Delft University of Technology

CS4110 Software Testing and Reverse Engineering

Malware Anaysis

z3r0-t0-h3r0-net

Students:	**Student ID:**
Ginger Geneste	4081315
Marijn Goedegebure	4013484
Sille Kamoen	1534866
Harmjan Treep	4011724

February 29, 2016

1 Introduction

For the course Software Testing and Reverse Engineering the students have been asked to summarise a set of papers related to the topic they have chosen. This document will contain all the summaries of both the mandatory as well as the self chosen papers.

2 Mandatory papers

This section will contain all the summaries of the mandatory papers that have been given via GitHub. Three papers had to be selected from the four possible papers. These papers will form the basis of our study.

2.1 Malware behaviour analysis

Author: Wagener et al. [7]
This paper discusses a technique to compare malware and a method to build a tree which shows the development of the malware strains. Deciding whether a piece of software is legitimate or malicious is not discussed, the method can only classify known malware.

The malware is run in a virtual environment with a few mocked network services to interact with, such as a DNS server. The virtual environment consists of a virtual user mode linux on which wine runs in which the malware runs. After a set amount of time the VM is killed. All the function calls are logged and exported after the VM is killed. The paper briefly discusses techniques for malware to circumvent analysis by this method by for example using a sleep of a long time before running or using anti-virtual OS functions. These methods are still successful against this measurement method while it is resistant against most code obfuscation techniques.

All the function calls are logged, so calls used by legitimate programs have to get filtered out. This is achieved by looking at what function is called and where the return address for the function points to. All OS function calls that come from user space are considered to come from the malware.

From these OS function call lists two pieces of malware are needed to be able to be compared. An important property is that the order of calls shouldn't matter much since the order may depend on the scheduling of the operating system. The solution the authors came up with is converting the function call list to a probability distribution of the chance the malware calls the next function without knowing any information about where the malware is currently in it's execution. The Hellinger distance is a function to quantify the similarity between two probability distributions, this function is then used to compute the similarity between malware.

To visualise the similarity between malware can a phylogenetic tree be constructed. This is done by finding the currently most similar pieces of malware and merge them into a new node with the pieces of malware attached as leaves. The new node similarity to all other malware is computed by taking the minimum of the similarity of all actual malware attached below the node.

The methodology has been validated by running a Nepenthes honeypot and applying the methodology on the captured malware. A quantifiable method of comparing similarity functions for malware is not discussed in the paper but several examples have been provided of known malware families that using this method have a high similarity and in the phylogenetic tree they were grouped together in a branch.

2.2 A survey on automated dynamic malware-analysis techniques and tools

Author: Egele et al. [3]
The need for malware analysis results from the fact that malware (i.e. software of malicious intent) is used by people on the Internet to take advantage of legitimate users. Malware is defined as:"Software that deliberately fulfills the harmful intent of an attacker". Software programs, malware included, include a signature. These signatures are nowadays used to detect malicious programs. Anti-virus scanners use a set of predefined signatures to check against the files of a user. Although this approach seems valid, it is currently labor intensive. This makes it both error prone and tedious work. Secondly, the usage of signatures prevents the detection of unknown (new) malicious code due to their signatures not being known, atleast not until the list of known signatures is updated. Following is a list of the currently known kinds of malware and their definitions:

- Worm, "A program that can run independently and can propagate a fully working version of itself to other machines."

- Virus, "Is a piece of code that adds itself to other programs, including operating systems. It cannot run independently, it requires that its host program be run to activate it.
- Trojan horse, "Software that pretends to be useful but performs malicious actions in the background is called a Trojan horse."
- Spyware, "Software that retrieves sensitive information from a victims system and transfers this information to the attacker."
- Bot, "Is a piece of malware that allows its author (i.e., the bot master) to remotely control the infected system. The set of bots collectively controlled by one bot master is a botnet".
- Rootkit, "The main characteristic of a rootkit is its ability to hide certain information (i.e., its presence) from a user of a computer system."

