Botnet detection using Hidden Markov Models

Master Thesis

Networks and Distributed System
Egon Kidmose (14gr1021)

Aalborg University
Department of Electronic Systems
Fredrik Bajers Vej 7B
DK-9220 Aalborg

This work is written with LaTeX in GNU Emacs, on Linux Mint 15. Illustrations, when no reference is made, are created in TikZ or using Inkscape. Work from Open Security Architecture is used in some figures, so in accordance with their licensing the following link is provided: `http://www.opensecurityarchitecture.org/cms/about/license-terms`. This work includes no improvements over the original work of Open Security Architecture. All practical work on the project is done with open-source software under Linux Mint 15 or Ubuntu 10.04. In particular Python has been used for the majority of the implementation and Snort has been relied on as a critical component. Microsoft Windows XP has been used only as a bot infection victim. During this project, no execution of any malware has been performed outside isolated, controlled environments.

AALBORG UNIVERSITY
STUDENT REPORT

Department of Electronic Systems
Fredrik Bajers Vej 7
DK-9220 Aalborg Ø
http://es.aau.dk

Title:
Botnet detection using Hidden Markov Models

Theme:
Master Thesis

Project Period:
4th semester
Networks and Distributed System
February – May, 2014

Project Group:
14gr1021

Participant:
Egon Kidmose

Supervisors:
Matija Stevanovic
Jens Myrup Pedersen

Copies: 4

Page Numbers: 88

Date of Completion: June 2, 2014

Abstract:

Based on a study of the botnet problem and related work solving it, a novel method to solve the problem is proposed. The method encompasses a life-cycle model for a host machine becoming infected with bot malware and being part of a botnet. It is argued that Intrusion Detection Systems can be used to obtain alerts conveying information about the unknown life-cycle state of hosts. The life-cycle model with unobservable states and the alerts related to states fits perfectly with a Hidden Markov Model. It is shown that the life-cycle, the alerts and the Hidden Markov Model can be combined to estimate the life-cycle state of hosts, only relying on data observable in the network. The result is a true positive rate of 100.000%, a false positive rate of 1.068%, yielding an accuracy of 98.947%, on the detection of host with a bot malware infection.

Preface

This master thesis is written on the 4th semester of the programme Networks and Distributed Systems at Aalborg University. Much to my satisfaction, the proposed method turns out to perform very well, naturally under the given evaluation conditions. Apart from the inherent technical challenges, which are expected to occasionally block a process like this, it has been a smooth and educating ride. Cooperation with supervisors has functioned very well and been according to my expectations. Being my first project working alone, some new experiences has been made. One being the sense of how much work is required and how much this report appears to cover. Another being the experience that when I am stuck on something, everything grinds to a halt. I found the fact that no progress was made to have harder impact on my morale than I anticipated. From this I seem to have gained more satisfaction and motivation from sharing the success of group mates in earlier projects, than I have realised. Considering what I have learned both in the technical and project domains, the performance results and, most importantly, this report, I am very satisfied with the project.

Reading instructions

The report is structured to document the process with relevant background and a proposal of a method in the introduction. Relevant, already established methods and the proposed method is described in a chapter dedicated to methods. For the evaluation of the method data is obtained as discussed and documented on the third chapter. The evaluation provides results, also covered in a chapter on its own, Finally, ideas for and thoughts on future work is presented, before the conclusion on the work, each in a separate chapter.

The appendices include a project schedule, code and configurations used during the project, a summary of the the contents of the attached DVD and a list of used acronyms.

Aalborg University, June 2, 2014

Egon Kidmose
<ekidmo09@student.aau.dk>

Contents

1 Introduction

This chapter describes malware and botnets, establishes what is referred to as the botnet problem and presents an analysis of botnet infection life-cycles. Related work is considered, particularly work on detection of bots and maliciousness and work applying Hidden Markov Models (HMMs) for detection. Finally a method based on the above is presented along with a research question.

1.1 Malware

With the continuous introduction of computers in many of the tasks we perform on a day to day basis, both at work and at home, possibilities of digital abuse keeps appearing. As a hardly avoidable consequence, abuses are committed by attackers. Some attacks are performed manually while others are implemented as malicious software, usually shortened to *malware* and used as the term to described any software written to serve a malicious purpose. The first large group of malware to be seen in the wild is called *viruses*, as it infects computers without user or owner consent in a parasitic fashion [Elisan, 2012, chap. 2]. The purpose of a virus is simply to propagate and sometimes also to cause disruptions. Initially the primary vector of propagation was removable media, but later e-mail also became an option.

As the Internet became available to everyone and brought a higher connectivity a whole new propagation vector became available, and so called *worms* appeared. Worms add to the functionality of viruses by spreading autonomously via network connections. While viruses and worms have a negative impact, it is noted that they have limited potential, because they are uncontrolled and simply attempt to propagate and cause havoc. This changed as malware evolved into *Trojans* and *backdoors*, where Trojans are programs that appear useful and benign, while hiding something malicious, such as a backdoor that enables an attacker to remotely control the machine and exfiltrate sensitive information without any user knowledge. This introduces potential for an illicit business, however as only single machines are targeted, this limits the scale and thereby the profit of the crimes committed.

The current state-of-the-art for malware is *botnet* malware - malware used to build botnets. A botnet is a network of robots that serve some master [1]. Some benign examples of use

[1]In some other works a bot is referred to as a zombie or drone, the network as a herd or an army and the master as the herder.

exist, but for the remainder of this work the term will be used for the malicious ones, hence a botnet is a network of compromised computers controlled by a master to achieve some malicious goal. Bot malware is distinguished from viruses and worms by the ability to be controlled, from trojans by typically not being observed by the user. It is distinguished from backdoors by the one-to-many relation between the attacker and the infected hosts. This indicates potential for much larger illicit business, and is therefore deemed a larger and more important problem, than the other discussed types of malware. The following section provides details on why botnets and therefore bot malware is a problem.

1.2 The botnet problem

Being in control of a botnet, a botmaster has a wide set of options for making profit. All of the activities are at least dubious due to the lack of agreement from the owner of the infected systems, but they are often also illegal as they are the digital equivalent of theft and fraud. Examples of illegal, profit generating botnet activities include:

- Theft of personal information and credentials for abuse in e.g. on-line banking or credit card fraud.
- Theft of user account credentials to enable spamming.
- Theft of e-mail addresses from address books to address spam.
- Fraud with pay-per-install or ad clicks, where bots are used to generate fraudulent installs or ad clicks resulting in payouts to the botmaster.
- Abuse of bot resources by sending spam and/or committing Distributed Denial of Service (DDoS) attacks.
- Abuse of bot resources by hosting e.g. malware supporting infrastructure.

In this work *the botnet problem* refers to the fact that botnets exist and are used for illicit activities such as the above, resulting in a negative effect on society. To gauge the size of the botnet problem and justify an effort to solve it, the size and financial impacts of botnets has been studied and is summarised in the following.

The work of Bauer et al. [2008] estimates the direct damage of malware across the globe to have been 14.2 billion USD in 2005 and 13.2 billion USD in 2006. While those are big figures, no information is available on how much is a result of botnet activities.

The cost incurred from spam is estimated by Bauer et al. [2008] to have been 100 billion USD in 2007 and Silva et al. [2013] claims that the majority of spam messages are sent using botnets, suggesting botnets cost at least 50 billion USD a year by sending spam alone. According to Messmer [2012] the Eurograbber botnet, which is built from the widespread Zeus bot malware, was used to steal 47 million USD from 30.000 private and commercial on-line banking accounts. A collection of similar attacks in Denmark resulted in 700, 000 kroner being stolen from 8 users through their on-line banking [Thomsen, 2012]. Globally the Koobface botnet has been estimated to include 800, 000 bots, of which 10, 000 was located in Denmark[Pedersen, 2012]. For the ZeroAccess botnet the number of bots in Denmark alone has been reported as $30,000$ [Kruse, 2012]. Examples of botnets being employed to perform DDoS attacks against governments and national infrastructure also exist [Stringer, 2008][Meister, 2013].

Beyond the estimates and single examples, the recent years have brought reports of a well established criminal industry and a market of specialised services, based on the capabilities of botnets and the profit from their abuse [Schipka, 2009]. Similar to the non-criminal concept of Software-as-a-Service (SaaS) criminal vendors are providing Malware-as-a-Service (MaaS), which underlines how well established the industry is and therefore how eminent the botnet problem is [Gutmann, 2007].

Considering the existence of the botnet problem sufficiently supported, some attention is now given to the nature of botnets to gain an understanding of how they function.

1.3 Botnets

As stated earlier botnets are networks of computers working together, although towards a malicious goal and without user consent or knowledge. This fits with the well established theory of distributed systems and the botnets are, like other distributed systems, characterised by the architecture, namely the topology of the botnet and the protocols for establishing and maintaining a Command and Control (C&C) channel. Furthermore, to avoid blacklisting, take down and legal consequences botmaster employ what is referred to as resilience techniques when implementing botnets. In the following the topologies, protocols and resilience techniques used in botnets are discussed.

