sent Unicast/sec	0.4330
\\PC\Network Interface(Realtek RTL8139...)\Packets Sent/sec	0.4326
\\PC\IP\Datagrams Sent/sec	0.4023
\\PC\Process(System)\IO Other Operations/sec	0.3758
\\PC\Network Interface(Realtek RTL8139...)\Packets/sec	0.3457
\\PC\IP\Datagrams/sec	0.3183

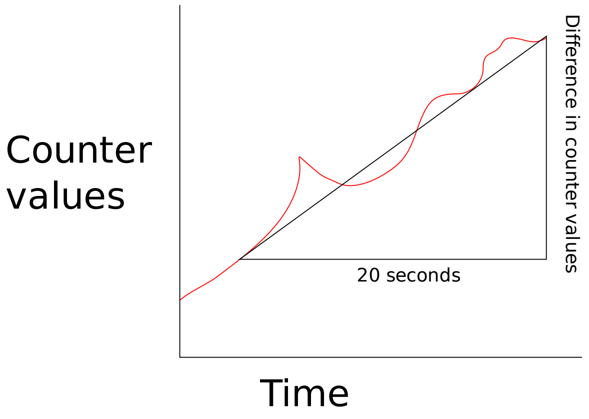


Figure 17: Applying a gradient to the counter values

The attributes ranked by information gain appear much more reasonable after applying a gradient, as all except one, are network based counters, which are especially useful in detecting the spread of worms.

The highest ranked attribute “\\PC\TCP\Connections Active” represents “the number of times TCP connections have made a direct transition to the SYN-SENT state from the CLOSED state” according to perfmon. The SYN-SENT state simply means the first stage of a TCP connection, were the client has sent a packet to another computer, indicating it wants to create a connection.

The rate of new TCP connections being created when a worm is present will generally be higher than under normal activity. Worms need to search for new computers to infect, which requires opening many TCP connections. According to Symantec, Padobot “attempts to exploit the LSASS Windows vulnerability on TCP port 445 (described in Microsoft Security Bulletin MS04-011), against random IP addresses.”, confirming this.

The number of connection failures is also very useful in detecting Padobot. Padobot causes connection failures by attempting connections to random IPs, many of those won’t be in use or their ports will be closed.

The following represents the top 10 information gain attributes for all the viruses we used, again using a gradient and simulating real-world activity.

Attribute	**Information gain**
\\PC\TCP\Connection Failures	0.6436
\\PC\TCP\Connections Active	0.6140
\\PC\TCP\Segments Retransmitted/sec	0.4356
\\PC\Network Interface(Realtek RTL8139...)\Packets Received Non Unicast/sec	0.2807
\\PC\Process(System)\IO Other Operations/sec	0.2785
\\PC\IP\Datagrams Sent/sec	0.2689
\\PC\Network Interface(Realtek RTL8139...)\Packets Sent Unicast/sec	0.2627
\\PC\Network Interface(Realtek RTL8139...)\Packets Sent/sec	0.2529
\\PC\Network Interface(Realtek RTL8139...)\Packets/sec	0.2096
\\PC\IP\Datagrams/sec	0.2089

Very interestingly these top 10 attributes for all the viruses we used are exactly the same as the top 10 for Padobot, although not in the same order. This gives us reason to believe that it may be possible to detect more than one specific virus, after a classifier has been trained with one virus.

5.3 Virus recognition

5.3.1 Using absolute values

In order to recognise specific viruses the naive bayes classifier was trained with Perfmon data from 3 viruses, as well as normal activity, meaning the following classes Net-Worm.Win32.Nanspy.b,Net-Worm.Win32.Padobot.f, Net-Worm.Win32.Sasser.gen,no were used (no represents normal activity).

The number of attributes were pruned to 50 of the most important as judged using information gain, using data from the training set.

The following table depicts the performance of virus recognition using perfmon data 15 minutes before/after the virus and tested on data 15 minutes before/after the virus. In this case the VMs were not simulating real-world activity throughout the experiment (**Scenario 1**):

Statistic	**Value**
% Correctly classified	99.8607
% Incorrectly classified	0.1393

We performed the experiment again, but simulated real-world activity throughout (**Scenario 2**).