Most malware is useless when it cannot be deployed or run on a system. Infection vectors are possible ways to put the malware in the proper place and to execute it. The paper lists the following vectors:

- Exploiting vulnerable Services over a network. Network services can include vulnerabilities. These network services are often used as a part in several systems allowing for automatic infection.
- Drive-by downloads. Two kinds can be distinguished: API misuse and exploiting Web browser vulnerabilities. A combination of multiple API's can result in introducing exploitations. Web browser vulnerabilities use the same vector as exploitable network services.
- Social Engineering, "All techniques that lure a user into deliberately executing malicious code on her machine, possibly under false pretences, are subsumed as social engineering attacks."

But how to detect whether a program is malicious? As mentioned before, signatures are used to detect malicious programs. But before these signatures can be written an analyst must determine whether the sample poses a threat to users. There are several tools that allow quick and detailed inspection of a program. On the other side are the malware authors that do not want their programs to be detected. This leads to an arms race that on one hand continuously improves the quality of the tools and on the other hand a constant push for evasion techniques.

Malware analysis can be separated into two general directions. Static analysis and dynamic analysis. Static analysis focuses on analysing a sample without running it, dynamic analysis focuses on analysing a sample while running it.

Static analysis relies on the availability of a kind of representation of a program's code. This might not always be available, making static analysis impossible. Secondly, the possible different representations come with different limitations to the information they provide. For example, a binary representation has less information and encrypted code does not allow any access. There exist several techniques that malware authors can use to thwart static analysis, dynamic analysis is seen as the alternative to static analysis that is resilient to these techniques.

Dynamic analysis is a broad research area that includes many kinds of approaches and techniques. The paper includes several:

- Function Call Monitoring, in which the functions a program calls are monitored. To accomplish this, the calls must be intercepted. This proces is called hooking. The result can either be semantically rich or detailed, depending on the API used. Native API returns detailed results, while the API of an operating system results in semantic observations.
- Function Parameter Analysis, aims to read out the actual values that are passed when a function is invoked. This enables to correlate different functions calls on the same object.
- Information Flow Tracking, is the process of tagging data with the goal of tracking the manipulation carried out by a program.
- Instruction Trace follows the machine instructions a program invokes. This information is difficult to comprehend, but may contain information that is not represented in a higher level.
- Autostart Extensibility Points, are possible hooks from programs to use to insert on start behavior. Since much malicious code wants to run continuously, it is good to look at what programs are using these AEP's.

The next section goes into detail about the different kind of strategies there are to implementing a software analysis system. Each strategy, equalling to a different space, has it's pros's and con's. They describe three kinds of approaches:

- Analysis in User/Kernel space
- Analysis in an emulator
- Analysis in a virtual machine,

Another factor to consider when analysing malicious code is the time required to reset the environment to it's starting state. This is necessary because results are only comparable when the starting states are equal.

Network simulation is something that is required more often in newer malware samples. Malware samples require extra downloads, updates or configuration data to be downloaded. Not allowing network access makes samples not perform any malicious attack, rendering the analysis impossible, but allowing full access would make it possible for the malware to infect other systems. This means that network simulation is required, including multiple connected simulated systems.

The malware arms race mentioned before makes the malware authors make it both more difficult to access the malicious code and detecting when the malware is being analysed to stop action. Packer's are used to pack and unpack the malicious code. The code is unpacked at the moment of activation and is stored in memory. There are several other limitations and problems described in the paper. This is followed by a survey of different analysis tools. Each of these tools uses several techniques described in the paper until now. Each paper description includes which techniques are implemented and a small discussion is included about each paper.

2.3 Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering.