1.3.1 Topology

Centralised

The first bot malware connected to an Internet Relay Chat (IRC) server and waited for commands to be issued from the master in the form of chat messages. Such a centralised topology is depicted in Figure 1.1 in the form of a star topology. The characteristic feature is the central C&C server, which results in a single point of failure and a lower resilience than the following topologies. [Bailey et al., 2009]

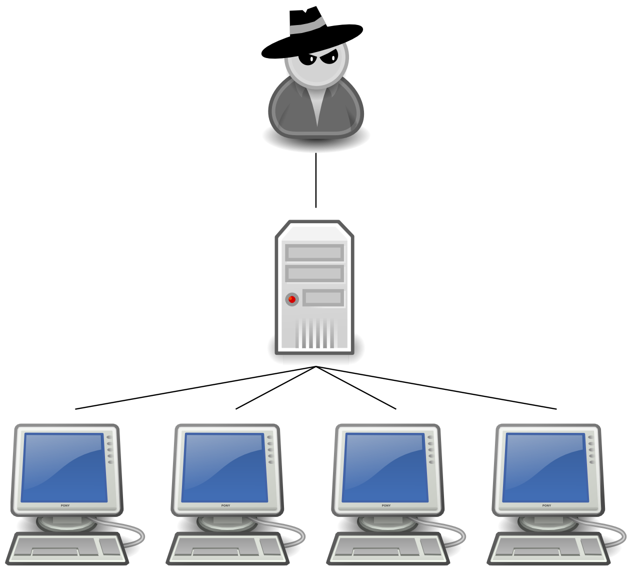


Figure 1.1: Botnet with centralised topology

Decentralised

The counterpart to centralised are the decentralised topologies, building on the ideas of Peer-to-Peer (P2P). Bots avoid dependence on a server by not assuming to be clients, but instead they exchange commands and information with peers. The botmaster can rely on any single bot for injecting commands, as illustrated in Figure 1.2. This makes the C&C protocol more complex as discovery methods become necessary and raises the issue of trust among peers. The fact that P2P does not depend on centralised resources improves the resilience compared to the centralised topologies, at the cost of complexity and higher, possibly unbounded, response times. [Bailey et al., 2009]

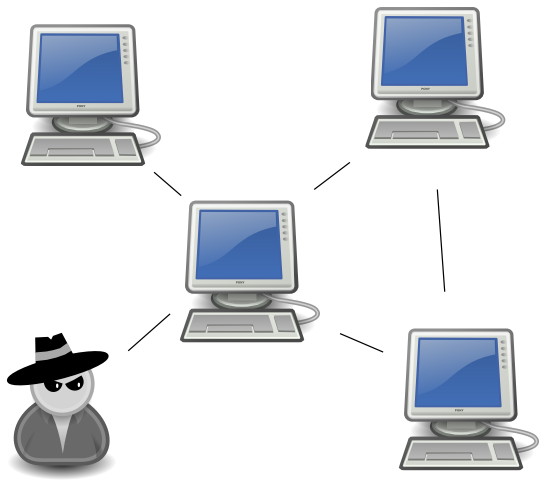


Figure 1.2: Botnet with decentralised topology

Hybrid

While the centralised and decentralised topologies cover most ideas on how botnets are organised many examples do not fit clearly into either and form what is referred to as hybrids. An example of a hybrid is when a collection of peers form a decentralised C&C overlay with a segment of the bots organised under each in a centralised topology. This improves resilience as a take-down might only affect a single peer from the overlay and the bots bellow it, leaving the majority of bots operational. Another example is when parts of the C&C infrastructure, such as seeding lists of peers for bootstrapping bots or drop zones for information, are centralised in an otherwise decentralised botnet thus removing some complexity with a limited cut in resilience as the centralised elements might be replaced. [Silva et al., 2013]

Unstructured/random

In the literature a fourth type of topology is also proposed as unstructured or random, where the master knows nothing about any specific bot, but relies on scanning to discover bots. Silva et al. [2013] mentions the random approach as fully dependent on discovering every bot where Bailey et al. [2009] refers to the unstructured approach where a bot will know a single neighbour. This sort of no-topology is only known from research literature and no implementation is known to have been observed.

1.3.2 Command and Control channel

Besides the topology a botnet can also be described by the C&C channel that makes it usable for the botmaster. Channel and topology are tightly intertwined such that a choice for one affects the other. As an example, the decentralised topology requires a P2P protocol while IRC and Hypertext Transfer Protocol (HTTP) protocols typically are implemented with centralised servers. The following is a discussion of various features of the C&C channel.

Protocol

As already mentioned some botnets use the IRC protocol, possibly due to the ease of use, but because of the rarity of benign IRC traffic this is considered easy to detect, however it is still commonly used. HTTP is seen in big volumes on most networks and many machines are allowed to access Internet services using this protocol. This makes it another common choice for botnet C&C traffic, as it blends into benign traffic. Botnets are also known to connect to established P2P services and use their protocol or establish a P2P network using a custom P2P protocol [Holz et al., 2008]. Examples also exist of botnets using Instant Messaging, the dynamic functionality of third party web services (web 2.0) and other custom approaches for communication. [Hachem et al., 2011]

Persistence of connection

Depending on the C&C protocol used, the channel can be through a persistent connection on e.g. Transmission Control Protocol (TCP), which is established while the bot is running, as seen for some IRC bots. The connection can also be intermittent, such that it is established either periodically or sporadically e.g. for HTTP usage. [Liu et al., 2008]

Initiation
The C&C channels of botnets also differ in how the propagation of commands are initiated. Either the botmaster can push commands to bots for high responsiveness or the initiative can be with the bots such that they pull commands. HTTP with a server in the botnet infrastructure is an example of pull commands, while commands in chat message sent by IRC are push commands. [Hachem et al., 2011]

Direction
Botnets can be implemented such that all communication is one-way, from the master to the bots, allowing for simple protocols but providing no feedback to allow data exfiltration, estimation of botnet size or success rate for attacks. The alternative is a bi-directional channel that provides more functionality. [Hachem et al., 2011]

With the essential features of botnet C&C channels established some attention is now dedicated to the more complex subject of techniques applied in botnets to become resilient.

1.3.3 Resilience techniques

Beyond resilience stemming from the topology, certain techniques can be applied to exploit common network services for yet higher resilience [Ollmann, 2009].

Proxies
A simple way to hide C&C infrastructure is to rely on proxies. By having bots establish C&C connections through proxies, without knowing whereto traffic is relayed, observation and reverse engineering of the bot can only lead to the proxy, hiding the real infrastructure. The proxy can be a bot, but it will require the bot to be accessible from the Internet and it will need a reasonable up time [Nazario and Holz, 2008].

Domain Flux
Domain flux abuses the wild-card functionality of Domain Name System (DNS) to have multiple domain names point to the same Internet Protocol (IP) address. This can be used to defeat e.g. spam filters by using different domain names, although they still resolve to the same IP address. It is also used to identify unique bots on the C&C channel Ollmann [2009]. A more recent variation on this is to use a Domain Generation algorithm (DGA) to rapidly generate new domains to contact. With the bots tolerating that most are unavailable, the botmaster can simply register one when contact to the bots is required. Reactively taking down domains has little use, as new domains are generated before this can be done. Proactively registering or blocking domains requires breaking the algorithm somehow, and yet the amount of domains predicted can be to vast to be of any use. In the case of the Torpig botnet Stone-Gross et al. [2009] did however manage to hijack the botnet by predicting and registering the domains of the DGA.

IP Flux
IP Flux turns the issue around and maps a domain name to one of many IP addresses, through short-lived DNS A-records [Nazario and Holz, 2008]. Having the reached hosts act as proxies to the C&C infrastructure, the latter is well hidden and therefore difficult to take down. The term Fast Flux is used for rapid IP Flux. DNS records with low Time To

Live (TTL) and retry times is typical for this technique. A variation named Double Flux, as opposed to the ordinary Single Flux, exist. Double Flux also abuses DNS NS-records to arbitrate the IP addresses of the names server subsequently queried for the A record. This results in two layers of Flux.

Ollmann [2009] orders the resilience techniques from brittle to very resilient as follows: Single domain, Single Flux, Double Flux and Domain Flux.

Having covered the most important features of botnets, knowledge gained from related work is now presented.

1.4 Related work

As this work evolves around botnets and detection of bot infections, related work on solving the botnet problem has been studied. The findings are organised into a section on existing detection solutions and a section specifically on how others use life-cycles, as this is an essential part of the proposed method. Finally a section is dedicated to how HMMs has been applied to solve detection or classification problems, both for botnet detection and in a wider scope.

1.4.1 Botnet detection

In this section contemporary work on detecting botnets is reviewed. The purpose of this is to establish the state-of-the-art for detection of bot malware infections. Entries are organised according to the kind of information used in the proposed methods.

Host based

Stinson and Mitchell [2007] presents the host based BotSwat which keeps track of network originated information in memory as tainted, observes when tainted information is passed in systems calls and detects infection on calls with more tainted than clean parameters. The evaluation is performed on six bots in a synthetic, controlled environment. BotSwat is able to detect malicious system calls for all six bots, but while some evaluation of false positive are done, no informative statistics on the performance are provided, making any comparison impossible.

BotTracer presented by Liu et al. [2008] is another host based approach, where hosts are cloned at start-up time and the clone is disconnected from any user input. As no user can initiate execution of processes on the clone, a process that starts by itself is considered suspicious. For suspicious processes, certain system calls and connections are monitored and used to detect infections. Furthermore the type of C&C channel is detected as well. 8 different bots are evaluated and successfully detected in a controlled, synthetic network. False positives are acknowledged, but rather than using them as measurable errors, they are continuously eliminated through white listing of the relevant processes. While this might be a valid and reasonable approach in a production setting, it provides no useful information for comparison with other scientific work.

Host and network based

One of the more complex approaches is that of Shin et al. [2013], which uses both host and network information. By correlating DNS queries and user inputs, an initial filtering of processes is done, to sift out most of the benign ones which doesn't initiate connections. For the remaining processes multiple modules are used to feed a correlation engine implementing One Class Support Vector Machine (OCSVM). The choice of OCSVM over an ordinary Support Vector Machine (SVM) is ascribed to the issue of obtaining labelled malicious data. The modules rely on information such as system resource usage, user input events and TCP/UDP connections on the host. In the network, modules rely on Whois, blacklists, search engine results and send/receive balance information. Evaluation is done with as many as 17 bots of various kinds and on substantial sets of benign traffic recorded in real life. All bots are detected and the false positive rate is found to be 0.68% measured as 8 out of 1165 processes, during 46 days of cumulative data on 8 hosts.

Masud et al. [2008] correlates logs for program execution and network traffic from a single host to detect when a incoming packet is followed by a reply to a C&C server, the execution of a program or a new connection to another host. The underlying assumption is that those three events are results of the host being remote controlled. A suite of different machine learning algorithms (Boosted J48, Bayesian Network, Naive Bayes, J48 and SVM) are applied to the timestamped logs and network traces of benign, real world usage and to of two IRC bots (SDBot and RBot) run in a synthetic, controlled environment. The results are good (Accuracy $\geq$ 95.2%, false positive rate $\leq$ 3.2%), however that is when training and testing on a data-set with only one of the bots combined with the benign traffic. As a consequence the results seems to be based on the implicit assumption that a model is trained for every bot to detect only that same bot, thus they provide no indications of future performance as novel bots appear.

Zeng et al. [2010] performs a horizontal detection, meaning that multiple host are noticed to exhibit similar behaviour in network flows, indicating that they receive and respond to the same commands because they are in the same botnet. The decision is done through clustering on host and network flow information. The evaluation includes six different bot types, with four infections for each, executing under controlled conditions allowing them to establish a C&C connection. For benign data around 2000 hosts are included in the data set. With the reservation that it is somewhat unclear how the metrics are calculated, promising results are achieved with a false positive rate of 0.16% and only missing one of 24 hosts infected with a bot, due to the bot not having any communication at all.