Statistic	**Value**
% Correctly classified	95.5028
% Incorrectly classified	4.4972

Unsurprisingly when the system was experiencing more real-world like activity, the number of correctly classified instances was reduced, as such activity presents more ‘noise’ for the classifier to deal with.

In order to ensure that the system was not functioning soley based upon the fact the virus is launched 15 minutes into the experiment; we trained the naive bayes classifier with 15 minute experimental data, and tested on 30 minute experimental data (we didn’t use simulated real-world activity in this experiment)(**Scenario 3**).

Statistic	**Value**
% Correctly classified	79.3812
% Incorrectly classified	20.6188

The previous experiments all ran using absolute values from the Perfmon counters which were then fed straight into the naive bayes classifier. The problem with this approach is that different computers would have dramatically different orders of magnitude of values for their counters, due to different uptimes of computers and also different uses (e.g. server, home computer etc.).

5.3.2 Using gradients

The following results depict the effect of using gradients instead of absolute values, when no simulated real-world activity was used (using the same 15 minute before/after data as before and 50 attributes, again ranked using information gain)(**Scenario 1**).

Statistic	**Value**
% Correctly classified	93.1298
% Incorrectly classified	6.8702

The following depicts results when simulated real-world activity was used (**Scenario 2**).

Statistic	**Value**
% Correctly classified	91.8937
% Incorrectly classified	8.1063

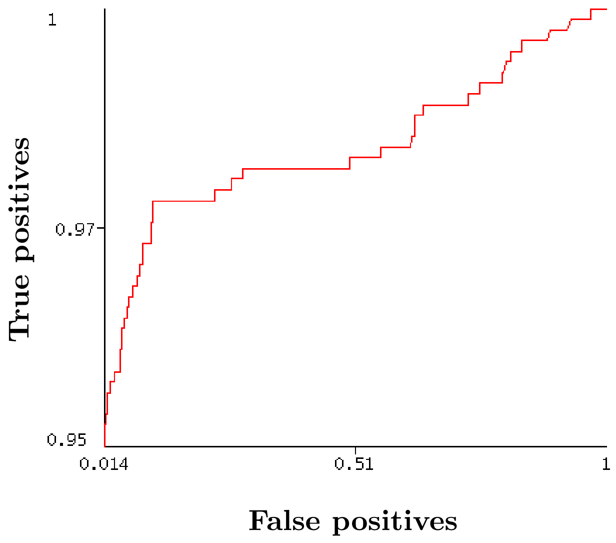
You can see that although the number of correctly classified instances is lower than with the absolute values experiments, it is still high. We found the optimal value of the difference in time between the two counter values for the gradient was 20 seconds, through a simple process of trial and error.

5.4 Detecting the presence of a virus

We now wanted to see if the accuracy of the system improves if we only want to know if a virus is present, rather than what the virus actually is. Meaning that instead of using a class which contains the names of all the different viruses we want to detect, we simply use binary class values: yes & no, which represent the presence or absence of a virus respectively.

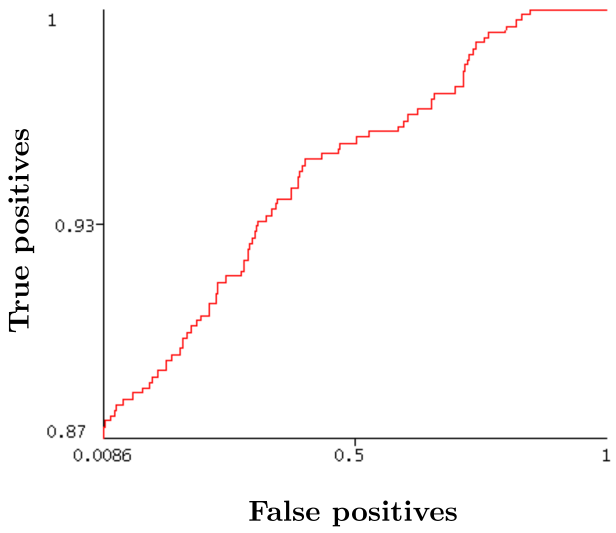
We use gradients from now on in all experiments, as we feel they give a more accurate picture of the accuracy of the system.