Author: Caballero et al. [2]
The paper proposes a method of reverse engineering the protocols used by infected machines within a botnet to communicate with the Command and Control (C&C) servers. When doing this manually, it may require large amounts of time and it is very error-prone. With the techniques proposed by Caballero et al, an automated analysis is done in a fraction of the time, with a much higher accuracy than previous methods. They developed an application called Dispatcher to automate this analysis. However, Dispatcher is not limited to simply analysing and extracting the used protocol. It can also be used to manipulate messages sent and received by malware.

The paper addresses two problems: 1) extracting the message field tree for the messages sent by the application, and 2) inferring field semantics, that is, annotating the nodes in the message field tree, for both received and sent messages, with a field semantics attribute.

Extracting messages sent between the C&C servers and the malware application can be quite challenging. First of all, the content of the message is unknown, as is the structure. Furthermore, messages are often encrypted. Reading the semantics from these messages can therefore be a complicated task. The paper proposes a method to extract both the message structure, content and semantics by using so called 'Message Field Trees'. Instead of intercepting network traffic, the memory buffers are located where the messages are stored before the actual composition and encryption takes place. By combining these buffers into a message tree, the messages can be extracted and analyzed. For the second problem, each individual section of the message tree is analysed. Based on several known factors such as size, keywords and delimiters, each node can be annotated. For this analysis, it is important to note that incoming and outgoing messages contain different information, making the annotation process easier. For example, an IP address is always has a fixed length. More complicated annotations can be added by using the execution trace of the malware and determining which arguments several (known) functions use.

Evaluation of these techniques is done by analysing a well known type of malware: MegaD. The analysis appeared succesful, with a lot of information being extracted by Dispatcher. However, because this was the first time anyone successfully did this, it was hard to verify the correctness of the extracted protocol. Therefore the generated grammar could not be fully checked for correctness. Rewriting of the messages on an infected machine proved successful as well. The authors were able to successfully infiltrate the botnet by manipulating the traffic between the bot and the C&C server.

3 Selected papers

This section contains six chosen papers on the related topic. The topic "malware analysis", and more specifically botnets, has been chosen, the selected papers should support our research for this course.

3.1 Exploring Multiple Execution Paths for Malware Analysis

Author: Andreas Moser et al. [5]
Dynamic analysis techniques such as the techniques discussed by [7] can miss malware behaviour because the actual payload might be triggered by a command from a C&C server or the current month becoming october. This paper discusses a technique to run malware in a VM and try to find branches where a decision is made based on an external trigger and observe both options. This works by tracking which memory values arrive from a external source and when a branch is made based on that information create a snapshot of the virtual machine. After the program closes or the program crashes the machine is reset to the snapshot and the tainted memory values are rewritten to a value that makes the program take the other branch.

This system is implemented by adding a shadows byte to each byte in memory that contains a source label. When an operation is executed on a value that is tainted, a value that originates from an external source, the resulting value also receives a taint label.

This paper introduces a new system that tries to update all previous values of a tainted source when rewriting. It does this by saving all relations between values and constraints on values and running a linear constraint solver. When executing an operation on tainted data such as addition or multiplication the new value receives a new taint label and the relation between the old label and the new label are stored in the linear constraint solver. If a branch is taken based on tainted data the constraint on the labelled value is stored in the linear constraint solver. When trying to rewrite a memory value the constraint needed to take a branch is added and the linear constraint solver is started. The result for each label is written to the memory locations for those labels. If the linear constraint solver cannot find a solution the other branch contains dead code and no input could satisfy the branch condition.

Since a linear constraint solver is used non-linear relations such as binary operations are not represented. Malware can already protect itself against analysis of this kind by using hash functions on tainted values so not being able to represent non-linear operations is not seen as a weakness.

The tests are done using an extension of the QEMU emulator. Every snapshot contains the RAM memory, taint-system, processor registers and the constraint system. To conserve the amount of memory used per snapshot not the entire available RAM space is saved, but only the pages used according to the OS. Swapping was disabled in the OS to make this process less complicated. Since only user memory is reset the OS is not reset for every snapshot. This introduces a memory leak because allocated memory after the restore point is not freed, no problems with this behaviour were observed during testing. File handles are another problem, if a file handle is closed after a restore point can the program not use the file handle anymore. This problem is solved by ignoring any close file requests from the program.

