Cyber Security: A (Very) Partial Literature Review

Gordon J. Ross, September 2013

[a] *Heilbronn Institute for Mathematical Research, University of Bristol, United Kingdom*

Contents

*Corresponding author
Email address: gordon.ross@bristol.ac.uk (Gordon J. Ross, September 2013)

1. Introduction

The increasing threat posed by cyber attacks has been widely noted, and there is now a substantial volume of literature proposing algorithms for detecting and responding to attacks. While much of this work has been carried out by computer scientists, the area is quite interdisciplinary and has also attracted statisticians, physicists, and complex systems theorists. As a result, the number of relevant papers is large, and continues to grow at a rapid rate.

Since I have an active research interest in this area, I have made an effort to try and get familiar with the literature. While reading around, I wrote notes and summaries of the more interesting papers to remind myself of their contents. Although these notes were originally only intended for myself, it seems possible that others may benefit from them as a partial literature review. This document is a very loosely structured version of these notes, collected together under some general headings. While reading it, please keep in mind that the notes were originally written only for myself and are therefore quite rough. Furthermore, they are also quite biased, in the sense that I have only made notes on the aspects of the literature which interest me personally as a statistician.

As a result, this document is not intended to be an exhaustive review; the field of cyber security has turned into something of a cottage industry, and the literature now consists of hundreds of papers, many of which contain little novelty. The papers which I have selected for inclusion in this review are those which I (subjectively) found to have value, either in the novelty of the algorithms/models, or in the datasets and methods of evaluation. Of course, due to the size of the literature it is almost certain that I have missed many important papers, and it goes without saying that my personal judgement of what constitutes an interesting contribution will differ from someone else's. In particularly, since I am a statistician my interest is mainly restricted to methodological and algorithmic contributions, and therefore I have largely ignored the vast literature relating to the practical implementation of security-related techniques at the infrastructural level, along with the literature which analyses the structure of particular worms, botnets, and attack patterns. Further, because my interests centre on detecting anomalies in network traffic, I have also ignored papers which deal with detecting intrusions based on local file systems and process information gathered from individual computers (e.g. the extensive literature on anti-virus techniques, and malware detection).

For each paper reviewed, I will give a short summary of its contents. This will hopefully be enough to provide the reader with an overview of its general approach. I have also coloured the titles of the papers which I have found to be especially interesting in red and put an asterisk (*) beside them. I would advise anyone delving into the literature to begin with these.

2. Intrusion Detection (Denial of Service Attacks, Port Scanning, and Worms)

2.1. Overview

The first category of papers are those which attempt to detect different types of intrusions into computer networks. This is carried out by monitoring all computers connected to the network in order to detect potentially malicious activity. The usual setting considered in the literature is as follows: software deployed on a set of network routers is used to collect information about all the traffic involving machines in the network, and the external internet (i.e. every connection made by any of the networked computers is logged). This data is then monitored by the network administrator in order to detect when computers on the network are being attacked, or have become compromised. Because the goal is usually to monitor every computer on the network for potential intrusions, the monitoring strategies need to be both fast and scalable, since they may simultaneously be monitoring hundreds or thousands of computers.

There are three primary types of network intrusions which have received attention in the literature:

1. Port scanning: Most attempts to hack into a target computer involve exploiting vulnerable services running on the computer, such as HTTP servers, mail servers, and so on. It is therefore very useful for the attacker to know which services are running on the potential target. Port scanning is the usual means by which this information is obtained. In a port scan, one or more ports on the target computer is queried to find what services are active on those ports.
   There are two types of port scan, vertical and horizontal. In a vertical port scan, many different ports on a single computer are scanned, and many widely available tools allow all ports on a target computer to be scanned very quickly. Such vertical port scanning will usually be used if the attacker has a specific interest in hacking into a particular target machine. In a horizontal port scan, many computers on the target network are all scanned on a single port – for example, the potential attacker may scan hundreds of computers on port 80, in order to find out which ones are running a HTTP server. Such horizontal port scanning is often used by worms, which usually contain only a small number of exploits and hence need to find computers running a particular vulnerable service.
2. SYN flooding (DoS attacks): A denial-of-service (DoS) attack is an attempt to disable a particular computer, removing its ability to communicate with other machines. This kind of attack is usually deployed against network servers, with the (e.g.) bringing down target websites or disrupting a network. The most common type of DoS attack is the SYN flood, carried out using the Transport Control Protocol (TCP). Typically, a TCP connection is initiated as follows: the client sends a TCP SYN packet to the server to request the connection, the server responds with a SYN-ACK packet, and the client then sends a further ACK packet. After this, the connection is established and data can be transferred.
   In a SYN flood, the attacker sends multiple SYN packets to the server in order to open many connections. The attacker then ignores the SYN-ACK packets sent by the server, and does not send any confirmatory ACKs. This results in the creation of a many 'half-open' connections which stay in the server's memory, consuming resources. By creating many such half-open connections, the server eventually

has all its resources consumed and is not able to open any more connections with legitimate clients. Typically around 500 SYNs per second are the minimum required to DoS a server, but when a sophisticated firewall is used, this can increase to 14000/second or more

3. Worms: A worm is a piece of software which has the capability to spread through the internet and local networks, infecting multiple computers. Typically worms are self-replicating – once they infect a computer, they will use this machine as a base in order to find more vulnerable targets to spread further. A worm is programmed with a number of exploits which can take advantage of security holes in unpatched services/software running on potential targets; typically a worm will infect a computer on a network and then engage in port scanning behaviour to find further targets running these vulnerable services, and attack them.
   Worms differ in terms of the exploits used for infecting targets, and also in terms of their payload (i.e. what they do to the computers they infect). Many worms are non-malicious; once they infect a computer, they carry out no further actions other than trying to spread themselves further. However malicious worms are also very common, and may cause infected machines to (e.g.) become part of large botnets, as discussed further in the second half of this document.

Methods and algorithms for detecting these types of intrusions can be classified in many ways. One of the most important distinctions is between methods which assume a great deal of prior knowledge about the attacks they are trying to detect, and those which do not. For example, it is relatively easy to write a detection system which can detect a particular worm, given extensive information about the mechanisms the worm uses to transmit itself. However such signature-based methods will not be effective at detecting new unseen worms, where no such detailed information is available. Therefore much of the literature (particular the branch in which I am interested) focuses on detecting intrusions when only very limited information is available about the characteristics of the attack. In this case, the usual way to proceed is to build some model of how computers on the network normally behave, and then try to detect any abnormal deviations from this, which may be indicative of an attack. This is essentially a case of anomaly detection.

One thing which struck me when reading through the literature is just how important it is to look at the right subset of features in the data; in general, intrusion detection seems to be very easy if one knows roughly what to look for, and very hard if one does not. For example, trying to detect DoS attacks by looking only at the number of connections a server receives is incredibly difficult, since a spike in the number of connections received may be due to perfectly benign reasons. On the other hand, tracking only the number of half-open connections (i.e. SYN packets without corresponding ACKs) makes detecting DoS attacks relatively easy [19]. Similarly, horizontal port-scans become much easier to detect when one monitors the number of **failed** connections, rather than the total number of connections [8]. Therefore, having some kind of solid domain knowledge is important, even when one does not want to (e.g.) assume detailed knowledge about the sort of worms which will attack the network.