Both figure 18 and figure 19 show that we achieve a higher number of correctly classified instances, when using only the binary classes yes & no.



Statistic	Value
% Correctly classified	96.7803
% Incorrectly classified	3.2197
Area under ROC	0.9767

Figure 18: ROC curve generated using no simulated real-world activity (**Scenario 1**)



Statistic	Value
% Correctly classified	93.0293
% Incorrectly classified	6.9707
Area under ROC	0.9460

Figure 19: ROC curve generated using simulated real-world activity (**Scenario 2**)

5.5 Novelty detection

We found when all attributes were included, the novelty classifier wasn't able to achieve greater than a random classification of 50% in testing.

The following graph depicts the effect of varying the σ (sigma) parameter whilst using different numbers of attributes chosen using information gain (using 15 minute train & test and using simulated real-world activity):

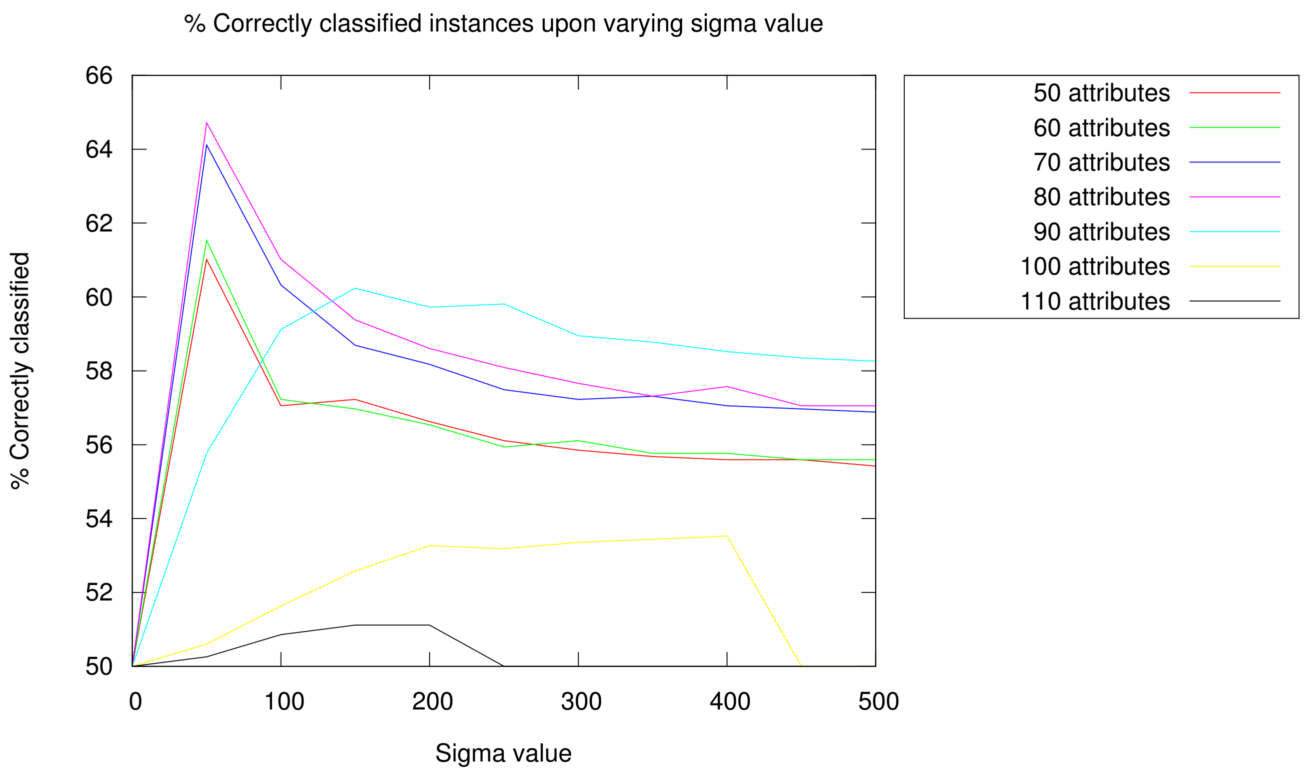


Figure 20: Novelty detection performance upon varying sigma (**Scenario 2**):