An important optimisation is recognising string comparison and rewriting the string value immediately instead of letting the problem take a branch for each byte and rewriting values per byte.

Using external sources does not make sense after the malware is reset. For example using an IRC C&C server does not work when the program state is reset, likely breaking the protocol. The same is valid for writing/reading from any other files. Instead no actual file operation are executed but the program is fed random data that gets rewritten to the used protocol.

The system was tested using 7 different sources of tainted data:

- Check for internet connectivity
- Check for mutex objects
- Check for existence of files
- Check for existence of registry entry
- Read current time
- Read from file
- Read from network

A set of 308 malware samples was used. 229 of those used the taint sources described above and 172 of those used tainted information in control flow decisions. About half of the malware samples analysed contains significant hidden functionality missed by simpler analysis techniques.

3.2 Your Botnet is My Botnet: Analysis of a Botnet Takeover

Author: Stone-Gross et al. [6]
The work focus on analysing Torpig specifically, a form of malware designed to harvest sensitive information from its victims. It distributed to its victims as part of the malicious rootkit *Mebroot*. The Torpig characteristics makes it possible to identify unique bot infections with a reasonable accuracy. The botnet is large and targets a variety of applications, gathering a rich set of data from the infected victims. Clients are infected via a vulnerable server which forces the client to visit the drive-by-download server that provides Mebroot. The infected client (bot) will now periodically contact the Mebroot and Torpig command and control server (C&C), for updates and to provide stolen data.

Bots are a type of malware that is written with the intent of taking over a large number of hosts on the internet. Regular analysis have either a *passive* or an *active approach*. This study tries to hijack the entire botnet using a sinkholing method. This is possible due to Torpigs use of *domain flux*, a technique where the infected machine (bot) tries to resolve a list of domains (by using a Domain Generation Algorithm, DGA). The first appropriate response is assumed to be a valid C&C server until the next list has been created. Since the weekly domains are not all registered in advance, the researchers aim to *sinkhole* or hijack the C&C server by registering the domain and forging a valid C&C response.

The researchers registered the domains, set up an Apache web server and collected over 8.7GB of Apache log files and 69GB of pcap data in the ten days they controlled the botnet. Torpis communicates trough HTTP POST requests, where the encrypted header contains all the information of the bot itself. The body will contain zero or more data items that have been stolen (HTTP account, FTP, POP, SMTP etc.).

The researchers also estimated the size of the botnet with regards to its *footprint* (total number of infected clients over time) and *population* (amount of compromised hosts simultaneously communicating with C&C server). To estimate the footprint, the `nid`, assumed to indicate a unique bot, are counted. However, after validating this assumption it appears the `nid` underestimates the botnets footprint. A total of 180.835 `nid` values have been observed. A more accurate estimation was to count the bots by submission header fields resulting in 182.914 bots. Security researchers and other probers have been identified based on invalid requests to the C&C server and configurations of virtual machines. These have been substracted from the total number of bots and thus the footprint was estimated to contain 182.800 hosts. The study also indicates that calculating the footprint based on unique IP's will overestimate the actual size by an order of magnitude in this case but it can be used to closely approximate the botnets size using other metrics such as the median and average size of Torpig's *live* population. The live population per hour can be determined accurately via IP count as the connection with the C&C occurs more frequently than a DHCP churn. New infections have been counted by inspecting the timestamp contained in the initial submission header which has a value of 0. Further analysis indicated the `bld` parameter denotes a build type used for different customers, and would mean Torpig is used as a malware service for third parties who pay a fee to use the botnet infrastructure.