2.2. Data Sets

Most of the papers reviewed in this survey lament the lack of publicly available data sets containing known attacks. This makes it difficult to assess the performance

of the proposed detection algorithms, since there is no truthed data against which their performance can be validated. There are hence three general techniques used to assess performance:

1. **Untruthed real network data**. Most of the authors writing on the subject of intrusion detection have access to real network traffic data. Typically this is taken from a university network and will consist of all the communications made by each computer on the network over a period of several days/weeks. It will not be known in advance whether this data contains any intrusions. The detection algorithms are run over this data, to see if any intrusions are found. If they are, then these flagged intrusions are investigated in more detail to check whether they seem to correspond to real intrusions. While this approach is useful for finding the false positive rate of the intrusion detection algorithm, it is possible that there will be intrusions in the data set which are not detected (i.e. the false negative rate is not obtained).
2. **Real network data containing injected intrusions**. In this case the authors begin with the type of real network data described above, and inject synthetic attacks. They then check whether their algorithms are able to detect these attacks. While this does allow the false negative rate to be estimated, care must be taken to ensure that the injected attacks actually resemble real attacks, rather than being over simplistic.
3. **Real network data containing known intrusions**. This is the gold standard for assessing detection performance, but unfortunately very little data sets exist. The exceptions are the various DARPA intrusion datasets used in some papers (see [11] for more information about these). However, they are now quite old and do not contain attacks from the last few years. This is unfortunate, as the behaviour of attackers can change over time as they seek to evade existing detection methods.

Additionally, it is worth distinguishing between two types of data which are commonly used. The first is **netflow** data, which is a high level aggregation of the packets sent between pairs of machines as part of a single communication. A typical netflow data set consists of a number of records, where each record contains the IP addresses of both machines involved, along with the time the connection was initiated, the port used, the length of time the connection stayed open and the number of packets/bytes transferred. However the contents of the actual packets transmitted (including TCP meta-data such as whether packets had their SYN/ACK/etc flags set) are not available. A typical netflow data set will resemble the following:

```
Date flow start          Duration Protocol Src IP Addr     Dst IP Addr    Src Pt Dst Pt Packets Bytes
2009-04-22 12:04:44.664   13.632   TCP      126.253.5.69    124.195.12.246 49882  80     1507    2.1 M
2009-04-22 12:04:42.613   16.768   TCP      126.253.5.69    124.195.12.246 49881  80     2179    3.1 M
2009-04-22 12:04:36.404   17.600   TCP      100.253.210.138 100.253.192.99 1104   80     6017    276899
2009-04-22 12:04:58.736   17.344   TCP      126.253.5.69    124.195.12.246 49888  80     1708    2.3 M
...
```

The second type of data consists of the raw **packets** themselves. This provides more information than netflow data, at the cost of being much larger and more difficult to work with. However the TCP (or similar protocol) meta-data stored in the raw packets can be invaluable for certain types of intrusion detection algorithms. This type of data can

be further divided into cases where only the meta-data is available, and cases where the application data (i.e. the user content) of the packets is also available. The latter case provides the richest and most detailed source of data, but will usually be unavailable for both privacy/legal reasons, and the more practical fact that many attackers and worms will often encrypt all their communications, rendering the packet contents unreadable.

2.3. Summary of Papers

I will first give the name of each paper in bold text, followed by a several paragraph summary. Recall that the titles of the papers which I think are most interesting are displayed in red text, and are preceded by an asterisk (*).

*** Gao et al - A DoS Resilient Flow-level Intrusion Detection Approach for High-speed Networks** [6]

The purpose of this paper is to monitor a network containing many computers, and find any machines which are being attacked. The aim is to develop a scalable solution which can be run on large networks, and which can distinguish between various types of malicious behaviour (e.g. SYN floods, horizontal/vertical port scans, etc).

Their method is based on key-value pairs, which track the number of times particular keys occur in the netflow data obtained from the network. Let (K,V) denote the tracked pairs. The choice of key depends on the type of anomaly being detected. For example, using keys of the form K=(DestinationIP:DestinationPort), for example K=(192.162.10.2:80), would count the number of times a particular IP on the network has been accessed on a specified port, while a key such as K=(SourceIP:SourcePort) would count the number of times a machine outside the network has connected to a machine in the network, using a specified port. To allow for scalability, they use the reversible sketch method from [16] discussed below to count the number of times each key occurs, which allows the keys which has abnormally high values (i.e. have occurred often) to be tracked.

The choice of key is based on the type of attack to be detected; Table 3 in the paper gives a good summary. For example, K=(DestinationIP:DestinationPort) can detect SYN flooding, but will not flag a port scan since only one port is being monitored. Similarly, K=(SourceIP:DestinationPort) can detect horizontal portscans, while K=(SourceIP:DestinationIP) can detect vertical port scans.

In order to get more sensitive results, they also build up a model for the expected behaviour in the network under normal (non-attack) conditions, and subtract the predicted value of each key from its actual value (i.e. they are not interested in the absolute number of times a key occurs, only the difference between its current number of occurrences, and the usual number of occurrences during typical behaviour).. Specifically they use a EWMA forecast, and consider only the forecast error

Finally, they note that there can be problems distinguishing between the different attack types, since attacks are multidimensional. Consider a SYN flood, and a (vertical) port scan on one computer. Both involve the attacker sending many packets to one machine. The different is that for the SYN flood only a small number of ports are targeted, whereas for a port scan a large number are targeted. They hence need to consider the port distribution, since using K = (SourceIP, DestinationIP) is insufficient. They instead use

a 2d hash table. The first dimension is the standard sketch for (SourceIP,DestinationIP). But then instead of just counting the number of such keys observed, they instead index this hash table by another hash table, which counts (for each port) the number of times that port was accessed. So they first key is (SourceIP,DestinationIP), and then they key based on DestinationPort.

They test their method by analyzing netflow data, which was collected in one day and is about 2 terrabytes in size. They do find a lot of anomalies, but since the data is untruthed its hard to make conclusions. However, they do check a small number of the anomalies manually, and claim most seem to correspond to real attacks.

My comment: The main reason why I like this paper is that it illustrates how important it is to track the right data; finding port-scans is different from detecting SYN flooding, and requires looking at a different aspect of the data (i.e. different keys). Also, although it is not discussed in the paper, it seems feasible this method could be used for bonnet detection by tracking keys relating to DNS failures or similar.

*** Jung et al - Fast Portscan Detection Using Sequential Hypothesis Testing** [8]

This paper begins by discussing the conceptual problems associated with defining a portscan - if (e.g.) many external IP addresses simultaneously scan several computers on a network looking for a HTTP behaviour, should that count as a horizontal port scan? But it is possible for such behaviour to be legitimate. Similarly, within what time window do the scans need to occur - if one port is scanned every 5 minutes, would that still be a port scan?

The authors decide to define a port access as being a port scan if the accessor doe not go on to establish a useful connection (in the sense of transferring any further data). They also decide to focus only on cases where it is a single malicious IP performing the port scan, and also consider only horizontal scans where many computers on the network are scanned on the same port.

In their literature review, they point out most previous work attempts to detect port scans by looking only at the number of FAILED connections (in the SYN/SYN-ACK sense), rather than the number of connection attempts total. This seems sensible

Their detection method uses a sequential likelihood ratio test. Suppose that an external IP starts scanning through different computers on the monitored network, and for each computer it tries to connect on the same port (e.g. port 80). For each of the connections i, define $Y_i = 0$ if the connection was a failure, and $Y_i = 1$ if it was a success. Then, $Y_i \sim Bernoulli(\theta)$ for some parameter θ.

If the external IP is non-malicious it will usually know which computer/port on the network to connect to, and so most connections will be successful. On the other hand, a malicious port scanner will create a lot of connection failures since it essentially scanning random machines. Define the hypotheses H_0: the external IP is benign, and H_1: the external IP is malicious. These hypotheses correspond to different values of θ. Under H_1 (i.e. assuming random scanning), the corresponding value of θ will be the empirical proportion of machines in the network running a service on that particular port (i.e. the probability of a connection to a random machine being successful). whereas under H_0, θ

will be high, since most non-malicious connections are successful. Therefore a sequential likelihood ratio test can be used to detect the intruder, according to a specified false positive bound. It is not clear how to actually choose θ under H_0 though, and this is never really discussed. Perhaps it is done empirically?