The greatest number of correctly classified instances was 64.72% obtained using 80 attributes and a sigma value of 50.

When not using simulated real-world activity as shown in figure 21 the highest percentage of correctly classified instances was 94.32% using 50 attributes.

We attempted to improve the accuracy of the classifier by first normalising the data and then applying a gradient. This however resulted in the classifier being unable to achieve a greater than random performance.

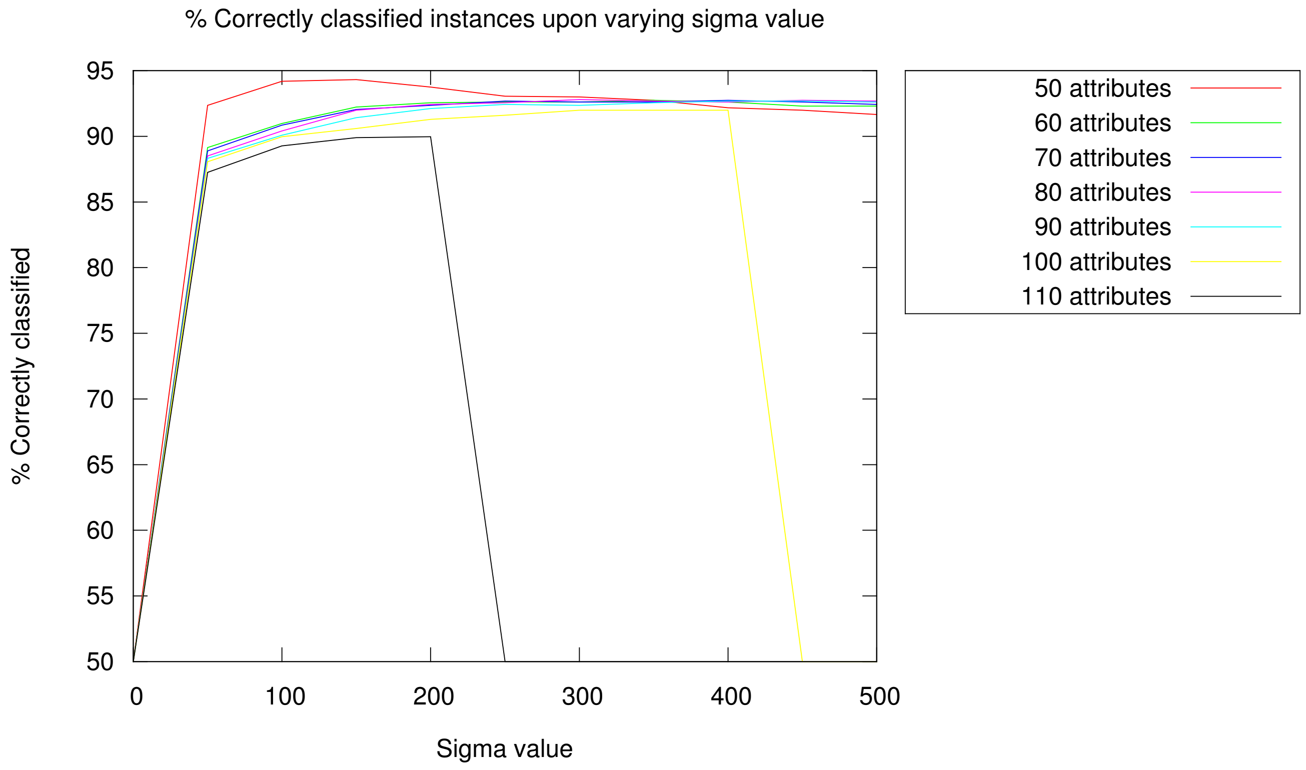


Figure 21: Novelty detection performance upon varying sigma not using simulated real-world activity (**Scenario 1**)

5.5.1 One Class Classifier

Additionally we wanted to test the performance of novelty detection using a different algorithm, referred to as OneClassClassifier [12].