Network based

BotHunter, presented in Gu et al. [2007], is the only strictly network based solution discussed here. Relying on three sensors, the Intrusion Detection System (IDS) named Snort and two custom modules for Snort, BotHunter detects infections. The sensors alerts on events in the life-cycle of botnet infections that occur on hosts. A model for infection life-cycles is applied such that only when certain specified pairs of alerts are seen within a time window, a profile is generated and passed on for the given host, signifying a detection. This serves to suppress individual alerts and sequences which are not among those selected for being significant for infections truly happening. The two implemented Snort modules are interesting on their own, and the method clearly has its merits as it detects all 10 bots evaluated in a controlled environment and are capable of detecting a wide selection of bots in both honeynet and real world deployments. A honeynet deployment produced 2019 infections in three weeks, with a true positive rate of 95.1%. The two real world deployments produces some false alerts, but as the traffic is sampled in an unspecified fashion and as the number of hosts are not specified the experienced rates cannot be calculated, although it appears to be low enough for the system to be usable in real life.

Summary
A selection of relevant features for the above papers are presented in table 1.1. In general for the studied work, the specification of how evaluation is done is surprisingly often vague or completely missing [Stinson and Mitchell, 2007] [Liu et al., 2008][Zeng et al., 2010]. Common metrics like rates or counts of false and true positive are often not available or used in a manor which is not consistent with the other studied work. Little is done to validate the results or support the robustness of the methods, as only Masud et al. [2008] applies cross-validation and Gu et al. [2007] performs a solid test in a production environment. All of the referred work use the same bot malware for training and testing, providing little confidence in how the method handles novel bot malware. It requires a sizable effort to obtain bots, which could be a common reason for the limited number of bots used and the omission of cross-validation with different bots.

Author	Evaluation environment			Common metrics	Cross-validation
	Synthetic	Honeypot	Real		
Stinson and Mitchell [2007]	6 bots				
Liu et al. [2008]	8 bots				
Shin et al. [2013]	17 bots		benign only	yes	
Masud et al. [2008]	2 bots*			yes	yes**
Zeng et al. [2010]	6 bots***		benign only		
Gu et al. [2007]	10 bots	yes	yes	yes	

Table 1.1: Overview of related work on bot detection. Synthetic is a completely controlled environment. Honeypot is a host deployed with the intent of observing real bot traffic. Real is when the system is deployed in a real production environment. Common metrics indicates if the work presents any metrics commonly used for evaluation of detection performance. *One bot is picked and used for test and training to obtain results. This is repeated with another. ** However not with different bot malware, but to handle randomness in the applied machine learning algorithms. *** Allowing the bots to establish a C&C channel through the Internet.

The work of Gu et al. [2007], Shin et al. [2013] and Masud et al. [2008] stands out by providing some common metrics, based on which it can be established that they are approximately equally good. Due to difference in the evaluation conditions a single best can not be selected, but these three are found to represent the state-of-the-art for bot detection. None of the other work present significant results close to those three. Having obtained an overview of current solutions and established the state-of-the-art, life-cycle in relation to bot malware infections on hosts is now the focus.

1.4.2 Life-cycle

As bots are abused for illegal purposes, often with the user of the infected host machine as victim, it is in the interest of the user to keep the host machine free of bot infections. As a consequence new PCs are assumed to be free of bot infections and users are expected to remove bot malware, once discovered. The botmaster needs bots in his botnet to maintain a cash-flow. This constitutes a game between the two, which is assumed to cause any host to cycle through a life-cycle of bot infections. In the following different work discussing this topic is summarised and aggregated to a single life-cycle model for a host going through bot malware infections.

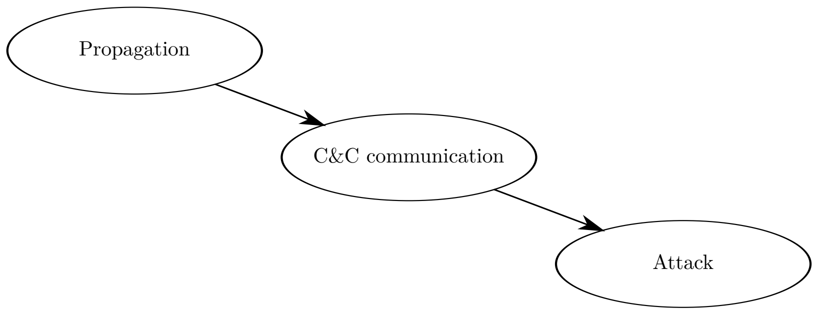


Figure 1.3: Life-cycle model of Bailey et al. [2009].

In the work of Bailey et al. [2009] three invariant behaviours of phases in bot life-cycles are presented: *Propagation*, *C&C communication* and *Attack*, as see on Figure 1.3. The propagation needs to take place to build or grow the botnet and can be detected by observing scans, bot binary downloads or vulnerabilities. C&C communication is necessary for the bot to be controlled and be of any use. It can be detected by its protocols, bots exercising behaviour indicating that they are remote controlled and by contact to a dubious rendezvous point. The attack is invariant because it is what the bot is used for. It can be detected by observing bots participating in DDoS or spam campaigns. Although it is represented as a life-cycle there is no explicit cycle, only a sequence of phases, suggesting that some non-infected state and some removal is implicit.

The work of Hachem et al. [2011] presents a model very similar to that of Bailey et al. [2009], with the most significant difference being the taxonomy; Propagation equals *Spreading/Injection*, C&C communication equals *Control* and attack equals *Application*.

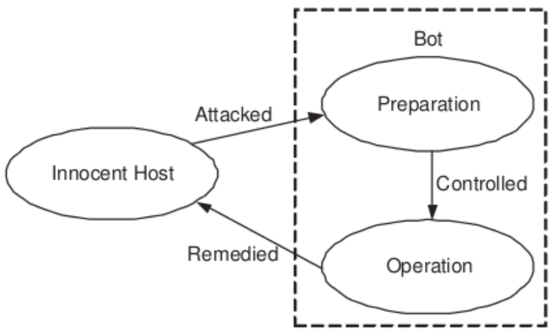


Figure 1.4: Life-cycle model of Gu [2008], the simplest of two.

Gu [2008] presents a simple life-cycle model, seen on Figure 1.4, with the victim host being in an *Innocent* phase until it is *attacked* and starts a *Preparation* phase, which upon it being *controlled* transcends into the *Operation* phase, from which it regains the innocence by being *remedied*. Compared to Bailey et al. [2009] this adds the innocent phase and encompasses the same basic behaviour, although omitting separation of infection and C&C channel establishment. In addition the two appears to have different takes on what to consider a phase and what to consider a transition in the life-cycle.

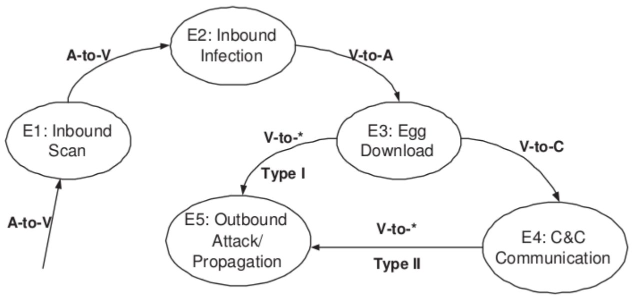


Figure 1.5: Life-cycle model of Gu [2008], the more complex of two.

A more complex model, Figure 1.5, is also presented, similar to what is used in the detection solution called BotHunter [Gu et al., 2007], which considers five phases: An initial *Inbound Scan* for discovering the host that can be turned into a bot by a remote exploit, the *Inbound Infection* followed by an *Egg Download* which takes control of the host and downloads the bot malware (egg), this is optionally followed by a phase of establishing *C&C Communication* before becoming active with *Outbound Attack/Propagation*. It is noticed that this is also not a true cycle and the innocent phase is omitted. This can be seen as an expansion of the simpler model, mapping preparation phase to inbound scan, inbound infection and egg download and the operation phase to C&C communication and outbound attack propagation.

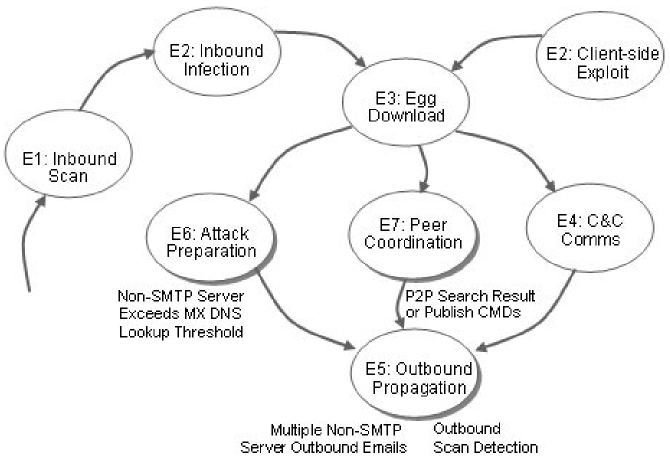


Figure 1.6: Life-cycle model of Porras et al. [2007]

With the Storm P2P bot malware as a case, Porras et al. [2007] extends BotHunter and the used life-cycle from Figure 1.5 to what is seen on Figure 1.6. A notable extension is that the later model includes infections that does not occur through the network, but by *Client-side Exploits*. The model makes a distinction between *C&C communication* in a centralised botnet and *Peer Coordination* in a decentralised botnet. Finally an *Attack Preparation* phase is introduced to capture behaviour that enables the bot to perform an attack.

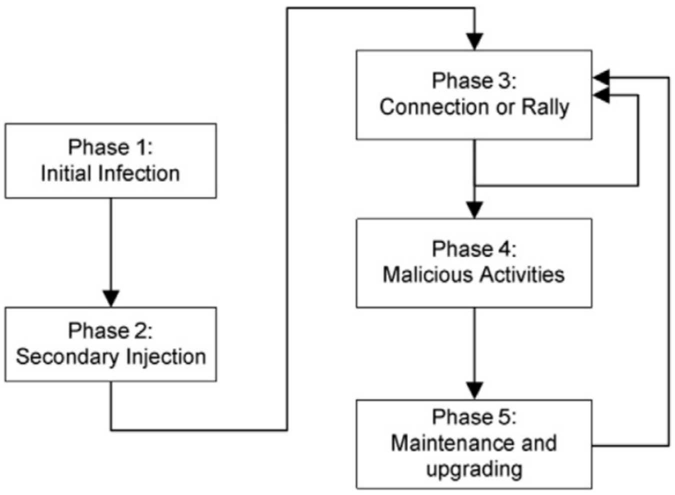

Phase 1: Initial Infection
Phase 2: Secondary Injection
Phase 3: Connection or Rally
Phase 4: Malicious Activities
Phase 5: Maintenance and upgrading

Figure 1.7: Life-cycle model of Silva et al. [2013].