As an important note, they only consider the FIRST connection attempt made to each computer on the network by the intruder. In other words, they only form a Y_i variable when the external IP connects to a host is hasn't connected to before – further communications to a host it has previously connected to are ignored. In order to find whether the connection was indeed successful, they look at whether the scanned machine replied with a SYN-ACK packet.

They use netflow data from a large network to evaluate performance. This network already uses a a port-scan detector (snort), so the data is partially truthed. However if their algorithm is better than snort, it will find portscans which are not currently identified. They admit this is somewhat circular reasoningr; they will argue that malicious scans have certain characteristics, and then show their method successfully detects connections that have these characteristics.

Despite this, they do find that their method generates false alarms. Checking in more details, it seems that most of these were generated by (non-malicious) web-crawlers and spiders.

Future work (from their paper): 1) using additional information, (e.g. using more conservative parameters for HTTP compared to other services since HTTP is legitimately scanned more by proxies), 2) modelling the behaviour of computers in the network to better find anomalies (e.g. if a machine has been inactive for a long time, then a failed attempt to connect to it is more suspicious than it would be for an active server), 3) explicitly modeling which machines are more likely to be visited by benign rather than malicious connections.

My comment: Their method won't work against distributed port scans where many remote machines scan different parts of the network. Detecting this would require some kind of pooling, where we pool together different anomalies at the same time, each of which individually may not be sufficiently anomalous to cause the detector to flag.

Collins et al - Hit List Worm Detection and Bot Identification in Large Networks Using Protocol Graphs [3]

Existing techniques for detecting the spread of worms through computer networks often involve tracking the number of connection failures initiated by an infected host. This is because once a host is infected, the worm will try to spread to new hosts. In order to do this, it will typically try to connect to a large number of neighbouring machines. For the machines which are not running a vulnerable service, these connections will fail. Therefore it is sensible to track features such as the number of unsuccessful connection attempts initiated by a host, or the rate at which the host initiates connections to new targets. Abnormally high numbers of connection failures often implies malicious worm-related behaviour, as worms will often engage in mass portscanning in order to find more potentially vulnerable targets on the network.. To avoid detection, some worms come with a hit-list, which has been generated beforehand and contains a list of machines

which are running a known server (i.e rather than the worm portscanning every machine on a network , it instead is given a list of machines which are known to be running a vulnerable server). Therefore it makes far fewer connection attempts since it does not need to engage in mass scanning.

This paper instead tries to detect worms by monitoring for abnormal connections within the network (quite similar to the LANL work) Given a protocol such as HTTP or FTP, they construct a protocol graph where each node represents a computer on the network and nodes are linked if they have communicated. If a worm is present in the network, it should cause connections between machines which do not normally interact. This results in a) more vertexes in the graph (since more machines are using the protocol), and b) more vertices in the largest connected component, since the worm will link previously disconnected components

Given some observation period (they use 60 second windows) they collect packet data of the form (source IP, destination IP). Each IP address is then a vertices, with links if they communicated during the window. Let $|V|$ denote the number of nodes in the resulting graph, and $|C|$ denote the number of nodes in the largest connected component. Their idea is to find the distribution of both $|C|$ and $|V|$ under normal (non-attack) behaviour, and then to look for anomalies where one of these increases.

The method is tested on real network data. They learn the normal behaviour using one day of data; interestingly, there is a problem since the data they have contains a port scan which causes the graph to blow up. They first remove this using the method from [8] discussed above. Synthetic attacks are then injected into the network, and detected.

My comment: Although the idea of monitoring the connection graph of the network is sensible, the actual method they use (simply tracking $|C|$ and $|V|$) seems very crude. However, in practice perhaps most worms are so obvious in their behaviour that this would be enough? I doubt it would be able to detect more sophisticated intrusions.

Wang et al - Detecting SYN Flooding Attacks [18]

Wang et al - Change-Point Monitoring for the Detection of DoS Attacks [19]

These two papers are very similar and attempt to detect SYN floods (DoS attacks). Recall this involves the attacker creating many half-open TCP connections on the target computer by sending a SYN packed but not sending the ACK after the target responds with SYN-ACK. Each half-open connection stays in the targets memory and uses up resources, meaning eventually that the target will be unable to open any new connections.

For each monitored computer, the authors track the **difference** between the number of SYN packets sent to the computer (i.e. the packets which open a connection) and the number of FIN packets sent to the same computer (i.e. the packets which close a connection). Under normal behaviour, these will be roughly equal since each opened connection should eventually be closed. But during a SYN flood, connections are not terminated, and so there will be more SYNs than FINs. Although this is a simple idea, there are some additional technical issues which arise, for example 1) TCP connections can also be closed via a RST rather than FIN flag, which the authors briefly explore, and 2) there are non-malicious scenarios where the SYNs may exceed the FINs, e.g. if there

are a lot of connected users with TCP connections which remain open for long periods (for example, a server hosting a very large file with multiple downloaders).

There is an interesting discussion about the distribution of the arrival times of TCP connections under normal (non-attack) behaviour. Apparently it is both bursty and heavy-tailed which makes it a non-Poisson process (see also: [15] described in detail below). This leads the authors to use a nonparametric CUSUM for change detection rather than a parametric model which assumes a Poisson distribution. They discretise the packet data into windows and define X_i as the difference between the number of SYNs and FINs in each time window, where the window length is set to 20 seconds (since apparently most legitimate TCP connections stay open for 10-15 seconds). They model seasonality by forming a running estimate of the number of TCP connections which are expected under normal behaviour (using a EWMA forecasting model), and then normalize the X_i variables based on this. An attack is flagged if the difference between the number of SYNs and ACKs in a window exceeds a certain threshold.

They evaluate performance on several real data sets. For each, the SYN/FIN packets are extracted by looking at the packet headers. They don't find any attacks in the data so they inject some, simulating a DDoS attack from many hosts, which is successfully detected.

My comment: again, this paper shows the importance of looking at the right features of the data. Once the idea of monitoring the difference between SYNs and FINs is mentioned, the algorithm almost writes itself.

* Chen and Yeung - Defending Against TCP SYN Flooding Attacks Under Different Types of IP Spoong [1]

This paper focuses on distributed DoS/SYN-flood attacks where the malicious IPs are spoofed (i.e. where the IP of the attackers is unavailable). This is a more realistic setting than many of the other papers. In terms of methodology, it is similar to the Wang paper immediately above [19], in that they count the number of SYN and ACK packets (they do not have access to the SYN/ACK packets for technical reasons).

Their algorithm is intended to monitor multiple computers in the network to check whether they are being DDosSed. It is based on two Counting Bloom Filters, which is an efficient way to count the number of times a key has been observed, for many different keys (similar to a counting sketch). Whenever a SYN packet is observed, they find the target IP it is directed at, and increase the count associated with this IP in the Bloom filter. Similarly when an ACK is received, they decrease the count associated with the target IP (i.e. in other words, for each machine in the network they maintain a count of the number of open connections). As in [19] above, they argue that in normal circumstances each target IP should have a number of SYNs and ACKs, so all counts in the Bloom filter should be close to 0. A nonparametric CUSUM is then used to monitor each of the counters in the Bloom filter for changes, indicating a DDoS. They then use the DARPA benchmark intrusion sets discussed in [11] to test their method.