Torpig steals financial data, credentials and also uses the computing power of the bots within the network. A percentage of the bots are marked by the Spamhous project as a verified spam source or flagged as having open proxies used for spam purposes. The aggregate bandwidth for the DSL/Cable connections has been estimated to be 17Gbps, indicating the botnet could also cause a massive distributed denial-of-service (DDoS) attack. The analysis has shown the immense size and power of Torpig, but it has been possible to take over the botnet for 10 days.

3.3 Disclosure: Detecting Botnet Command and Control servers through large scale NetFlow analysis

Author: Bilge et al. [1]
This study presents a large scale, wide area botnet detection system: Disclosure. Groups of features will be identified, allowing to distinguish Command and Control channels from benign traffic using NetFlow records (flow size, client access pattern, temporal behaviour).The system will be evaluated over two large, real-world networks.

According to this study two approaches have commonly been investigated by researchers: *vertical correlation*, where a method tries to detect C&C channels; and *horizontal correlation* where botnet detection is based upon patterns of behaviour. Previous botnet detection methods were only accurate

within a certain domain and thus only small parts of the internet are protected. As data source the researchers have used NetFlow, a data-source collected by large ISPs. Unfortunately the downside of NetFlow is that its records do not include packet payloads, the NetFlow records only contain one direction of a network connection (half-duplex) and NetFLow data are often samples collected at much slower rates than real traffic.

This study will distinguish C&C channels from benign traffic by identifying the flow sizes, client access patterns and temporal behaviour based on the NetFlow data. Disclosure is able to recognize approx. 65% of the known botnet C&C servers, of which only 1% was a false positive.

In order to distinguish benign traffic from actual C&C server traffic, several features were selected. First of all, because Disclosure is explicitly looking for servers, it gathers from the NetFlow data whether a system is a client or a server. Next to that, features are extracted in the following categories:

- Flow Size-Based Features
- Client Access Patterns-Based Features
- Temporal Features

To build detection models for identifying C&C servers, Disclosure uses the random forest classifier machine learning algorithm, as this gave the best results in comparison to other methods like J48 and SVM's.

Due to the nature of the NetFlow data, the chance of false positives was significant. To reduce this risk, Disclosure uses 3 external sources that generate reports of malicious activities on the internet. The services used are FIRE, EXPOSURE and Google Safe Browsing. Each of these sources are given a certain weight, and are combined with results from Disclosure afterwards. This results in a threshold that can be tuned to achieve a low false positive count.

The evaluation of the system is done by applying the application to two different types of network. The first being a university network in Europe, and the second an ISP network located in the USA and Japan. The actual detection consists of two modules: Feature extraction and detection. The detection module performed very well, by going through an entire day worth of data in several minutes. However, this depends on the analysis done by the feature extraction module. Because each feature extraction is an independent process, which can therefore be easily distributed. For the two networks used for evaluation, this wasn't actually required however. On a single machine with 16 cores and 24GB of RAM, the system was able to perform at approximately 2X real-time.

3.4 BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection

Author: Gu et al. [4]
This paper proposes a new way of detecting existing botnets. They propose a detection method that is independent of the protocol and structure used for communicating with the botmaster (the C&C channel) or peers, and is resistant to changes in the location of the C&C server(s). C&C stands for command and control and is a channel of communication through which the bots receive commands and coordinate attacks.

They accomplish this by looking at both who is talking with whom and who is doing what. The information about who may suggest C&C communication activities and what suggests malicious activities. The goal is to find patterns in the extracted information. This is done by applying clustering several times on different sets of data. The system consists of several separated parts:

The C-plane monitor captures network flows and records information on who is talking to whom. The A-plane monitor logs information on who is doing what. The C-plane information is passed to C-plane clustering and the A-plane information to A-plane clustering. First the C-plane data is passed through two filters: basic-filtering and white-listing. The basic-filter filters out all the flows from internal hosts to external hosts. The white-listing uses a list of the top 100 USA and top 100 global most popular websites from alexa.com. The data is further aggregated into communication flows (C-flows). All TCP/UDP flows that share the same protocol (TCP or UDP), source IP, destination IP and port, are aggregated into the same C-flow.