The main advantage with this classifier over ours, is there is no need to tweak any specific parameters like the 'sigma' value, as it is capable of determining such parameters automatically.

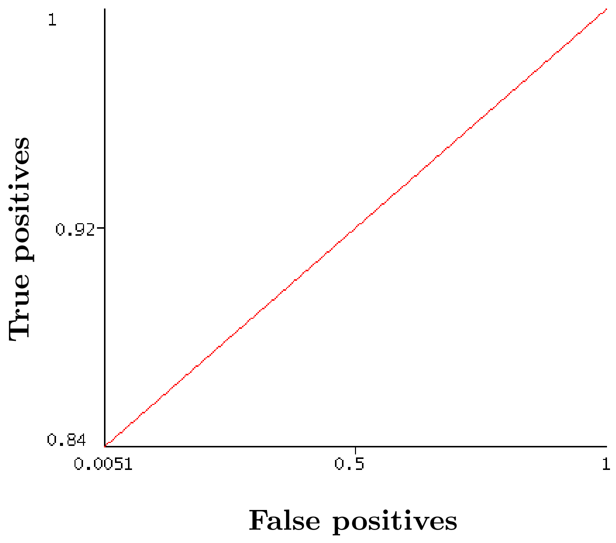
Unfortunately the source code of this classifier is not currently available, but it will be interesting to see its methodology, when it is released.

We performed novelty detection using the OneClassClassifier with 50 attributes and found the accuracy was substantially poorer when simulating real-world activity. This is likely due to the additional 'noise' that real-world like computer usage adds to the system.

Figure 22 represents the performance of the OneClassClassifier when not simulating real-world activity.

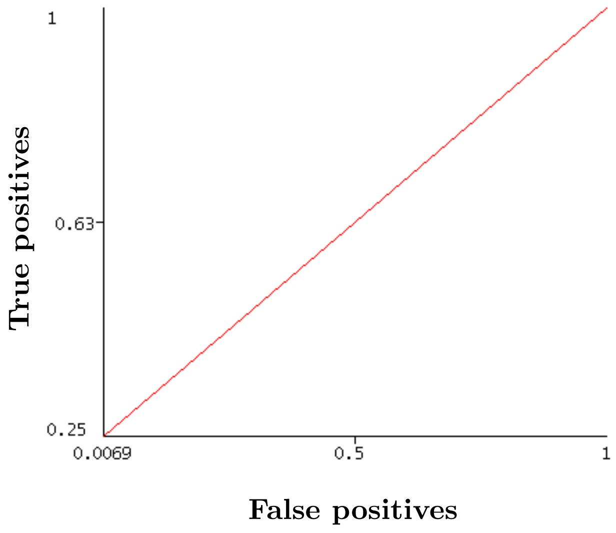
[12]OneClassClassifier - `http://www.cs.waikato.ac.nz/~kah18/occ/`

Figure 23 represents the performance of the OneClassClassifier while simulating real-world activity, which shows substantially poorer results. At 62.22% of correctly classified instances this is slightly higher than the 61.02% of correctly classified instances using the parzen window for 50 attributes.