My comment: I think this paper has good practical value: the efficiency of the Bloom filter allows them to monitor many computers at once and, since it doesn't use source IP information, it can handle iP spoofing. However their method would probably benefit

from using some kind of False Discovery Rate control rather than monitoring each count separately; when a DDoS occurs, more than one counter will always be affected (since each key is mapped to several counters in the Bloom filter) so FDR would be nicer. Its also not clear how to choose the CUSUM parameters

Wang and Stolfo - Anomalous Payload-based Network Intrusion Detection [20]

This paper focuses on detecting zero-day worms (i.e. new worms which were previously unknown). Since these worms are unknown, one cannot rely on signature-based anomaly detection methods. Instead, they plan to model what the normal content of the payload of a non-malicious network TCP packet looks like, and then detect deviations from normal behaviour. Zero-day worms should contain payloads which are different from anything seen before, since they are new. Payloads are analyzed based on their byte frequency (note that 'payload' here is used to denote the user content of the TCP packets - i.e. the actual application data - rather than the headers and TCP meta-deta).

Since a byte contains 8 binary bits, there are only 256 unique bytes. For each port, they estimate the empirical byte distribution associated with typical (non-malicious) packets. Then, if any new packet contains bytes which are unusual given this empirical bytes distribution, it is flagged as anomalous. There is a small complication in that they also construct a different byte distribution for packets of different lengths, but this is conceptually unimportant.

My comment: This method is very impractical since a) it requires too much knowledge (its unlikely one would actually have access to user packets), b) it won't detect any worms that use non-standard ports, c) there will be many false positives when users change their behaviour.

Ellis et al - Graph-based Worm Detection On Operational Enterprise Networks [4]

This paper begins with a discussion of just how fast worms can spread; e.g. the Slammer worm infected 95% of all vulnerable hosts on the internet within 10 minutes of being released. This means that it is vital for enterprise networks to use fast automatic detection systems rather than relying on slower human analysis. False alarms (= false positives) are a huge problem for such systems due to the high costs associated with mistakes; responding to a worm attack may involve shutting down a company network. This can have a higher associated cost than just letting the worm run without intervention. The authors aim to build a system that can detect many types of worms, with less than 2 false alarms per day (which still seems high to me...)

They then discuss just what a 'false positive' means. One single event may cause the intrusion detection system to signal multiple alarms. However in some sense these are all 'the same' alarm, in that they are a response to the same event. They refer to this sort of clustering of alarms as 'deadbanding'. Therefore they also use another false alarm

measure defined as the percentage of time each day that the system is signaling for an alarm.

Next, they give a simplified taxonomy of existing worms. Roughly: worms can be classed as unimodal or multimodal based on whether they transmit using a single exploit, or multiple exploits. They can also be classed as polymorphic or nonpolymorphic based on whether the worms adaptively change their behavior and/or network footprint. There is also a distinction based on how fast worms attack other machines; the higher frequency ones are typically easer to detect. Finally, worms can be classed based on how they find new targets to attack; some worms will scan random IP addresses, some have a predefined hit list of vulnerable machines, some scan subnets, some look in the .rhosts file of infected machines, and so on.

Their detection algorithm involves constructing a communications graph to model causal flows in the network. As usual, every machine in the network is a node in the graph. An edge exists between two nodes if a certain predicate is satisfied. The predicates depend on the worm being detected. Page 12 has an interesting list of predicates specific to different worms; if one knows how a worm behaves, it is possible to be quite specific. For example, the Sasser worm spreads if A opens a TCP connection with B on port 445, then a connection on port 9996, and then B connects to A on port 5554 and downloads the worm code. To make their method more general, they consider predicates which are less specific such as TCP SYN packets being sent on specified ports (port 445 is used by both Sasser and Zotob, so using this can catch multiple worms), and any UDP packets being sent.

The graph they construct is actually a tree, in that it is causal with one infected host then infecting others. Based on the time ordering of the packets, they can construct a descendent tree, and then compute summary statistics like the branching factor, depth, and suchlike. Given a set of netflow data, they create this descendent tree for each node. They then flag an anomaly if certain summary statistics of the tree exceed critical values. The summary statistics they use seem quite convoluted, there is a table in their appendix but it all depends on the particular port, and suchlike. I think they found these thresholds by picking sets of values which have no false alarms on the real data they have (which is assumed to be worm-free) - i.e. they found them via a greedy search by putting a bound on the alarm rate, and finding thresholds which achieved this threshold. Finally they have to choose their window size, which is (as usual) slightly arbitrary, and involves a trade-off between false alarms and fast detections. However they note the window size should also depends on the particular worm and how fast it spreads/scans. Eventually, they chose a 5 minute window. With this window, they found that any host having 2 grandchildren (i.e. depth 2 descendants, or paths of length 2) within a minute was sufficient to sound an alarm

Interestingly, they found that most false alarms were caused by the same small number of hosts, which repeatedly exhibited abnormal behavior due to the services they were running. These hosts were put on a white list and ignored. This does not impact worm detection, since if these hosts actually did get infected, the worm should be caught when it spreads to another host.

Their empirical study of how well their algorithm detects worms is interesting. Although they do not have a worm in their real data, they pick a real worm (Zotob) which has a well understood transmission mechanism. Then they investigate how well their detector would have detected it, had this worm connected to the network. They look

at best/worst case estimates based on how many machines are vulnerable, whether the worm has a hit list, and so on. However this is quite an easy , since they are monitoring port 445 (used by Zotob) as one of their link predicates, which is very strong prior information.

*** Paxson and Floyd - Wide-Area Trafc: The Failure of Poisson Modeling** [15]

(First note this paper is from 1995 and so is quite old, which is relevant since they are doing data analysis – for example they claim most of their datasets don't contain much WWW traffic)

The point of this paper is to argue that many types of web-traffic do not follow Poisson processes, in the sense of having Exponentially distributed inter-event times between the creation of connections. They argue recent (in 1995) work shows that 'self-similar' and heavy-tailed renewal processes are a better fit than the Poisson process. But their contribution is to show that in their datasets, different protocols have different distributions; for example Telnet and FTP tend to be Poisson, while NNTP and WWW tend to not be. Their data uses SYN/FIN packets to log the times at which connections were initiated, and they also have raw packet data to monitor for burstiness.

To expand: they claim for both Telnet and FTP, the initiation of connections by a host is well described by a Poisson process. But within a particular connection, the packets are not Poisson due to burstinesss. For SMTP traffic, even the initiation of connections is not Poisson, since this is mainly machine initiated and can be periodic.

In Section 3 they divide their data into 1 hour windows and subtract daily seasonality effects. For each protocol, they look at the inter-arrival times between session initiations within each 1 hour interval, and test whether these have an Exponential distribution. They then claim that at this level, only FTP and Telnet arrivals are homogenous Poisson, whereas WWW and SMTP aren't. I found this section quite confused because they don't make it clear whether the inter arrival times are between the imitation of ANY 2 connections, or between two connections initiated by the same person. I would expect these to have very different characteristics, because if one is aggregating traffic rather than looking at individual people, burstiness should be averaged out.

They then claim NNTP (usenet) traffic is not Poisson because clients often spawn secondary connections (i.e when a computer connects to a news server, it will typically connect to several other hosts in order to download all messages). Similarly with WWW (internet) traffic, when a user connects to a site they will usually also spawn connections to all the advert and image hosting servers , creating burstiness.

They next look at inter arrival times within sessions for both Telnet and FTP. Unlike the session initiation times, these are not Poisson and they discuss heavy tailed power law (Pareto) distributions for modelling this.

Krishnamurthy et al - Sketch-based Change Detection: Methods, Evaluation, and Applications [10]

Schweller et al - Reversible Sketches for Efficient and Accurate Change

Detection over Network Data Streams [16]

These papers are similar in that they both develop sketch-based methods to find changes in a computationally efficient manner, which allows a large number of machines to be monitored simultaneously, even when traffic is arriving at a high rate. More algorithmically, they perform change point detection in cases where there is a very large number of sequences to be monitored.