C-plane clustering searches for clusters that share similar communication patterns. First, a clustering is done on a reduced feature space. This feature reduction is done by computing the mean and variance of the distribution of flows per hour, packets per flow, bytes per packet, bytes per second each contained in the C-flow. Afterwards, the result is refined by performing a second-step clustering on each different cluster, using a simple clustering algorithm on the complete description of the C-flows.

The clustering used is the X-means clustering algorithm. It is different from the popular K-means clustering, it does not require the user to choose the number of final clusters in advance. The algorithm runs multiple K-means internally and performs cluster validation to compute the best value for K.

Separately of the C-plane clustering, is the A-plane clustering. This is a two-layer clustering on the activity logs. Given the list of clients that have performed at least one malicious activity during one day, it is clustered on the different types of activity. For each activity type, another clustering is applied according to specific features associated with the kind of activity.

The last step in the process of detecting is the cross-plane correlation. Two planes are intersected and evidence of a host being part of a botnet is combined. For each host a bot net score is calculated for which atleast one kind of suspicious activity was detected. Hosts below a threshold are filtered out and the remaining group is judged using a similarity metric that takes into account the A-plane and C-plane clusters these hosts have in common.

The resulting system is independent of the protocol and structure used for communicating with the botmaster (the C&C channel) or peers, and is resistant to changes in the location of the C&C server(s). It is independent of the content of the C&C communication. That is, it does not inspect the content of the C&C communication itself, because C&C could be encrypted or use a customised (obscure) protocol. It generates a low number of false positives and false negatives. The analysis of network traffic employs a reasonable amount of resources and time, making detection relatively efficient.

To evaluate BotMiner detection framework and prototype system, it was tested on several real-world network traffic traces, including both (presumably) normal data from their campus network and collected botnet data. BotMiner performs quite well in their experiments, showing a very high detection rate with relatively few false positives in real-world network traces.

3.5 An Advanced Hybrid Peer-to-Peer Botnet

Author: Wang et al. [8]

While a lot of papers go into detection of botnets, this paper goes into detail on what the next step in botnet development could be and how this can be countered. They consider research into the threats of today to be equally important to the future developments of botnets. The first to battle the techniques used today and the other to battle future botnets. In current botnets the command and control (C&C) servers, which are needed by the botmaster to instruct the bots, are the weakest link. The C&C servers can quite easily be shutdown since they are limited in number and crucial to the process. Secondly, defenders can easily obtain the identities of the C&C servers by capturing a single bot. Thirdly, an entire botnet may be rendered useless when a single C&C server is captured or hijacked. It makes sense for botnet creators to steer towards peer-to-peer technology, but this step is not a trivial one and current implementations include many weaknesses.

Proposed architecture

Two classes of bots can be distinguished in the proposed architecture. The first group consists of bots that have static, non-private IP addresses and are accessible from the global Internet. These bots are called servent bots, since they will behave as both client and server. The second group contains the remaining bots, called client bots, since they will not accept incoming connections. A botmaster can inject a command through any bot in the botnet. Both client and servent bots actively and periodically connect to the servent bots in their peer lists in order to retrieve commands issued by their botmaster.

The proposed P2P botnet is an extension of a conventional C&C botnet. The servent bots take the role of C&C servers, and thus the number of C&C servers is greatly increased. The proposed botnet requires a more robust and complex communication protocol compared to a C&C botnet.

Strong command authentication is necessary because commands can be injected by each bot. A public/private key authentication is sufficient. The public key is pre-generated and hard coded into a bot.

Each bot also uses symmetric key authentication for his peer list. This guarantees that if defenders capture one bot, they only obtain keys used in the captured peer list. The encryption of the remaining botnet is not compromised. The peer list also contains a port that is associated with a certain address. This port is randomly selected to prevent defenders from scanning specific ports.