Silva et al. [2013] presents a detailed model, Figure 1.7, similar to Figure 1.5, where the *Initial Infection* phase corresponds to inbound infection or client side exploits, *Secondary Injection* is related to the egg download, as the egg is what is being injected, *Connection or Rally* is the C&C communication and *Malicious Activities* is the equivalent of outbound attack/propagation. This model adds a new phase, namely *Maintenance and Upgrading* and also introduces transitions for re-establishing the C&C channel when a host is rebooted.

Based on the different takes on modelling life-cycles related to bot infections, this section is concluded with an aggregate model, presented on Figure 1.8. Different phases of a life-cycle, represented with ellipses, are defined by some condition that holds for a period of time, until some event breaks a condition and a transitions is made, seen as arrows labelled with the event.

With this work aiming to detect bot infections it seems reasonable to include a *Clean* phase, corresponding to a negative detection result, only seen in the above as the innocent phase of Figure 1.4. [2]

Based on Figures 1.5 and 1.6, the occurrence of an inbound scan finding a host with a *Scan hit*, bringing the host to a *Discovered* phase, has to take place prior to a *Remote exploit* making the host *Infected* with bot malware. Alternatively the host might experience a *Local exploit* making it Infected, known from Figure 1.6 as a client-side exploit. The distinction between inbound infection and egg download (Figure 1.5) or initial infection and secondary injection (Figure 1.7) is omitted, as no documentation was found to support that this fits all bot malware.

When infected the bot might autonomously start performing some malicious activity, such as scanning, participating in DDoS attacks, exfiltrating information or propagating, collectively referred to as an *Attack*, following which it will be considered as being in a phase of *Auto-active*. This corresponds to the fork seen in Figure 1.5.

As the bot is infected or while it is autonomously active it will eventually establish a C&C channel, as all the above models agree, captured by the *C&C Connect*, bringing the bot to a *C&C Connected* phase. When connected the bot is expected to attack at some point, after which it will be considered to be in an *Active* phase.

A "Remedy" transitions from Infected and its subsequent phases to Clean are considered implicit. Set-backs due to host restart or outages in network connection or botnet infrastructure are deemed straightforward as additional transitions. The same applies when the botmaster orders the bot to stop attacking, forcing a transition from *Active* to *C&C Connected*. These transitions are omitted in Figure 1.8 for a cleaner, less cluttered graphical presentation. For the rest of this work no concurrent infections are considered, a bot is assumed to be affiliated with only one botnet and the omissions discussed above are made.

[2] *Negative* being the outcome of a statistical test that no infection is present and *positive* the outcome that an infection is present

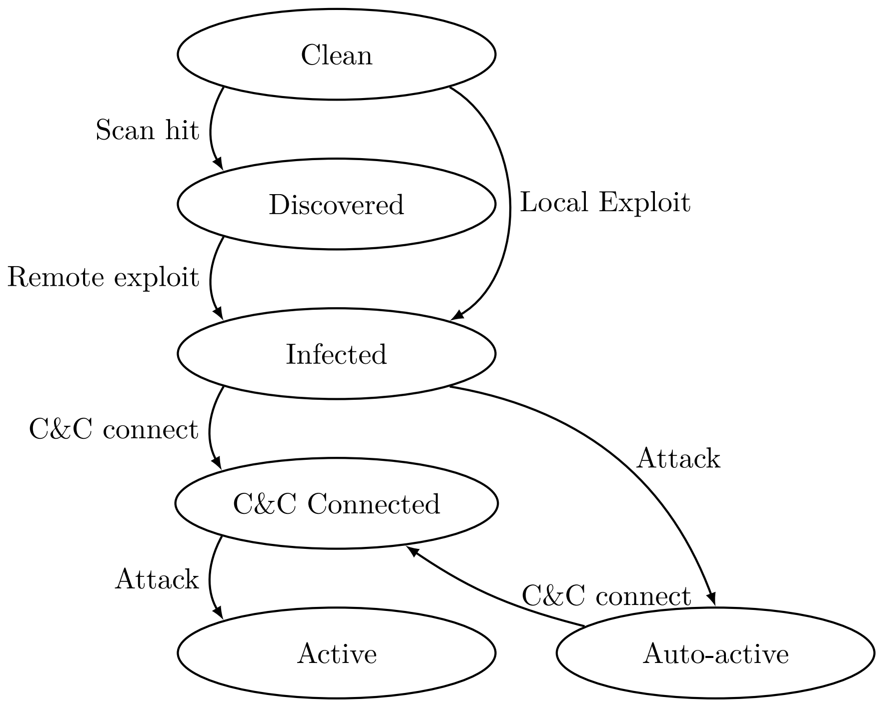


Figure 1.8: Life-cycle model of a host getting infected with bot malware, proposed in this work. Note that transitions caused by host restarts, infrastructure outages and remedy of infections are omitted to provide a simpler representation.

1.4.3 Hidden Markov Model

As the HMM is an essential part of the proposed method, related work relying on HMMs for detection or classification is summarised below. Two of the examples detect attacks through anomalous program execution on a host, five rely on network information to detect or classify bots or attacks in general and the last example is an interesting one from the domain of real life credit card fraud.

Host based

Hoang et al. [2003] seeks to detect anomalies in program execution, by monitoring system calls. A HMM is trained to capture the patterns in sequences of system calls during normal program execution. The idea is then to calculate the probability of an observed sequence of system calls to be emitted by the normal model, and if it is low it is deemed to be an anomaly. The evaluation is limited to show that a signal can be generated on anomalies, while no metrics on detection performance are presented. What can be learned from this work is therefore primarily that the authors also appears to believe in the HMMs capabilities for detection in a security context.

Cho and Park [2003] makes a similar attempt to detect anomalous privilege escalation, but relies on a subset of systems calls that are found to relate to attacks leading to escalated privileges. The authors emphasises that they use relatively small models and short symbol sequences ($N \in \{5, 10, 15\}$, $T \in \{20, 25, 27, 30\}$, $M = 18$) to obtain short computation times. The results include high detection rates of 1 and corresponding false positive rates in the range $[0.00602, 0.2210]$. What is noted as important in this work is the proof that a HMM can be used to detect maliciousness, at least in the given settings.

Network based
Ourston et al. [2004] present work concerned with classifying attacks by 13 different classes, apparently general attacks rather than bots. The authors argue that an attack goes through different phases and is therefore well matched by the HMM. Additionally the presented implementation relies on alerts from undisclosed network sensors as symbols. A HMM is trained for each class, test samples are classified in accordance with the best matching HMM.

Lu and Brooks [2011] train a HMM to detect presence of infections with Zeus bot malware. By separating inter-packet delays into two groups, long and short, and using those two as the alphabet, a HMM is trained and used to determine if an observed sequence of packet delays is produced by a Zeus bot or not. Good results are achieved (detection rate of 0.95 and false positive rate of 0.02) for the one binary in the evaluation. In a following paper the authors apply the same method to an artificial hierarchical P2P bot, created by the authors for the purpose, for which the method fails completely (*".. it fails to accurately detect the traffic data..."*) [Lu and Brooks, 2012].

Venkatesh et al. [2013] performs detection of Spyeye and Blackenergy bot infections, by training a Hidden semi-Markov Model (HsMM) with two states, signifying if the host is under a botmasters control or not. The symbols are based on the Management Information Base (MIB) variables, available from the used network equipment, through the Simple Network Management Protocol (SNMP). No useful details are provided on the evaluation data and the presentation fails to convince that the method is not simply an obscure measurement of TCP throughput, where the used benign and malicious samples are easily separable. If this is the case the solution is expected to be so unstable that it is practically useless in most other settings. It is acknowledge that this might not be the case, but with in the given form this work is of little use.

Joshi and Phoha [2005] trains one HMM for every training sample in the publicly available KDD 1999 data set, using ranges of values for five features of the TCP sessions as symbols. The models consists of a state for every feature ($N = 5$). Detection is done by comparing samples against all the trained HMMs and passing a verdict according to the best matching HMM. An accuracy of 79% is obtained, but no information on how the errors are distribute between false negative and false positives is presented. According to Elkan [2000] the test and training cuts of the used data set contains respectively 79.24% and 73.90% traffic caused by Denial of Service (DoS) attacks. The amount of benign traffic is 19.69% and 19.48%. Taking that into account, the obtained accuracy result is hardly better than that of a rule stating that all traffic is malicious, which obviously is of little use.

Other applications
Srivastava et al. [2008] is the last example op applying HMMs in a security context. It is significantly different from the above, as it is not directly concerned with attacks of hosts, however it remains an interesting example. The authors propose to model every

credit card holder with a HMM, where the states correspond to purchases in different categories and the symbols are ranges of transaction sizes, found by K-means clustering. The authors implement detection of credit card fraud on simulated data, by calculating the probability of the model producing the same symbol sequence as the user, both including and excluding the incoming transaction. If the inclusion of the most recent transaction results in a much less likely sequence, the transaction is deemed fraudulent. The evaluation is done with simulated data. Across multiple instance, tuning the parameters of the HMM, the highest true positive rate is 64% and the lowest false positive rate is 2%.

Summary
To sum up this section, multiple applications of HMMs for detection of maliciousness in security contexts have been made, using either host, network or domain specific features. Some rely on domain specific knowledge to determine their model, others take a zero-knowledge approach. Some estimate states for training data and let the estimated state determine the verdict, while others make use of the probabilities that a trained model emits the sequence of symbols seen in the test data. Some train a single model and others use multiple ones.

The concerns raised above on the work of Venkatesh et al. [2013] and Joshi and Phoha [2005] highlight the importance of well documented evaluation conditions and describing metrics on the results.

With some knowledge on how the HMM framework is utilised for detection in previous work, an outline of a proposal for an approach to be investigated in this work is now presented.

1.5 Proposal of a method

It has been established that the botnet problem exists, a life-cycle of bot infections has been defined and previous work relying on IDS alerts for bot infection detection has been studied along with examples of applying HMMs to solve detection problems. In this section the above knowledge is combined to form the proposal of a method for solving the botnet problem, or parts of it, by enabling detection of bot infections.