The basic problem they are trying to solve is this: we observe (key,value) pairs, where the value counts the number of times the key has occurred. We wish to detect changes in the rate at which the keys arrive (which can correspond to an increase in the number of TCP packets, portscans, etc). These papers work in discrete time using windowing – for each window of size W, changes in a single sequence can be detected by counting the number of times the key appears in the window, and comparing this to the forecasted value, based on all the previous windows and some forecasting model (e.g. ARIMA, Holt-Winters, etc). If the difference between the empirical value and the forecast exceeds some threshold, a change is flagged. The question is then how to scale this up so that many different streams can be simultaneously monitored. Their solution involves sketches.

The sketch they use is an extension of the Count Sketch. They have H hash tables each containing K bins, where K is much smaller than N (N = the number of keys that are being monitored). Each hash table is associated with a unique hash function. Whenever a key is received, it is hashed using each of the H functions, and the corresponding bin in each hash table is incremented. Based on this, they propose an estimator of the count (value) associated each key which is unbiased, with variance proportion to (K-1). However, the variance they get is only a bound, not an exact number. Similarly, they propose an estimator of the sum of squares of all elements in the sketch, which is used for change detection.

The change detection is carried out by simply checking whether the forecast error is above T * sqrt(sketch variance), for some threshold T.. Because of the inaccuracy in the count estimator, combined with their forecasting model, they cannot produce a p-value. (which also prevents them from doing anything more sophisticated like False Discovery Rate control).

The Krishnamurthy paper has a serious problem related to the Count Sketch that it uses: since they do not log the values of the keys themselves (as there are too many, and doing so would remove the efficiency gains from the sketch), they are only capable of saying whether there has been a change point in the sequence of counts associated with at least one key, but they cannot recover which particular key has changed. The Schweller paper introduces a new reversible sketch method to fix this problem, and allow the changed keys to be recovered. Their insight is based on the fact that only the sketch updating needs to performed fast, in a streaming manner. The change detection part on the other hand can be performed offline, and perhaps only needs to be run every few seconds. This is essentially an example of stream-cloud integration.

Note the original Count-Sketch paper ("Finding Frequent Items in Data Streams") seems to have more detailed theoretical results.

My comment: as future work, is it possible to get p-values? This involves compensating for the inaccuracy in the sketch counting, and would make it possible to use more sophisticated change detection methods.

3. Detecting Botnets

3.1. Overview

The second class of papers deal with the detection of botnets. A botnet consists of a number of host computers which have been compromised, and are under the control of a botmaster. Typically the botmaster will use their botnet for purposes which are both illegal and profitable, such as mass email spam, distributed denial of service attacks, and Bitcoin fraud. The largest botnets in the world are estimated to contain hundreds of thousands, if not millions, of bots, and their use in criminal behaviour is becoming an increasing cause for concern.

Most botmasters recruit new bots into their botnet through the use of worms. Whenever a computer is infected with the relevant worm, it will be recruited into the botnet and fall under control of the botmaster. Therefore, the worm detection techniques reviewed in the previous section can also be considered as botnet detection algorithms.

Rather than focusing on worm-detection, the papers reviewed in this section typically attempt to find computers which have already been infected, and are now functioning as part of a botnet. Further, a recurring goal is to find multiple machines on a network which are part of the same botnet. Typically this is done in a two-step procedure; first, a detection algorithm is used to find all machines on the network that are behaving suspiciously. Then, some kind of clustering is performed in order to cluster these machines into groups which are exhibiting similar suspicious behaviour, in the hope that machines which are grouped together will be part of the same botnet.

In order to identify suspicious machines, two criteria are used. The first attempts to find machines which are involved in typically bot behaviour, such as port scanning or DoSing other computers, and so on. The second attempts to find machines which are receiving suspicious communications, which may be the botmaster giving them instructions. These communications between the botmaster and their bots is known as command-and-control (C&C) and are the distinguishing characteristic of botnets. Different botnets use different types of (C&C), and some of the most popular include:

- **Centralised static C&C**. This is the oldest form of C&C infrastructure, and the most easy to detect. In a typical centralised scheme, all bots in the botnet receive their instructions from a single source, with an IP address that is fixed over time. Historically, the most widely used type of centralised control was Internet Relay Chat (IRC). Whenever a machine became infected and joined the botnet, it would join a specified IRC chatroom on a particular IRC server and await instructions. When the botmaster wanted infected machines to perform an action, he would transmit the instruction to the chatroom, and all bots in the room would automatically carry it out. Similar centralised control can also be designed in ways which do not use IRC, such as by having the bots connect to a specified website in order to download botmaster instructions. The vulnerability inherent in all these centralised control schemes is that if the centralised source gets disabled (such as the IRC chatroom being closed, or the website being taken offline) then the botmaster will no longer be able to control their bots, and the botnet will be disabled.
- **Centralised dynamic C&C**. Using a dynamic control structure allows the above problems to be mitigated. In this case, bots still receive their instructions from

a centralised source, but the IP address of this source changes over time. This avoids the problem of the botmaster losing control due to the source being taken down. Sophisticated modern dynamic C&C infrastructures often make use of domain generation algorithms (DGAs). In this case, each bot is equipped with software which generates a new random set of website addresses every day (typically these names will be nonsensical, such as www.qpznynszue.ru). These generated addresses change each day, but all bots in the botnet generate the same names. The bots then try to connect to each of these generated address until they find one which is operational, and then proceed to receive their instructions from it. Because the botmaster knows which addresses will be generated on a given day, he is able to register one of these in advance and will be able to issue commands to the botnet from it. If the registered domain is subsequently shut down then the botmaster will not lose control of the botnet, since he need only register one of the new addresses that will be generated on the following day.

- **Distributed C&C**. Distributed C&C provides an alternative to centralised methods, and is typically implemented using a peer-to-peer (P2P) protocol. In such a system, each bot contains a list of the IP addresses of a selection of neighbouring bots on the network. Each bot in the botnet then regularly communicates with its neighbours. When the botmaster wishes to issue commands to the botnet, he need only give the command to a single bot. The command will then spread through the botnet, passing from neighbour to neighbour, until all bots have received the command. Such distributed infrastructure is difficult to take down, providing the botmaster with a great deal of security.

3.2. Summary of Papers

Again, I will first give the name of each paper in bold text, followed by a several paragraph summary. Recall that the titles of the papers which I think are most interesting are displayed in red text, and are preceded by an asterisk (*).

* Gu et al - BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection [7]

This paper is quite influential in the field of botnet detection, and is heavily cited. It focuses on detecting non centralised botnets (e.g. P2P). Its distinction between clustering bots based on their malicious activity, and their C&C, is also widely used.

The intuition used in this paper is that all bots in the same botnet should behave in similar ways. There are two aspects to this. First, all bots in a botnet will communication in a similar manner (either receiving instructions from the same centralised botmaster, or by making similar peer-to-peer requests to their neighbours). Also, they will tend to perform similar activities (e.g. DDoSing or port scanning the same target). The authors hence try to detect botnets by clustering the computers in a network based on their behaviour patterns, in the hope that certain clusters will pick out all the machines that are part of the same bonnet. The task is then not to detect particular compromised machines (as in intrusion/worm detection), but instead to detect a whole botnet containing similar bots. This obviously implies the network dataset must be large enough to contain multiple bots from the same bonnet.

The clustering is carried out in two different spaces - the 'C-plane' where machines are clustered based on their communications (i.e. C&C), and the 'A-plane' where they are clustered based on their activity. C-place clustering is carried out by capturing all net flows on the network, and filtering these down to a manageable size based on white-lists and aggregation. Features are defined relating to the number of flows per hour, number of packets per flow, average number of bytes per packet, and average number of bytes per second. A variant of K-means clustering is then used, along with certain dimensionality reduction techniques.

For the A-place clustering, again all net flows are collected. Because the number of such flows is enormous, the authors use an interesting data reduction technique where they throw away all the flows except those which were identified as being malicious by snort (which is a fairly simple open-source system for detecting port scans, DoS attacks, etc). Then, they only perform clustering on the flows which have been classified by snort as being malicious.