Statistic	**Value**
% Correctly classified	91.6667
% Incorrectly classified	8.3333
Area under ROC	0.9167

Figure 22: ROC curve generated without simulated real-world activity (**Scenario 1**)



Statistic	Value
% Correctly classified	62.2203
% Incorrectly classified	37.7797
Area under ROC	0.6222

Figure 23: ROC curve generated with simulated real-world activity (**Scenario 2**)

We were able to dramatically improve the performance of the novelty classifier for detecting a virus during the use of simulated real-world activity, by enabling the densityOnly option of the classifier which "sets whether to exclusively use the density estimate P(X|A)". After which, we obtained the following results:

Statistic	Value
% Correctly classified	90.0172
% Incorrectly classified	9.9828
Area under ROC	0.9205

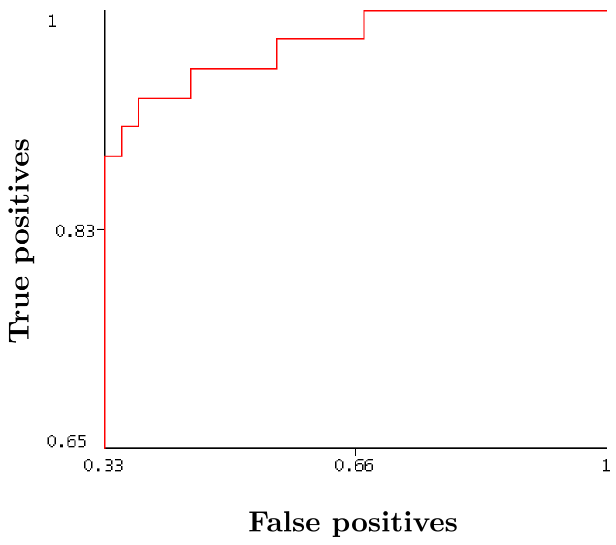
5.5.2 Novelty detection in a distributed fashion

We were curious to see if the novelty classifier could be used in a distributed fashion using the following technique:

Use perfmon log data from all machines in an experiment, apart from data from the machine on which we want to determine whether it has a virus or not. Average the counter values from each of these machines for a specific instance in time (after the gradient had first been applied). Perform this process for all instances.

We are attempting to create a representation of normality by averaging the output from many machines, in an attempt to minimise the effect of the virus from those machines which have been infected. We then use these averaged instances as the training set for the novelty classifier. The instances from the machine we are attempting to determine whether it has a virus or not are used as the test set.

Figure 24 depicts the results we obtained. We discuss an alternative approach to this in the critical evaluation.



Statistic	Value
% Correctly classified	69.7674
% Incorrectly classified	30.2326
Area under ROC	0.7626

Figure 24: ROC curve generated with simulated real-world activity (**Scenario 2**)

6 Critical evaluation

We now offer a critical evaluation on the project. In order to do this we will refer to the original objectives.

6.1 Build a virtual network to safely observe virus propagation

We believe this was successfully accomplished as a virtual network was set up, which to a reasonable extent modelled a small LAN (local area network). The virtual network also provided Internet access in the experiments were we simulated real-world activity.

In hindsight it would have been interesting to compare the effects of virus propagation under different network topologies.

6.2 Capture or download viruses

We were able to obtain viruses from `http://www.offensivecomputing.net/` and `http://vx.netlux.org/`. Unfortunately a large number of viruses which we tested, especially modern ones such as Conficker, didn't actually spread inside the virtual environment. As mentioned in section 4.4 we believe this is due to the ability of some viruses to determine whether they are in a virtual environment and refuse to propagate. However we feel we were still able to conduct useful experiments with viruses which propagated successfully.

Additionally we were able to set up a 'honeypot' to catch viruses. It was especially interesting to see how many viruses this caught in a reasonably short amount of time.

6.3 Design a visualisation to view the spread of viruses

We feel the virus observatory was able to successfully depict the propagation of a virus in a visual manor whilst also graphing the number of viruses in the whole network.