The method is to apply the HMM framework to the bot infection life-cycle and alerts related to a given host, in order to estimate the life-cycle phase of the host. The alerts of an IDS form be the observations of the HMM. The states of the HMM correspond to the phases of the life-cycle. Given an observation sequence and a HMM for a machine going through a bot infection life-cycle, the HMM framework enables estimation of life-cycle states, where anything but the clean phase signifies a bot infection.

The project has the hypothesis that the approach will improve the output of IDSs by

i) aggregating alerts on a per host basis,

ii) suppressing false alerts through the HMM and

iii) allowing a ranking of hosts by probability to be in another phase than Clean, enabling an effective and organised effort to combat bot infections.

In order to organise the process of investigating if the approach can be implemented and provide the expected output a research question is formulated in the following.

1.6 Research question

With an outset in the following research question and a set of compositional questions, this work aims to contribute with knowledge on how bot infections can be detected, through modelling their life-cycle and determining their phases using HMM:

Can the framework of HMMs be used to model the bot infection life-cycle of a host, based on IDS alerts, and thereby provide improved detection of bot infections?

The research question is broken down into the following compositional questions:

1. How can IDSs be used to generate alerts related to the phases of a bot infection life-cycle?
2. Can HMMs be applied to estimate phases in the bot infection life-cycle and thereby detect infections?
3. How well does the HMM framework match a bot infection life-cycle?

The following chapters will be concerned with answering these question, with the final result being a verdict on how the performance of the proposed method compares to the-state-of-the-art, as this indicates how useful it is. As a start, the next chapter is concerned with relevant details of the methods applied, both those established by others and the one proposed above.

2 Methods

To support the proposed method, the well established theory of Hidden Markov Models and the commonly deployed concepts of Intrusion Detection Systems are presented. These are followed by a detailed description of the proposed method, which relies on Hidden Markov Models and an Intrusion Detection System.

2.1 Hidden Markov Models

A Markov Model is a stochastic model consisting of a state machine and probabilities for making the transitions of the state machine. The model assumes the Markov property, meaning that the next state only depends on the current state and not those before that. A model fulfilling the Markov property is also said to be memory-less.

Hidden Markov Models (HMMs) is a subset of all Markov Models and are characterised by the actual state and occurring transitions not being observable. To overcome the unobservable information an initial state probability distribution over the states is defined and when transitions are made a symbol from a known alphabet is emitted with some probability.

An example, from Rabiner [1989], of a process that can be modelled with a HMM is a coin toss, where a fair and an unfair coin is tossed interchangeably, but without the observer knowing which. The outcome is observed, being either head or tail. Two states can be used to represent the two coins, with self transitions and transitions between the two states. The self transitions indicate that the same coin is used in the next toss and those between states indicate that the coin is swapped, all with some probability. The transitions to the "fair coin"-state have equal probability of emitting head or tail, while it is skewed for the "unfair coin". This is represented with the state machine in Figure 2.1.

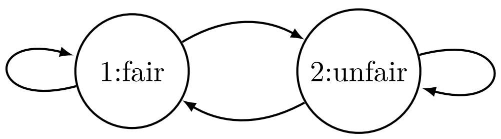


Figure 2.1: HMM for a coin toss with a fair and an unfair coin.

2.1.1 Formal definition

With Rabiner [1989] as the primary reference the same notation and definitions are largely adopted here. The HMM (λ) is a state machine, that starts in one of its states (S), determined by the initial state probability (π) and transitions through the states, with probabilities determined by the state transition probabilities (A). As an addition to the notation of Rabiner [1989] the alphabet of discrete symbols (V) that can be emitted on transitions and the symbol emission probabilities (B) are included in the model.

$$\lambda = (S, A, \pi, V, B) \quad \text{: Hidden Markov Model (HMM)} \tag{2.1a}$$
$$S = \{s_1, s_2 \dots s_N\} \quad \text{: Set of states in state machine} \tag{2.1b}$$
$$N = |S| \quad \text{: Number of states} \tag{2.1c}$$
$$V = \{v_1, v_2 \dots v_M\} \quad \text{: Alphabet of discrete symbols} \tag{2.1d}$$
$$M = |V| \quad \text{: Size of alphabet} \tag{2.1e}$$

The initial state probabilities (π) describe how likely the first state (q_t) is to be a given state (s_i). π must sum to one to be a probability distribution.

$$\pi = \{\pi_1, \pi_2, \dots \pi_N\} \quad \text{: Initial state probability} \tag{2.2a}$$
$$\sum_{i=1}^{N} \pi_i = 1 \quad \text{: } \pi_i \text{ is a probability} \tag{2.2b}$$

The state transition probability matrix (A) encodes the probability of making a transitions from one state to another. For every state, signified by a row in the matrix, the probability of transitioning somewhere must be one. This is sufficient to encode the movement through the state machine as we assume the Markov property.

$$A = [a_{i,j}]_{i,j \in \{1,2 \dots N\}} \quad \text{: State transition probability matrix} \tag{2.3a}$$
$$a_{i,j} = P[q_{t+1} = S_j | q_t = S_i] \quad \text{: Trans. prob. from state } i \text{ to state } j \tag{2.3b}$$
$$\sum_{j=1}^{N} a_{i,j} = 1 \; \forall i \in \{1, 2 \dots N\} \quad \text{: } a_{i,j} \text{ is a probability} \tag{2.3c}$$

On every transition a symbol from the alphabet V must be emitted according to the probabilities in the symbol emission probability matrix (B).

$$B = [b_{i,k}]_{i \in \{1,2 \dots N\}}, k \in \{1, 2 \dots M\} \quad \text{: Symbol emission prob. matrix} \tag{2.4a}$$
$$b_{i,k} = P[o_t = v_k | q_t = S_i] \quad \text{: Prob. for emit. sym. } k \text{ in state } i \tag{2.4b}$$
$$\sum_{k=1}^{M} b_{i,k} = 1 \; \forall i \in \{1, 2 \dots N\} \quad \text{: } b_{i,k} \text{ is a probability} \tag{2.4c}$$

Finally, sampling a HMM produces an unobservable state sequence (Q) according to A and an observable sequence of symbol observations (O) according to Q and A, both with entries at corresponding times from 0 to $T-1$.

$$t \in \{1, 2, \ldots T-1\} \quad \text{: Discrete time steps, observations} \tag{2.5a}$$

$$T \quad \text{: Number of observations made} \tag{2.5b}$$

$$Q = \{q_0, q_1 \ldots q_{T-1}\}, q_t \in S \quad \text{: Sequence of unobservable states} \tag{2.5c}$$

$$O = \{o_1, o_2, \ldots o_{T-1}\}, o_t \in V \quad \text{: Sequence of observable symbols} \tag{2.5d}$$

While a HMM can be defined with continuous and mixed emissions as well as continuous time this is not considered here, as it finds no direct use in the remainder of this work. The same applies for the generalised HMM where some memory is added to the model. Future work could be to investigate if these variation/generalisations could be beneficial.

2.1.2 Typical problems solved

HMMs as can be applied to solve multiple problems, of which some are:

- Given λ, O's that are samples of the process described by the model can be generated
- Given λ and O, the probability that the process will emit O can be calculated efficiently ($P(O|\lambda)$, Problem 1 of Rabiner [1989])
- Given λ and O, the "best" state sequence (Q') can be calculated, with "best" referring to one of multiple optimality criteria (Problem 2 of Rabiner [1989])
- Given λ and O, the probabilities (A, B, π) in the model can be adjusted to maximise $P(O|\lambda)$ (Problem 3 of Rabiner [1989])

As stated, given λ and O the "best" state sequence, Q' can be found. "Best" can be chosen to mean a Q' which maximises the expected number of correct states:

$$q'_t = \text{argmax}_i[P(q_t = s_i|O, \lambda)] \tag{2.6}$$

This can possibly produce invalid Q's, which make transitions that are not possible as this doesn't consider the validity of the sequence $(q_t = s_i,\ q_{t+1} = s_j,\ a_{i,j} = 0,\ \exists i, j)$. Whether this is truly the best Q' depends on the scenario.

Another definition of best could be the most likely valid Q', known as the Viterbi path:

$$Q' = \text{argmax}_Q[P(Q|O, \lambda)] \tag{2.7}$$

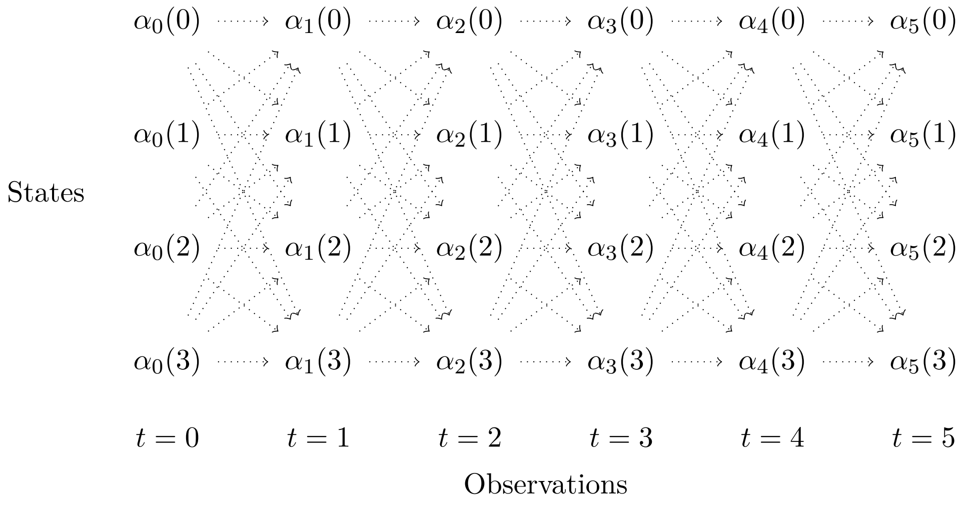
According to Rabiner [1989] what is truly "best" depends on the scenario and could also be some other criterion than those presented here.

For building and training the model S and V must be fixed initially. V must simply contain all symbols that occur, possibly including a symbol to represent any symbol occurring in training but not in test data. S must also be fixed initially and in the found literature the prevalent method is to select a reasonable size ($\{5, \ldots, 35\}$), possibly by cross-validation. The states are often not interpreted as having any meaning in the domain of the modelled system. Some other work apply domain specific knowledge to select the size of S. [Cho and Park, 2003][Hoang et al., 2003][Ourston et al., 2004][Joshi and Phoha, 2005]

The probabilities of λ, namely A, B and π, can be adjusted to train the model, as mentioned in the listing above. No known analytical solution to optimising the probabilities given an observation sequence O exists, and with a finite observation sequence no optimal solution can be found. The best solution available is the Baum-Welch method for finding a local optima, which encompasses Expectation-Maximisation. If training data with known state is available, the probabilities can also be estimated directly using this.