A botnet should be difficult to monitor by defenders, but easy for a botmaster. A botmaster has access to a report command that instructs every bot to send it's information to a specified machine. A changeable sensor prevents defenders from quickly knowing the identity of the sensor. This security can be further increased by implementing a probabilistic report. A small probability p specified in a report command to decide whether to report. The bot net has roughly X/p bots if X bots report.

Botnet construction Basic construction of a botnet is done by building peer lists during propagation. The initial set of bots should contain servant bots whose IP addresses are in the peer list of every initial bot. Propagation procedure:

- Bot A passes its peer list to vulnerable host B. If B is a servant host, A adds B into it's peer list. If A is a servant bot, B adds A into it's peer list.
- If reinfection is possible and bot A reinfects bot B, bot B will then replace a number of randomly selected bots in it's peer list with bots from peer list provided by A.

The results of simulation experiments showed that a botnet constructed with the above two procedures is not robust enough. A more advanced propagation procedure is required to improve network connectivity. A peer-list updating command, enabling all bots to obtain an updated peer list from a specified sensor host. Entries of the peer list are randomly chosen from the peer-list updating servant bots. The questions when and how often to run this command remains. The command should be executed once shortly after the release of a botnet. Second, when the botnet is spreading out, each round of this updating procedure increases the connectivity of the botnet, but a risk of being exposed to defenders is associated with running this command.

The paper tested the robustness of a constructed hybrid P2P botnet. Two factors affect the connectivity of a bot net: Some bots are removed by defenders; and some bots are offline. Each factor has a different cause, but their effect on the connectivity is the same. They studied the connectivity of a botnet when a certain fraction of peer-list updating servant bots are removed. It shows that the proposed botnet is robust to removal of servant bots.

Defense against Hybrid P2P botnets If the botnet is not able to acquire a large number of servant bots, the botnet is degraded to a traditional C&C botnet, which is much easier to shut down. Defenders should focus their efforts on computers with static global IP addresses. Secondly, quick detection and response systems can prevent hybrid P2P botnets from expanding by shutting down the initial set of servant bots. Lastly, defenders can poison the communication channel by using honeypots. If honeypots join the botnet and claim they have static global IP addresses. This will make them occupy positions in many peer lists, decreasing the number of valid communication channels.

Regarding monitoring, defenders can effectively use honeypots. If the bot program cannot detect the honeypot and passes it peer list in each infection attempt, the defenders could get many copies of peer lists, obtaining information on many servent bots in the botnet. Secondly, honeypot bots would allow the defenders to determine the plain text commands issued by the botmaster. Once the meaning of the commands is understood, the sensor machines can be found and the entire report might be collected. It would also possibly allow the defenders to know the target in an attack command so that adequate countermeasures can be made.

A botnet is proposed that uses hybrid peer-to-peer, resulting in a botnet that is harder to hijack, shut down or monitor. It provides robust network connectivity, individualised encryption and control traffic dispersion, limited botnet exposure by each bot, and easy monitoring and recovery by its botmaster.

Mathematical analysis is done to estimate the effects of defensive measures against the botnet. It scores highly, as was guessed, on the removal of botnets from the network.

3.6 Botnet detection based on traffic behavior analysis and flow intervals

Author: Zhao et al. [9]
The work of Zhao *et al.* proposes a new approach to detect botnet activity based on traffic behaviour analysis by classifying network traffic using machine learning. The method does not depend on the payload and will thus also work on encrypted network protocols.