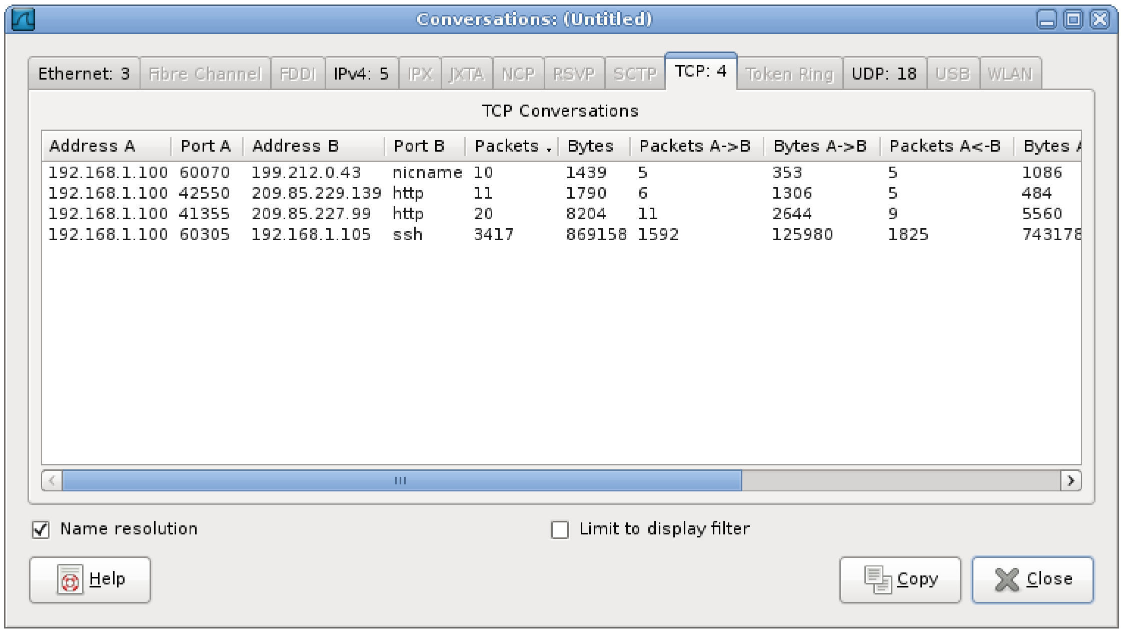
6.4 Allow the virtual machines to log system & network activity (features)

We logged perfmon data from the virtual machines, which we felt allowed the machine learning techniques to adequately detect virus like activity.

Unfortunately time didn't permit us to create a program to calculate the ratio of fan-out to fan-in connections, but we believe their use in addition to the perfmon counters, could enable additional increases in classifier accuracy.

Calculating fan patterns was more difficult than expected, as perfmon and network monitoring commands such as netstat aren't able to give figures for the number of incoming and outgoing TCP connections. They are able to give the number of active TCP connections, but not whether they were initiated by this computer or another.

We found the packet sniffing program Wireshark was able to display TCP conversations along with the source of the connection (see figure 25). Wireshark is primarily a GUI program, but as it is open source and makes use of PCAP, a packet capture library, a program could be developed from it, which simply outputs fan ratios.



Address A	Port A	Address B	Port B	Packets	Bytes	Packets A->B	Bytes A->B	Packets A<-B	Bytes A
192.168.1.100	60070	199.212.0.43	nicname	10	1439	5	353	5	1086
192.168.1.100	42550	209.85.229.139	http	11	1790	6	1306	5	484
192.168.1.100	41355	209.85.227.99	http	20	8204	11	2644	9	5560
192.168.1.100	60305	192.168.1.105	ssh	3417	869158	1592	125980	1825	743178

Figure 25: Depiction of network conversations and the address they were initiated from

6.5 Produce a visualisation of these features in the form of a matrix

We successfully made a program to visualise features in the form of a matrix, where each square corresponds to a particular counter and the intensity of the colour corresponds to the value of the counter.

It was useful in that we could use it to easily spot features which don't change at all. We found however that when normalisation was enabled the counter values changed too rapidly to be effective at allowing us to find patterns between the features of multiple machines.

In retrospect it would have possibly been more useful to show matrixes of a single machine's activity with the virus and also without it, simultaneously (naturally this would require the machine's activity to have been logged beforehand).

6.6 Use formulae such as information gain to automatically select important features

Information gain proved to be highly effective at determining which features should be used by the classifier or novelty detector.