2.1.3 Algorithms

A great benefit of HMMs is that they are computationally cheap in use. Limiting N and T to some constants and keeping the Markov property in mind, a trellis can be constructed to illustrate why, as done on Figure 2.2. States over time are shown as nodes and new observations are made with every new column towards the right. For any state at any time the probability of arriving there only depends on the state distribution probability for the previous observation and the static transition probabilities.



$\alpha_0(0)$ $\alpha_1(0)$ $\alpha_2(0)$ $\alpha_3(0)$ $\alpha_4(0)$ $\alpha_5(0)$
$\alpha_0(1)$ $\alpha_1(1)$ $\alpha_2(1)$ $\alpha_3(1)$ $\alpha_4(1)$ $\alpha_5(1)$
States
$\alpha_0(2)$ $\alpha_1(2)$ $\alpha_2(2)$ $\alpha_3(2)$ $\alpha_4(2)$ $\alpha_5(2)$
$\alpha_0(3)$ $\alpha_1(3)$ $\alpha_2(3)$ $\alpha_3(3)$ $\alpha_4(3)$ $\alpha_5(3)$
$t = 0$ $t = 1$ $t = 2$ $t = 3$ $t = 4$ $t = 5$
Observations

Figure 2.2: Trellis for a HMM with 4 states and 6 observation.

An algorithm named the forward algorithm exploits this structure to efficiently calculate the probabilities of making a given sequence of observations for a fixed model. For the algorithm the forward variable is defined as the probability of making the first t observation in a sequence and being in a certain state, given a model.

$$\alpha_t(i) = P(o_1, o_2, \dots o_t, q_t = s_i|\lambda) \tag{2.8}$$

The forward variable is initialised according to the initial state probability:

$$\alpha_1(i) = \pi_i b_{i,O_1},\ 1 \leq i \leq N \tag{2.9}$$

For all the following time steps the variable can be calculated inductively:

$$\alpha_{t+1}(j) = \sum_{j=1}^{N} \alpha_t(i) a_{i,j}, \ 1 \leq t \leq T-1, \ 1 \leq j \leq N \tag{2.10}$$

The algorithm is terminated by summing the forward variable across all states to obtain the probability of making the given observations from the given model:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i) \tag{2.11}$$

As evident from equation (2.8) the forward variable is the probability that the HMM is in a certain state given an observation sequence and a model. The most likely individual state of equation (2.6) is then determined as the state which maximises the forward variables for a given time step:

$$s'_t = \text{argmax}[a_t(i)] \tag{2.12}$$

Equation (2.11) is the probability of making a given sequence of observations from a HMM. This is used when training a HMM, λ, for comparison against some new sequences. As an example anomaly detection can be done by relying on low values of $P(O|\lambda)$ to indicate an anomaly.

An analogue backwards algorithm calculates the probability of making the remainder of observations from a current time t, given some current state and a model. By combining the forward and backward algorithm and dynamic programming, the Viterbi algorithm can be obtained. The Viterbi algorithm is used to calculate the Viterbi path that satisfies equation (2.7).

This concludes the coverage of HMMs and the next section is concerned with the sources of observations, IDSs.

2.2 Intrusion Detection Systems

Intrusion Detection Systems (IDSs) are solutions intended to raise alerts when intrusions takes place. They are of interest to this work because it is assumed that their alerts in some way convey information about the life-cycle of bot infections.

They are typically deployed by administrators hoping to notice and react to any intrusions affecting their network and the hosts therein, suggesting that useful information is contained in the alerts. IDSs and intrusions cover not only bot infections, but also a wider set of intrusions, such as worm and trojan infections and targeted attacks. A targeted attack is when an attacker focuses an effort at breaching the security of the systems within the network, rather than the less focused and often fully automated spreading of worms and bot malware.

Some IDSs are deployed to the covered hosts and are known as Host-based Intrusion Detection Systems (HIDSs), while others are in the network and are referred to as Network-based Intrusion Detection Systems (NIDSs). HIDSs have the advantage of accessing information internal to the host, but also the disadvantages of requiring to be installed on all covered hosts in order to obtain the information. Furthermore HIDSs rely on proper execution on the host, which might not hold if the host is attacked. NIDSs have an advantage in being deployed in the network, which enables central and simple management and deployment along with possibilities for stealth and passive operation. Their disadvantage lies in less information being available, compared to host-based solutions. Capacity and computational issues might arise for high bandwidth networks.

While both types of IDSs have potential for carrying information about bot infection lifecycles, only network-based IDSs will be considered in the remainder of this work. This choice is due to their simpler architecture, which enables a single host running the NIDS software to cover a full network, while setting up an HIDS requires software on every host as well as a central point for aggregating information. Furthermore, network-based solutions seem to be prevalent in the industry, leading to an expectation of more mature and well-documented solutions. In addition, earlier work suggests that detection can be done in the network, without host-based information[Gu et al., 2007]. For the remainder of this work IDS will refer to NIDS.

With the focus narrowed down to network-based solutions there are still multiple options available, where those known to the author are: Snort, Suricata and Bro. Snort is selected due to the conception that it is the most widely used, best documented and most mature. Snort is also what forms the base of the well regarded Bothunter [Gu et al., 2007]. Appendix D contains further details on how Snort is installed, configured and used. Before using Snort the limitations of IDSs are discussed in the following section.

2.2.1 Limitations

IDSs in general suffer from the base-rate fallacy, as discussed by Axelsson [2000]. The base-rate fallacy is the result of applying Bayes theorem to the typical operation conditions of an IDS; Only little malicious traffic and a much larger amount of benign traffic. A false positive rate that at a glance seems low, might still be too high given a large amount of benign traffic, leading to the administrator of an IDS seeing mostly false positives among the received alerts, hence the fallacy. This would result in the administrator being more likely to ignore alerts and the volume of alerts, which still needs manual investigation, would be too high for the system to be of any use.

The fallacy can be demonstrated by approximate $P(I|A)$, the probability of intrusion taking place when receiving an alert, under the conditions mentioned above, namely the probability of an intrusion taking place tending to zero. Using Bayes theorem, $P(I)$ for the probability of traffic being intrusive, $P(A)$ for the probability of traffic raising an alert and $P(A|I)$ for the probability of seeing an alert on traffic that is an intrusion (True positive rate) the following is obtained:

$$P(I|A) = \frac{P(I) \cdot P(A|I)}{P(A)} \tag{2.13}$$

Using $\neg$ to indicate absence of an event and introducing the probability of an alert without any intrusion $P(A|\neg I)$ (False positive rate), the above can be rewritten:

$$P(I|A) = \frac{P(I) \cdot P(A|I)}{P(I) \cdot P(A|I) + P(\neg I) \cdot P(A|\neg I)} \tag{2.14}$$

By the fact that almost no traffic is due to intrusions, the result is that an alert carries almost no information about whether an incident occurred or not:

$$\lim_{P(I) \to 0} P(I|A) \to \frac{0 \cdot P(A|I)}{0 \cdot P(A|I) + 1 \cdot P(A|\neg I)} = \frac{0}{P(A|\neg I)} = 0 \tag{2.15}$$

Loosening the assumption of $P(I) \to 0$ to $P(i) << 0.5$ equation (2.14) still shows how the false positive rate dominates $P(I|A)$.[1]

This effect can possibly reach a point where the system is unusable, as alerts become frequent and most likely are raised on benign traffic. Therefore IDSs must not only be good at detecting when an intrusion occurs, but also be *very* good at not falsely raising alerts on benign traffic. It is concluded on this result that an improvement on detection performance is desirable. Furthermore it appears that an ability to rank or sort alerts based on a certainty of the alert and aggregating them in a meaningful way will lower the amount of manual processing and make the effort more targeted/efficient.

Another relevant problem for IDSs arises when they rely on Deep Packet Inspection (DPI) for matching strings or patterns of known intrusions against packet payloads. Beyond being computationally heavy, any method relying on DPI is completely defeated by basic encryption. Strings and patterns also tend to be a poor choice for detecting novelties. If the strings or patterns are very specific, many are needed to catch everything and rapid updates are needed as the strings and patterns will be outdated. On the other hand, it becomes hard to avoid false positives with more general signatures. IDS need not rely on DPI alone, as e.g. network scans and send/receive ratio also carries information [Gu et al., 2007]Shin et al. [2013].

With the essential fundamentals of IDSs sorted, an example to convey the high level concepts is presented in the next section.

2.2.2 IDS setup

With an outset in Snort as the state-of-the-art IDS, the relevant documentation has been studied and a typical IDS setup has been established [Sourcefire and Snort community, 2014]. Figure 2.3 shows this typical setup and the following will explain a typical usage scenario.

[1]The derivation in equation (2.13) through (2.15) is based on Axelsson [2000]

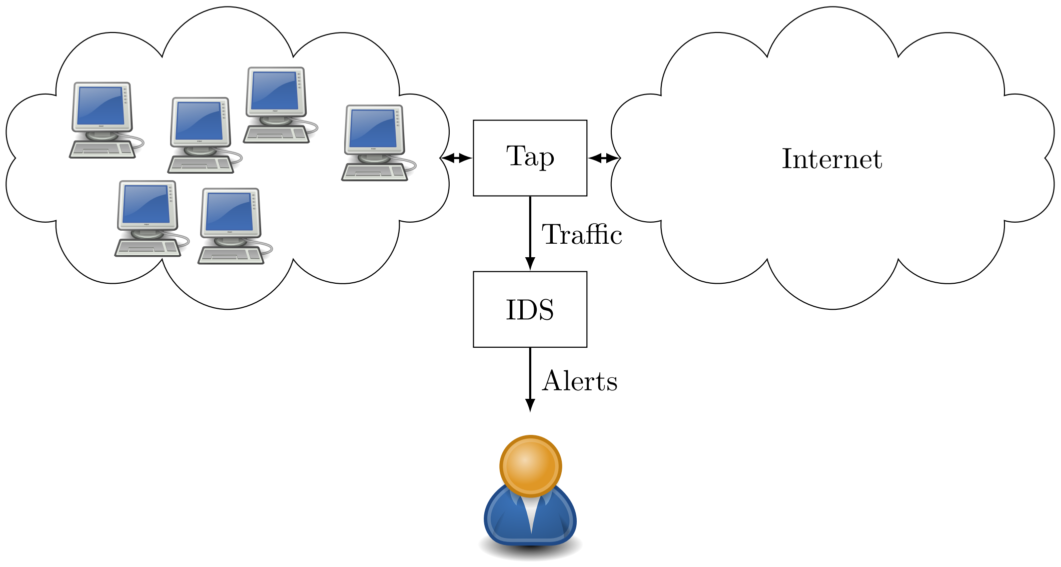


Figure 2.3: Diagram of a typical IDS setup

Figure 2.3 shows a number of hosts on the top left. They are connected in some topology, that can include switches, routers, firewalls etc. The common features for the hosts are their need to access the Internet and the importance of detecting any bot infections on the hosts. Different topologies affect the traffic through switching, routing, Network Address Translation (NAT) fire-walling, tunnelling and encryption, which is assumed to influence the detection task, but this work is delimited by assuming that the IDS deals with such issues. This might not be true, among other reasons, because tracking hosts becomes difficult when applying NAT and because encryption renders payload signatures and any DPI useless. It would at least require a thorough inspection of the individual IDS documentation to establish how it handles such issues.