Traditional botnets (early 2000s) interacted over Internet Relay Chat (IRC) channels where a bot master established IRC command and control (C&C) channels to communicate with the bots. However, the entire botnet can be disrupted by shutting down the IRC server and its communication was easily monitored. Peer-to-Peer (P2P) networks have also been utilised as botnets. This architecture has no centralised point as a bot can act as both client and server. P2P main challenge is the latency of bot synchronisation. Recent developments in C&C schemes are based on HTTP traffic by hijacking a legitimate communication channel and its traffic is often encrypted to avoid detection. The phases of infection include the *formation phase* (attacker exploits a vulnerability), *C&C* phase, attack phase (when the bot is actively performing malicious activities) and post-attack phase (where a bot is commanded to update its binaries to improve functionality). While existing studies typically detect bot activity only in the initial formation phase based on existing signatures, this work will also present a method to detect a botnet in the C&C phase at the level of TCP/UDP flow by extracting a set of attributes from multiple time windows.

The assumption is made that if there exists a unique signature for the flow behaviour of a single bot, it can be used to detect many bots which are part of the same botnet. Therefore the network

flow characteristics of network traffic will be analysed specifically where a flow is defined as a collection of packets exchanged between two unique IP addresses using a pair of ports within a given time window which utilises one of the Layer 4 protocols. Based on the extracted attributes, the traffic will be classified. A set of 12 attributes are selected based on various well known protocols and behaviour of known botnets like: size of the first packet, source IP address, port address, number of flows etc. For the classification the decision tree classifier has been chosen as it is very accurate in network detection. The classifier builds a decision tree as predictive model where the leafnodes represent a class or subtree.

Two non-malicious (such as Bittorrent, Skype and e-Donkey) and two malicious datasets (Storm and Waledac botnet data) have been merged into a single trace file, replayed and caputured by wireshark again for evaluation. The framework, implemented in Java, extracts the flow from a pcap file and parses the flows into relevant attribute vectors for classification. Each vector represents a 300s time window in which at least 1 packet was exchanged. For evaluating the accuracy 10-fold cross-validation is used to guard against Type III errors and to get a better idea on how the algorithm performs in practice. The decision tree produce an above 90% detection rate and low false positive rate, indicating there are unique characteristics of botnet flow traffic. The system outperforms BotHunter (v1.6.0.) on this dataset. While the results are satisfying, the system might perform differently to new botnet implementations that is based on other protocols. Therefore continuous refinement of the classifier is recommended.

For new botnet detection, training pcapfiles containing malicious and non-malicious traffic data has to be uploaded to generate a signature for detection. Two classes of HTTP-based botnet frameworks, Weasel (encrypted communication) and BlackEnergy, have been considered to investigate the detection performance on botnet behaviour which has not been served to the system for training. During the test all malicious machines of BlackEnergy and Weasel have been flagged as malicious. However, 82% of the alerts produced with the Weasel traffic were false positives and the test-traffic of BlackEnergy did not contain normal traffic. The challenges with unseen normal traffic (that has similar behaviour as malicious traffic), can be approached by training the detector with the sample of normal traffic. Another approach is to filter out traffic whose destinations are legitimate servers. The last or a balanced approach would have improved the performance of the system.

References

[1] L. Bilge, D. Balzarotti, W. Robertson, E. Kirda, and C. Kruegel. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 129–138. ACM, 2012.

[2] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 621–634. ACM, 2009.

[3] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[4] G. Gu, R. Perdisci, J. Zhang, W. Lee, et al. Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. In *USENIX Security Symposium*, volume 5, pages 139–154, 2008.

[5] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.

[6] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 635–647. ACM, 2009.

[7] G. Wagener, A. Dulaunoy, et al. Malware behaviour analysis. *Journal in computer virology*, 4(4):279–287, 2008.

[8] P. Wang, S. Sparks, and C. C. Zou. An advanced hybrid peer-to-peer botnet. *IEEE Transactions on Dependable and Secure Computing*, 7(2):113, 2010.

[9] D. Zhao, I. Traore, B. Sayed, W. Lu, S. Saad, A. Ghorbani, and D. Garant. Botnet detection based on traffic behavior analysis and flow intervals. *Computers & Security*, 39:2–16, 2013.