6.7 Develop a system to allow the exchange of each virtual machine's feature set with other machines, with the goal of improving each machine's ability to detect a virus

This was accomplished to a degree by making use of the OneClassClassifier and using averaged instances as a training set as explained in Section 5.5.2. The accuracy of this was only 69.76% however.

In retrospect a more sophisticated mechanism for achieving this objective could have been developed, taking into account the fact that some machines generating data for the training set may have viruses.

One way in which this could be done is through the use of clustering algorithms, to group machines which appear to show similar behaviour together. This clustering process would be on-going, as the clusters of similar machines would change during virus infection. We hope with such an approach the majority of machines would be clustered together, then when a virus outbreak occurs, the cluster may fracture. By monitoring this process, detection of machines exhibiting virus-like behavior should be possible.

6.8 Conclusion

The OneClassClassifier was able to dramatically outperform our novelty classifier when its densityOnly option was enabled. Given more time, we would investigate ways in which our classifier could be improved.

The fact that the OneClassClassifier was able to obtain an accuracy of 90.02% whilst using simulated real-world activity shows good promise for the use of novelty detection in detecting unknown viruses.

7 Improvements

One particularly interesting area for further research would be investigating the detection of multiple viruses on a computer using machine learning techniques. This wouldn't be possible using a single naive bayes classifier, as it can only return a single class for an instance of data.

By obtaining more viruses which propagate effectively we could further strengthen our conclusions.

Another approach for detecting viruses which would be interesting to try, could involve comparing system calls made by viruses to those made by normal programs. This could be performed using the program 'strace' which is able to record system calls a program makes as it runs. This information could be logged for both viruses and normal programs. We could then attempt to find system calls which are most prevalent across normal files and also system calls most prevalent across viruses. This information could then be used to rank a file on whether it is more likely to be a virus or a normal file.

The quality of the virtual network used could be improved by taking advantage of some of VDEs (Virtual Distributed Ethernet) advanced features, such as the ability to simulate packet loss and the ability to set the bandwidth of connections.

We would be interested in performing novelty detection on a 'honeypot' such as the VM we ran on Amazon's ec2 service, as this would present us with a real-world measure of its effectiveness.

8 Bibiliography

References

[1] T. Brown. A practical approach to solving performance problems with sas, 2007.

[2] G. Cohen, H. Sax, and A. Geissbuhler. Novelty detection using one-class parzen density estimator. an application to surveillance of nosocomial infections, 2008.

[3] Fayyad and Irani. Multi-interval discretization of continuous-valued attributes for classification learning. pages 1022–1027, 1993.

[4] K. Juszczyszyn and G. Kolaczek. Assessing the uncertainty of communication patterns in distributed intrusion detection system. In *KES (2)*, pages 243–250, 2006.

[5] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures, 2005.

[6] K. A. Michael, M. B. Greenwald, S. Ioannidis, A. D. Keromytis, and D. Li. A cooperative immunization system for an untrusting internet. In *In Proceedings of the 11th IEEE International Conference on Networks (ICON)*, pages 403–408, 2003.

[7] C. Richardson. Bsc thesis: Sentinel - virus detection & prevention.

[8] K. Rieck, T. Holz, C. Willems, P. Dssel, and P. Laskov. Learning and classification of malware behavior. In D. Zamboni, editor, *DIMVA*, volume 5137 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2008.

[9] D. Stevens, 2007. http://blog.didierstevens.com/2007/10/23/a000n0000-0000o000l00d00-0i000e000-00t0r0000i0000c000k/.

[10] P. Szor. *The Art of Computer Virus Research and Defence.* Addison Wesley, 2004.

[11] M. M. Williamson, M. M. Williamson, J. Lveill, and J. Lveill. An epidemiological model of virus spread and cleanup, 2003.

[12] D. yan Yeung and C. Chow. Parzen-window network intrusion detectors. In *In Proceedings of the Sixteenth International Conference on Pattern Recognition*, pages 385–388, 2002.

[13] A. Young and M. Yung. Malicious cryptography: Exposing cryptovirology, 2004.