The traffic between the hosts and the Internet comes through a network tap, which mirrors the traffic and thereby enables the IDS to process it passively. By applying rules to the traffic the IDS determines when to produces alerts. These alerts can be inspected by an administrator to discover infections or be stored and used as a manageable sized trace, as opposed to all traffic, when investigating infections discovered by other means.

The data recorded as described in section 3.3 indicates that an ordinary office user can produce more than a million packets with a trace size of gigabytes every day. Put through an IDS, such as Snort configured as described in appendix D, this results in thousands of alert per non-infected user per day. For the synthetic bot traffic, obtained as described in section 3.2, a few hundred kilobytes of network traffic can contain a bot infection that causes no more than 16 alerts.[2] Discovering bot malware through manual inspection of this is unfeasible, so tools are available for generating reports with aggregated statistics. Though this is apparently usable, alert counts and such statistics are suspected to be a poor indicator of bot infections.

[2]See appendix F for exact figures.

From investigating the capabilities of IDSs it is found that they can generate alerts on suspicious network events, which answers the first of the compositional questions to the research question in section 1.6. With HMMs and IDSs as the essential components, the proposed method of section 1.5 can be specified in further detail.

2.3 The proposed method

The method proposed in section 1.5 fits into the IDS setup of Figure 2.3 between the IDS and the administrator as depicted on Figure 2.4. Grouping alerts by host and applying a HMM trained to match the bot infection life-cycle provides an estimate of each hosts state in a bot infection life-cycle. Beyond enabling the discrete estimation of states, the HMM framework also enables cheap calculations of the probabilities for being in certain states, so a detection threshold can be implemented. The probabilities can also be used for ranking hosts according to how likely they are to be infected with bot malware. This is a much more useful output for an administrator looking for bot infections, compared to the volume of alerts produced by an IDS as discussed in section 2.2.2.

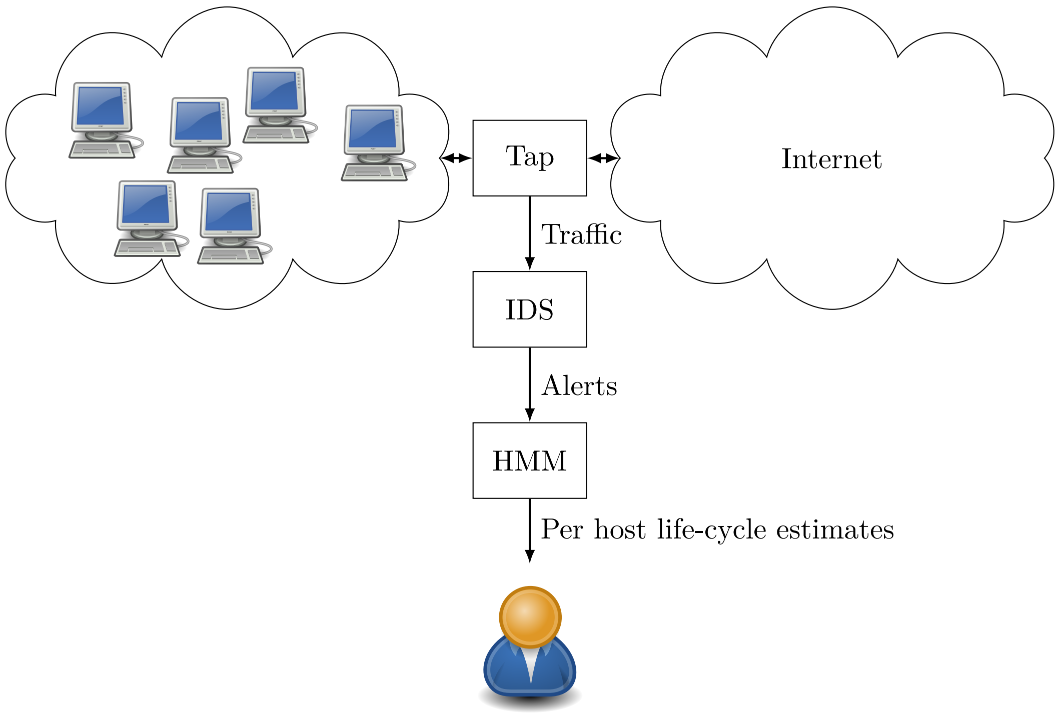


Figure 2.4: Diagram of a setup with the addition a HMM, as proposed in section 1.5.

The method is implemented for off-line execution, as this greatly simplifies implementation, debugging and bug fixing. This is done without any assumptions that prohibits a migration to an on-line solution, see discussion in chapter 5. Each block of Figure 2.4 represents a procedure, which is executed separately. The data passed between procedures is stored, such that any procedures can be repeated on its own.

Designing an approach to obtain network traffic involves a lot of pitfalls and considerations and as a consequence this is omitted in the current chapter but covered thoroughly in the next. For now it is assumed that a procedure exists for generating traffic traces labelled with the true life-cycle phase of hosts seen in the trace. Such procedures are presented in chapter 3. Snort is applied to the traffic traces as described in appendix D and produces alerts. A preprocessing procedure is applied to the alerts and labels to obtain observable events with a known corresponding states. This preprocessing serves to separate practical issues handled in the implementation from the core of the method, which is the application of the HMM. Finally the events and states are used either for building a model or evaluating the model's detection performance.

2.3.1 Preprocessing

Preprocessing consists of the following steps:

1. Load alert list and convert to symbol sequence
2. Load labels and convert to state sequence
3. Combine symbol and state sequence

The data generation procedures of chapter 3 results in traffic traces stored as Packet Capture (PCAP) files, which is processed by Snort to produce lists of alerts. This condenses the information quite significantly, with some loss, according to the inner workings of Snort and the used Snort rule set. In this work Snort is considered a black box, though it is an important component in the proposed solution. Some configuration must be done in order to get it running, as discussed in further details in appendix D, so when a configuration decision is necessary, high sensitivity is chosen. With this decision some extra alerts are presumably generated, which carries little or no information, but as processing some extra alerts has little cost and as alert are aggregated per host before being presented to the administrator, this is a good price for a chance of seeing additional useful alerts.

The output alerts come as lines in a CSV file containing a time stamp, the IP addresses of the source and destination hosts and an ID for the rule that triggered on the traffic. A rule describes certain events/conditions on the network and when they are seen/met an alert is raised. This makes the ID of a rule a good representation of a certain type of network event and it is therefore mapped to symbols. On a side note, tuning and tweaking Snort and writing additional rules is an area with many potential improvements, which is left for future work. The possibilities of extending Snort are supported by Gu et al. [2007] relying heavily on custom modules for Snort.

The alert list is converted to a list of symbols (O) by mapping alert IDs to symbols.

For the ground truth on states (Q) logs and assumptions discussed in chapter 3 are processed/used.

2.3.2 Model building

Model building consists of the following steps:

1. S is chosen according to the life-cycle in section 1.4.2.
2. V is defined as all symbols seen in training data and an symbol referred to as *any other*, for new symbols appearing in test data.
3. π is set to $\pi_{clean} = 1$ and $\pi_{k \setminus clean} = 0$, as we assume all hosts to start as clean.
4. $a_{i,j}$ is calculated as an empirical value from the training state sequences by counting and normalising.
5. $b_{i,k}$ is calculated as an empirical value from the training state and symbol sequences by counting and normalising.

V is found by looking through all emitted symbols, S comes from section 1.4.2 and π is chosen such that a host always begins as clean, as mentioned above. For the data at hand all state sequences does indeed begin as clean, but this might not hold in reality.

With that change it must be considered to build a model with a more appropriate π.

$a_{i,j}$ is estimated from the training data, specifically the number of transitions made from s_i to s_j is counted in all of the test data for every combination of i and j. Under the assumption of the Markov property it is only necessary to look at single transitions between states, as earlier and later states have no influence. In order for $a_{i,j}$ to be a probability each row in A is normalised according to equation (2.3c).

$b_{i,k}$ is estimated in a similar fashion, by counting the number of times a transition from s_i emits v_k, again using the assumed Markov property and applying normalisation (equation (2.4c)), to obtain a valid empiric probability.

2.3.3 Detection

Detection performance evaluation is done with the following steps:

1. Load the model (λ), built as in section 2.3.2.
2. For every host: Estimate a state sequence (Q') from the observation (O) as the most likely individual state.
3. Compare all the estimated states (q'_t) with the ground truth (q_t).
4. Determine detection outcome as positive if the host is estimated to be in any other state than Clean at any time, otherwise negative.

Loading the model is straight forward as it only contains a limited amount of data, nicely organised as matrices (A, B), a vector (π) and lists of possible states (S) and symbols (V).

As discussed in section 2.1 on HMMs states can be estimated in various ways. For this work the most likely individual state is used, as found by the forward algorithm. As for computation this requires one iteration of the algorithm per alert. The memory footprint is a floating point value to hold the forward variable for every state in the state machine for every host, which is a small number that scales linearly with the number of hosts. The forward algorithm also has the benefit of relying only on the past and present observations, making it suitable for real-time implementations.