After performing clustering in both the A-place and C-plane, the authors try to find correlations between these two clusterings (i.e. groups of computers which have been clustered together in both planes). They define particular scoring methods for measuring this purpose.

Their experimental evaluation consists of collecting data from the university network, which is thought to contain no botnets (based on other botnet detection tools). They then obtain traces of real botnets, and inject these into the network. They then run their algorithm, and find the injected botnets.

My comment: I like the idea of using snort (or similar) to pull out the potentially malicious flows, and then only looking at these. This highlights how botnet detection in the activity-place is somewhat hierarchal; first we need to find the malicious flows, and then cluster them. Clustering is carried out using features such as the type of malicious behaviour (scans vs DoS etc), the ports used, the subnet distribution, etc. Also, the botnet traces they use for evaluation seem useful.

* Choi et al - Botnet Detection by Monitoring Group Activities in DNS Traffic [2]

This paper attempts to find potential C&C communications between hosts and a botmaster in DNS traffic. They focus on detecting IRC botnets.

Most botmasters do not use fixed IPs for issuing commands to bots, since these can be blocked by network administrators. Instead, the botmaster IP is usually dynamic, and bots must use DNS queries to find the current IP of the controller. In most botnets, there are several possible C&C servers rather than just one, to allow adaptability. Typically this server list is stored in the bot (or generated via a domain generation algorithm), and dynamic DNS is used to update the botmaster IP if it changes.

There are 3 main features that distinguish botnet DNS queries from legitimate queries: 1) botnet queries usually occur in groups (i.e. all bots in the botnet DNS the same server). Even if this is staggered over time rather than simultaneous, there should still be a fixed size group contacting one server, and then later DNSing a different server. 2) bots tend to act simultaneously, 3) bots tend to use dynamic DNS rather than ordinary DNS

The algorithm used in the paper is quite simple. In each (discrete) time window, they construct a list of all the destination addresses which were DNS'ed. For each one, they store the unique hosts on the network which performed the DNS. Then for pairs of non-overlaping time windows, they check whether any destination address was in both lists, and compute a similarity score based on whether the hosts performing the DNS queries overlap. To spot migrating C&C servers, they also compare destinations which have a similar number of hosts DNSing them.

They evaluate their algorithm by running their own botnet on one of their networks. They find that their false positives are generally caused by either legitimate P2P traffic such as Bittorrent, or mass downloading from sites like megaupload.

My comment: I like this idea, and it will also work on modern botnets which use domain generation algorithms to find their command server. However, I doubt it would detect P2P botnets since there is no centralised control point. But the idea that a botnet should involve a roughly fixed size group of computers acting 'in the same way' (in some sense) over multiple time windows, seems generally applicable.

* Wurzinger et al - Automatically Generating Models for Botnet Detection [21]

This paper is very interesting and takes a completely different approach to detecting botnets from the others in this review. It is essentially a signature-based method (i.e. it only detects known botnets), but the authors discuss in detail how to go about automatically generating the rules used to detect malicious signatures.

To do this, they set up a particular computer on their network as a honeypot – i.e. a machine which is designed to be infected by botnet software so that the behaviour of the botnets can be isolated and studied. They download the binaries of 18 major bot worms (e.g. ZeroAccess, Storm etc) from the internet, covering HTTP/IRC/P2P botnets. These are all run on the honeypot so that the machine because infected and joins these botnets. They then leave the honeypot connected to the internet for several days, during which time it will presumably be receiving instructions from the botmaster, and carrying them out. The honeypot will not be doing anything else, so most traffic received is likely to be bot-related. All traffic involving the honeypot is then collected, and analysed for malicious behaviour (e.g. to find whether the honeypot DoSed another computer, or was involved in portscanning or email spam). Whenever the honeypot is found to have doing something suspicious, they check whether there was an incoming instruction received by the honeypot immediately beforehand. If so, then this is likely to be the botmaster instruction. They then save these incoming packets which are identified as botmaster instructions, and use these to generate signature based rules.

They also detect potential botmaster communications in a second way –for each honeypot computer, they wait until it receives a command which is known to correspond to a botmaster instruction (e.g. 'advscan' being a common IRC command). If the honeypot then initiates new connections immediately after the command is received (e.g. if it starts por tscanning a target, or sending spam emails), then this command it is flagged as being a botmaster communication. This requires deep packet inspection to get the commands, as well as knowledge of the particular commands likely to be used.

Once this traffic has been scanned and the packets containing botmaster commands have been identified, they use this information to generate a series of signature based rules for detecting such packets. After they have found all the anomalies + commands they perform clustering to group similar anomalies together, since the same behavior should be triggered by the same command. This allows these bots to be automatically detected if they infect other computers, since their command behaviour is now known.

The experimental results seem plausible, and the deployment on real data seems to suggest a low false alarms rate. They get better performance than BotHunter (an alternative botnet detection algorithm), but presumably this is partly because they are only for the bots they trained their honeypot and detection system on - they would not be able to detect new zero-day bots.

I like this paper a lot – it isn't interestingly statistically, but it seems like the sort of thing that actually would work in practice. It might not detect new unseen botnets straight away, but once a binary is captured and ran on a honeypot, it will be detectable in future. The main limitation in practice is that (depending on the signature-based rules which are learned) it may require deep packet inspection to work, since it needs to gain access to the incoming commands in the potential botmaster packets. But the method seems more sensible than the byte histogram stuff other papers have used for similar purposes. Another major limitation; if the botmaster communicates using encrypted traffic, they may not be able to gain access to the commands.

* Strayer et al - Botnet Detection Based on Network Behaviour [17]

This is a good paper, and I think it's one of the most useful practical papers They give a very detailed description of what they do, particularly how they go about filtering traffic in order to perform large scale data reduction.

They are only interested in detecting IRC bots. Their method is as follows. First, they aim to separate IRC traffic from non-IRC traffic. Then, they try to correlate suspicious flows together, where suspicious here refers to potential C&C communications (rather than malicious bot behavior such as port scans). This is similar to other work I've read on IRC botnets and the immediate difficulty is that distinguishing IRC traffic from non-IRC traffic without using deep-packet inspection (which fails if the data is encrypted) is non-trivial, since the traffic may be HTTP tunnelled or use non-standard ports.

The data set they use is taken from a university network, and they manually inject botnet traffic to give partially truthed data (this procedure seems common in the botnet detection literature). In total they have around 1.4 million netflow records, collected over several months. There is a long section detailing how they go about performing data reduction, which is very useful and provides details about how many flows were eliminated at each stage. First, they focus only on TCP flows and discard all non-TCP traffic. Next, they discard flows containing only TCP packets with SYN or RST flags set, since these are likely to just be port scans. Next, they remove flows containing a large amount of bytes, since C&C flows are unlikely to be small in size. Finally, they throw away both large packets, and very brief flows (i.e. flows which contain less than 2 packets, or last for fewer than 60 seconds). The latter filter has a huge effect and filters out many packets. Overall, they get from 1.4 million flows down to around 36000,

which over several months is not too many. If they used whitelisting and eliminated flows to/from popular websites (e.g. the Alexa 500) like many other papers do, they could probably reduce this number even further.

Next, they take the remaining flows and try to classify them as IRC vs non-IRC. What I really like here is that since IRC flows usually use port 6667, they can simply look at the flows on port 6667 and build a one-class classifier by using these as truthed data. They can this apply this trained classifier to detect potential IRC flows on other ports (since botnet traffic may use IRC on non-standard ports). They use C4.5 decision trees for the classifier, along with Naive Bayes. Table 1 contains the features they use, and they say that average bytes per packet is the best single feature for classification, while others like packets per second, bits per second, and duration are also good. Their classification results aren't great though, and they note that C4.5 seems to overfit based on the training set. This classification procedure could perhaps be improved.

Next, they try to correlate the flows which were identified as being IRC traffic. By this, they mean that (e.g.) if several computers belong to the same IRC-based botnet, then when the IRC server sends a $C\&C$ communication, all these bots will react simultaneously. So there will be a tight correlation among bot-related flows. Similarly for P2P botnets there will be causal relationships ('stepping stones') where events on one flow cause those on another (i.e. bot A contacts bot B, which contacts bot C).

Although they have 36000 flows, these are collected over several months and not all are active at the same time. They pick a period where the injected botnet was known to be active, and this contains 95 potential IRC flows (after filtering), of which 20 are bots. They then look for flow correlations. Section 5.1 discusses useful features for this purpose. It is important to note that flows are dynamic - they have temporal duration (unlike packets) and so can have characteristics changing over time. They pick out characteristics like packet inter-arrival times, inter-burst times, bytes per packet, periodicity, and so on. Since these are not using payload information (i.e. the user-contents of TCP packets), the fact that payloads are often encrypted doesn't matter. They describe several existing papers which also investigate correlation between flows, these should be worth reading ([37, 35, 33, 3, 36, 9], but especially [28]. They construct time series of various flow features, and define a distance function between them. They then look at which pairs of series have a low distance. This part is less convincing, they should probably be using clustering instead. However figure 6 shows it does work.

Question: how do they inject the botnet? I don't think this is described in detail. Are the injected bot hosts honeypots which ONLY perform bot-related behaviour (which should be fairly easy to detect), or are they adding the bot traffic into otherwise normal computers (which is more realistic, and harder to detect)

Feily and Shahrestani - A Survey of Botnet and Botnet Detection [5]

This paper gives a review of the botnet literature and so is a useful introduction. However it doesn't describe methods in much detail, and misses out several papers which I think are interesting.

The taxonomy used in this paper classifies bot detection techniques into four categories: signature based, anomaly based, DNS based, and data-mining based

- Signature-based methods (such as Snort) contain a list of rules which describe known attack types. Bots are detected by matching flows against these rules (e.g. certain bots are known to connect only on certain ports). These approaches are limited in that they can only detect known bots, rather than zero-day attacks.

- Anomaly-based methods passively monitor traffic to detect unusual behaviour which is indicative of bots performing attack. There are not many citations given for this type of detection procedure.

- DNS-based methods are very similar to anomaly based methods, however the authors make a distinction in that DNS-based methods are looking for anomalies which may relate to *C&C* behaviour, while anomaly-baesd methods are looking for suspicious actions carried out by the bot (e.g. DoSing a target). Methods such as those based on DNS failures belong in this category.

- Mining-based methods also look for C&C traffic. However such C&C communications are typically only a very small part of the overall traffic flow, therefore anomaly based methods may not be able find it. Mining-based method looks for C&C traffic more explicitly, for example looking for IRC connections, strange server ports, etc.

Table 1 gives a nice taxonomy of methods. None seem to work in real time! They conclude as follows: "According to our comparison, the most recent botnet detection techniques [33, 34] based on data mining as well as DNS- based botnet detection approach in [15] can detect real-world botnets regardless of botnet protocol and structure with a very low false positive rate".

The papers which they identify as especially good are the Gu Botminer paper, the Choi group activity paper, and the Masud multiple log file papers (all three described in this report).

Karasaridis et al - Wide-scale Botnet Detection and Characterization [9]

This is very similar to the Gu Botminer paper discussed above, in that the algorithm has two stages; first using anomaly detection to find potential bots that are performing suspicious actives such as port scanning and email spam, and then clustering these based on their communication (C&C) behaviour. Afterwards, the activity data generated by the hosts is used to find further clusters (recall Botminer performs both activity and communications clustering simultaneously).

Unlike Botminer, they focus on detecting botnets which use centralised IRC command-and-control structures rather than P2P. They use TCP level netflow data rather than packet data.

As in the Botminer paper, they first scan through all flows to find the hosts engaging in suspicious behaviour (e.g. mass port scanning, or making many SMTP connections). These suspicious hosts are then isolated, and their full flow information is examined, with the goal of finding potential communications to the botmaster. This is done by looking for multiple suspicious hosts that have connected to the same remote server (after

whitelisting). They look at the standard IRC port (6667) and also non-standard ports, since bots sometimes tunnel through these to avoid detection. Interestingly, they also look for communications patterns which are suspicious, e.g. periodic communication with a host which may indicate the PINGs sent by an IRC server. For this latter purpose, they assume that each communication is periodic, with some noise component, and estimate the period (noting there may be missing data). They note they could also use k-means clustering on the inter-arrival times to perform quasi-periodic analysis

Next, they analyse the suspicious communication records in more detail. They focus on the remote servers which have the largest number of suspicious hosts connecting to them. For each of these servers, they compute the distance between all the connecting hosts, based on a metric that uses features such as packets-per-flow, flows-per-address, etc. The idea is that for a botmaster, most hosts connecting to it should display similar behavior (an insight also used in several other papers).

Finally, they take the suspicious hosts and try to cluster them based on their behaviour. They define a distance metric based on the ports each host has connected to, and look at how the resulting clusters change over time.

Lu et al - Automatic Discovery of Botnet Communities on Large-Scale Communication Networks [13]

This paper focuses on finding decentralised botnets, and cites Botminer [7] as one of the most promising existing approaches. The method in the current paper is based first on a) classifying all packets based on their protocol, and b) analysing each protocol separately (so HTTP traffic is analysed separately from Bittorrent traffic, and so on). Note that unless it is assumed that the protocol can be inferred from the port number alone (which is unrealistic since bots often use nonstandard ports or mask their protocol), this would imply some kind of deep packet inspection to read the protocol, and could not be done using netflow alone. They note that even given deep packet inspection, protocol classification is very hard due to masking behaviour, and the fact that even legitimate traffic may (e.g.) HTTP tunnel.

For each protocol, bot detection is carried out by examining the byte frequency in the payload (user part of the TCP packets) data, as in the [20] paper discussed above in the worm section. Packets are clustered based on their byte distributions, with the idea being that bot traffic should be clustered together. Interestingly, they assess how likely each cluster is to be a bot by looking at the standard deviation of the byte frequencies of all the members - the idea is that since bots typically behave in a similar way, they will have low standard deviation within their clusters.

My comments: since this method requires deep packet inspection, it is unlikely to be useful, unless it is assumed that the protocol can be inferred from then port number alone.

Musad et al - Flow Based Identification of Botnet Traffic by Mining Multiple Log Files [14]

This paper tries to identify the botmaster running a botnet, with the ultimate goal of disabling it. They assume that they have complete access to each host computer on the network, including all the local applications that it is running, and their start times. For each host, they maintain two logs: a tcpdump which contains all the network packets, and an application execution trace exedump, which logs the start times of all local applications launched on the host. In theory, whenever a bot receives a command from the botmaster, it will typically start some local application to carry out the command. Since automated bots should respond to incoming communications faster than humans do, the time between receiving a command (via the network) and starting an application should be much lower for bot communications.

For each host, the authors look for timing-related correlations between incoming packets (which are potential botmaster communications) and a) outgoing packets, b) new outgoing connections, c) application startups. These packets are then labelled as possible C&C packets. Their approach is flow-based - it is not individual packets which get categorised as C&C, but whole flows.

Essentially, they create a set of features associated with each incoming packets, such as "whether there is an outgoing packet to the same ip:port within 200ms", "time until the response", "size of packet", etc. A support vector machine (SVM) is then used as a classifier to classify hosts as bot vs non-bot. In order to get a truthed dataset, they construct a synthetic dataset using an IRC C&C bot they downloaded. Im not sure whether such a trained classifier would generalise to other (non IRC) botnets...

My comment: While this is a cute idea, I just don't understand how it could be used without getting a huge number of false positives. There are so many legitimate packets which have causal automated effects. Also, getting local data on the applications being run by each host is infeasible in most cases, for both privacy-related and practical reasons.

Zeidanloo and Manaf - Botnet Detection by Monitoring Similar Communication Patterns [23]

This paper starts by pointing out that although the Choi paper discussed above [2] is an interesting idea, it will not detect P2P botnets. This is the goal of the method in this paper. Like the Gu Botminer paper, their approach is based on the insight that bots in the same botnet will share similar communication and activity patterns

Their goal is to identify all flows for each machine on the network which are likely to be P2P traffic. Directly identify particular flows as being P2P is difficult, so they instead try to find the traffic which is unlikely to be P2P, and discard it. For each computer on the network, they first filter out all flows which are communications with popular websites (using the Alexa top 500 whitelist) in order to reduce the amount of data. Then they filter out TCP connections which did not complete the handshake. Finally, they then filter out HTTP and IRC traffic by using deep packet inspection to identify flows as being HTTP/IRC (e.g. looking for GET/POST within the first few bytes). The remaining flows are candidates for being P2P traffic. Note that their approach would not be able to cope with P2P traffic which is HTTP funnelled, since this would be identified as HTTP. Other researchers have worked on techniques to identify P2P traffic which may be better suited for this purpose (for example [22] which is briefly discussed below).

After filtering the traffic flows, their detection mechanism is quite simple – they cluster the remaining flows based on source port, destination port, average number of bytes per second, and average number of bytes per packet. Next, they look for malicious outgoing activity, particularly email spamming and scanning. Scan detection can be done by (e.g.) checking whether a source IP touched more then N ports or M number of IP addresses within a time window, or any other similar algorithm such as Threshold Random Walk (SPRT) or Snort. Spam can be detected by seeing whether the sender is on an email blacklist (e.g. the DNS Black/Black Hole List). But they use a traffic based approach instead where they look for hosts that send an abnormally high number of emails, which is done by monitoring their ports 25 and 587 (email ports)

Livadas et al - Using Machine Learning Techniques to Identify Botnet Traffic [12]

This paper focuses on detecting botnets which use IRC for *C&C*. Their method has two steps; first they pick out the traffic which is likely to be IRC related, and then they look for bot traffic within this. As usual, they look at TCP netflow. They drop packets containing only TCP SYN/RST flags (likely to be scans) and high bandwidth traffic (likely to be p2p). They didn't (but should have) discarded http flows to popular websites as in the previous paper (i.e. whitelisting).

In order to identify IRC traffic from non-IRC traffic among the remaining flows, a Naive Bayes classifier is used with the usual features like bytes per packet, etc. I don't like this, they should use actual features of IRC traffic (e.g. ping periodicity) for more powerful classification. But Table II is interesting nonetheless since it lists the features which seem to do well at discriminating IRC traffic. The actual bot detection performance is slightly unconvincing since they have no truthed data for their classifier.

*** Yen and Reiter - Are Your Hosts Trading or Plotting? Telling P2P File-Sharing and Bots Apart** [22]

This is a very nice paper, and focuses on identifying bot communications within P2P traffic (i.e. how does one distinguish between C&C P2P traffic, and legitimate P2P traffic such as Bittorrent?).

They have a real data set collected from their network which contains known (non-bot related) P2P file-sharing traffic, which was obtained via using deep packet inspection to identify bittorrent/gnutella traffic. They then inject the bottraffic into the network by superposing it with normal P2P host traffic, which I like since it means the bots have both bot-related and benign traffic coming from the same hosts. However their approach avoids the very hard problem of distinguishing between P2P and non-P2P traffic which is very hard in general due to (e.g.) http tunneling.

They then analyse the difference between legitimate P2P file-sharing users and P2P bots in detail. There is a lot of information here so I won't attempt to summarise it, but reading this paper would probably be useful for anyone trying to detect P2P botnets.

4. References

[1] W. Chen and D. Yeung. Defending against TCP SYN flooding attacks under different types of IP spoofing. In *Proc International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies*, 2006.

[2] H. Choi, H. Lee, and H. Kim. Botnet detection by monitoring group activities in DNS traffic,. In *Proc. 7th IEEE International Conference on Computer and Information Technology*, 2007.

[3] M. P. Collins and M. K. Reiter. Hit-list worm detection and bot identification in large networks using protocol graphs. In *Recent Advances in Intrusion Detection*, 2007.

[4] D. R. Ellis, J. G. Aiken, A. M. McLeod, and D. R. Keppler. Graph-based worm detection on operational enterprise networks. Technical report, George Mason University, 2006.

[5] M. Feily and A. Shahrestani. A survey of botnet and botnet detection. In *Proc. 3rd International Conference on Emerging Security Information, Systems and Technologies*, 2009.

[6] Y. Gao, Z. Li, and Y. Chen. A DoS resilient flow-level intrusion detection approach for high-speed networks. In *Proc 26th IEEE International Conference on Distributed Computing Systems,*, 2006.

[7] G. Gu, R. Perdisci, J. Zhang, and W. Lee. Botminer: Clustering analysis of network traffic for protocol- and structure independent botnet detection. In *Proc. 17th USENIX Security Symposium*, 2008.

[8] J. Jung, V. Paxson, A. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proc. IEEE Symposium on Security and Privacy*, 2004.

[9] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale botnet detection and characterization. In *Proc. 1st Workshop on Hot Topics in Understanding Botnets*, 2007.

[10] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: Methods, evaluation, and applications. In *Proc. 3rd ACM SIGCOMM conference on Internet measurement*, 2003.

[11] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks*, 2000.

[12] C. Livadas, B. Walsh, D. Lapsley, and T. Strayer. Using machine learning techniques to identify botnet trafc. In *Proc. 2nd IEEE LCN Workshop on Network Security*, 2006.

[13] W. Lu, M. Tavallaee, and A. A. Ghorbani. Automatic discovery of botnet communities on large-scale communication networks. In *Proc. 4th International Symposium on Information, Computer, and Communications Security*, 2009.

[14] M. M. Masud, T. Al-khateeb, L. Khan, B. Thuraisingham, and K. W. Hamlen. Flow-based identification of botnet traffic by mining multiple log files. In *Proc. International Conference on Distributed Frameworks and Applications*, 2008.

[15] S. Paxson, V.and Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE Transactions on Networking*, 3(3), 1995.

[16] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reversible sketches for efcient and accurate change detection over network data streams. In *Proc. 4th ACM SIGCOMM conference on Internet measurement*, 2004.

[17] W. Strayer, D. Lapsley, B. Walsh, , and C. Livadas. Botnet detection based on network behaviour. *Advances in Information Security*, pages 1–24, 2008.

[18] H. Wang, D. Zhang, and K. G. Shin. Detecting SYN flooding attacks. In *Proc. of IEEE INFOCOM*, 2002.

[19] H. Wang, D. Zhang, and K. G. Shin. Change-point monitoring for detection of DoS attacks. *IEEE Transactions on Dependable and Secure Computing*, 1(4), 2004.

[20] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *Recent Advances in Intrusion Detection*, 2004.

[21] P. Wurzinger, L. Bilge, T. Holz, J. Gobel, C. Kruegel, and E. Kirda. Automatically generating models for botnet detection. In *Proc. European Symposium on Research in Computer Security*, 2009.

[22] T. F. Yen and M. K. Reiter. Are your hosts trading or plotting? telling P2P file-sharing and bots apart. In *Proc. 30th International Conference on DIstribetud Computing Systems*, 2010.

[23] H. Zeidanloo and A. Manaf. Botnet detection by monitoring similar communication patterns,. *International Journal of Computer Science and Information Security*, 2010.