The detection outcome is determined from the state estimation sequences. This can be done in multiple ways producing a detection outcome on every state estimation, over a time window or for the whole sequence. Relying on a time window, as seen in Gu et al. [2007], introduces a risk of completely missing bot infections with slow life-cycles. As the HMM is memoryless and therefore virtually free to run indefinitely, this is deemed a bad choice. In a scenario where the outcomes are monitored live for rapid responses, a detection outcome on every incoming alert is sensible, as every alert potentially changes the outcome. In the current setting of evaluating the detection performance on data with hosts spanning no longer than a day, detection on the whole estimated state sequence is found suitable.

As discussed in section 2.1 the forward algorithm provides the probabilities of being in any given state, which allows for ranking hosts by the probability that they are not in the Clean state of the infection life-cycle. This probability also allows an adjustable detection threshold, allowing for tuning of the detection. For the current work it is selected to make the detection outcome be determined by the following rule: If the state of a host at any time is most likely to be in any state but clean the host is deemed infected/positive, otherwise it is not infected/negative. Hence, Clean refers to detailed information on the state of the model or the phase in the life-cycle, while infected is used for the less detailed information of whether an infection has occurred on the host or not.

This concludes the chapter on methods. The next chapter describes the procedures for obtaining the data used to evaluate the proposed method.

3 Obtaining data

To evaluate the detection performance of the proposed method a HMM must be trained as described in section 2.3.2 and thereafter tested as described in section 2.3.3. This chapter contains a discussion on how data for test and training can be obtained or generated. Four different kinds of data are considered: Real bot traffic (section 3.1), Synthetic bot traffic (section 3.2), benign traffic (section 3.3) and mixed traffic (section 3.4). Real bot traffic is simply traces of bots in the real environment of the Internet, synthetic data is traffic generated with a bot in a fully controlled network, benign is real traffic with no bot activity and mixed traffic is a merger of the synthetic bot traffic and benign traffic.

Running botnets is an illicit activity and botmasters therefore seek to hide the activities of and information about their botnets, including the traffic they generate. A concrete example is that bots are known to encrypt or obscure their traffic, presumable to avoid discovery and reverse engineering. Another example is that a bot might assume to be running in a honeypot or in other ways be subject of investigations, if it detects that it is executing in a virtual machine or with limited network connectivity, resulting in the bot remaining dormant forever or attempting to remove itself. Bots can also exercise aggressiveness by automatically joining a DDoS attack, presumable to "retaliate" or deter anyone from performing research on the bot Hachem et al. [2011]. The result is that the task of obtaining data on bot malware execution is complicated.

Contemporary research efforts on detecting botnets or maliciousness in network traffic uses labels of either positive/negative values for bot presence/maliciousness or classes, when classifying attacks. This work is concerned with detection, but through life-cycle estimation, hence positive/negative labels are not enough, instead the labels must be on the life-cycle phase. No other work is known to use labels with this detail and as the final life-cycle presented in section 1.4.2 is a product of this work, data meeting the requirements is not expected to exist.

As a result part of this work has been concerned with deciding how to obtain traces, which need to be labelled with life-cycle phases. In the following sections, thoughts on relevant issues are presented, with some solutions.

3.1 Real bot traffic

In order to estimate life-cycles of bot infections and detect their presence, data containing bot infections are necessary. The apparently obvious choice is to record the traffic of bots running in real life, also known as *in the wild*, on real computers, committing real crimes on the Internet. The following advantages and disadvantages for this kind of traffic is acknowledge and discussed in further details below:

- Advantages:
  - Most realistic solution
- Disadvantages:
  - Observing badness without intervening
  - Privacy concerns
  - Obtaining labels
  - Access to data

This approach has the advantage of providing conditions exactly as the intended final ones. The bots in the data are as up-to-date as possible, and while future performance is important, it is inherently impossible to know, so this is the best possible kind of data, with regards to the bots seen in the traffic.

It does however have some disadvantages, one of them being the ethical and/or legal issues of observing something likely illegal and likely to harm an unaware victim, without intervening. It can be argued that it is not an issue, as the attacks take place whether they are observed or not. There can also be a big effort involved with validating attacks and tracking down victims in real life, an effort to big to be considered a real option. It can also be argued that, in a greater perspective, time is better spent solving (a large part of) the problem, rather than single incidents. The act of tracking down someone brings up another issue of a similar kind, namely privacy. Considering the information of a packet to be confidential makes it complicated to record, store or process traffic. User consent might solve this issue, but it will need to cover all the users, whose traffic is recorded. Figuring out what can and can not be done will be a task for lawyers rather than engineers.

Yet another issue is the practical one of labelling the data. Assuming that an infected host also generates much benign traffic, as discussed in section 2.2, this leaves a herculean task of labelling the data. While applying an existing detection solution might enable a rough filtering and lessen the manual effort, errors by the selected solution will introduce noise, and the task of determining life-cycle phases remains, which is unique for this work. Furthermore manual processing is expected to be error prone, again introducing noise in the data. Another very critical problem and possible decisive on its own is where to get access to enough traffic. Although there are many bot infections there and many more uninfected hosts, meaning that traffic for a large number of hosts most be recorded to have a good chance of observing enough bots to train and test the method. While this might be solvable, it is expected to require a significant effort as the legal and privacy related issues will have to be thoroughly dealt with and it will require some luck to find e.g. an Internet Service Provider (ISP) interested in cooperation.

A variation over real bot traffic recorded in the wild is to deploy honeypots, which is hosts with an Internet connection and possibly some known vulnerabilities, that are left for bots to compromise, so their traffic can be recorded. While a honeypot can be a physical machine it is still constructed by researchers and not real in the sense of being deployed and used for real purposes, by real users in a real environment. The traffic traces of honeypots are therefore less realistic, but likely to contain a much larger fraction of bot traffic, as no users generate any benign traffic. This eases the task of identifying the bot traffic and labelling it. Using honeypots eliminates the issue of not acting on knowledge of possible ongoing perpetrations and the possible resulting damages. It does however mean that a platform is intentionally left open and available for a botmaster to use for his malicious purposes, which might have legal implications. It is possible these implications can be mitigated through smart network filtering or other technical means.

Real bot traffic is the best data, as it is perfectly realistic, but the issues mentioned above are overwhelming. Real bot traffic is deemed unsuitable for this work.

3.2 Synthetic bot traffic

An alternative to observing bots in the wild, with all the hurdles mentioned above, is to run bots in fully controlled environments, as seen in much of the related work. This approach has the following advantages and disadvantages:

- Advantages:
  - No damage to third parties
  - No real users - no privacy issues
  - Capability to generate samples as needed
  - Easily labelled data
- Disadvantages:
  - Loss of realism by being generated
  - Life-cycle is forced on data
  - Bot malware must be selected and obtained
  - Need for a secure execution environment

It is obvious that with proper containment in place no damage is done, with no real user no-ones privacy is breached and when controlling the execution, samples can be generated as needed. Furthermore a large gain lies in the capability to control how the host moves through the infection life-cycle, providing the required labels with hardly any effort.

Using synthetic bot traffic still has some disadvantages, of which the most notable is that a bot will behave according to the instructions given. By using real bots, found on Internet forums and the like, it will be real bot code that executes, but with modifications and commands deemed suitable by a researcher, not the botmaster, which introduces a bias. The selected timing of commands and the commands executed will result in a different traffic trace, leading to different alerts and a different result. Bots for which source code is available free of charge as semi-public downloads are expected to be outdated compared to some of those seen in real bot and honeypot traffic, as botmasters have an interest in keeping well performing, novel bots to themselves or make money by selling them.

This entire work evolves around the assumption of one life-cycle model for a host being infected fits for any bot, so using this model when generating data for evaluation has some implications. Testing with such synthetic data impacts the results if the proposed method captures the behaviour of the life-cycle model in the synthetic training data and relies on the same in the synthetic test data. This can be overcome by evaluating with real bot traffic, as the life-cycle model has no influence on how that is produced.

Furthermore the selection of bots to use will also cause a deviation from real botnet traffic, as the selection is almost certain to be different from what truly represents real bots and their traffic right now on the Internet. How big the difference is can only be truly known by comparing the results on real and synthetic traffic, and as the first has been deemed unsuitable for this work this will not be known. It is however clear that the issue is inherent in the approach.

The last hindrance considered is that no background traffic is generated by a real user. One solution is to emulate user inputs by executing macros or programs. This opens up a host of new considerations of how to mimic true users, yet it will still be synthetic. For the most realistic background traffic, real user traffic can be merged with synthetic traffic in time and used address space, as discussed in section 3.4. As bots attempt to be stealthy it can be assumed that bot and user traffic from the same host is reasonably additive, and therefore this is the most realistic solution.

Running bots raises some practical issues that needs to be handled. The first has already been mentioned, namely how to obtain bots. It is time consuming to find bot source code, understand it, configure it, run it and validate that it actually runs and truly is a functioning bot. When running bots, either in the process of configuring and validating their basic functionality or as a part of a procedure to generate synthetic bot traffic, precautions most be taken, as it is malicious software executing. Before running a bot on a host it is necessary to ensure that the previous bot has been removed fully, to avoid cross-contamination of executions, and it must be ensured that no third party incurs damage resulting from the bot execution. Beyond this the required infrastructure for running a bot must be set up.

Some related work relies on running selected bots with a limited Internet connection, as discussed in section 1.4. This can be seen as something in between the synthetic bot traffic, where bots are selected and executed by researchers, and the honeypot, where a bot executes with Internet access. By executing real binaries and allowing them to contact the real C&C infrastructure more realistic traffic can be observed, and no code modifications or C&C infrastructure setup is needed. A benefit of this is that binaries are much easier to obtain than source code. It does of course also come with some disadvantages. One of them is the need to consider containment, such that no third parties incur damages as a result of the research effort. Another drawback, compared to the synthetic approach discussed above, is the lack of labels.

The obstacles of running bots to synthetically generate bot traffic are found to be manageable, and it is believed that the loss of realism must be tolerated, as this is the only feasible method for obtaining labelled bot traffic.

In the following essential issues from above are discussed and solved. A setup that provides containment and ensures that the hosts are cleaned between bot runs is presented in section 3.2.1, details of the used bots are found in section 3.2.2 and a procedure for generating the synthetic bot traffic is presented in section 3.2.3.

3.2.1 Setup

The setup is to be used when running bots to validate their functionality and when running them to generate synthetic traffic. It is therefore designed to handle some of the issues above.

A management PC is placed on the border between the inmate network, used for bots, and the rest of the world, this provides convenient remote control functionality and infrastructure services needed when working with or running botnets. Isolating the inmates from the rest of the world with a single PC makes it easy to harden the interfaces towards the inmate network, through up-to-date and securely configured services. A diagram for the setup is presented on Figure 3.1.

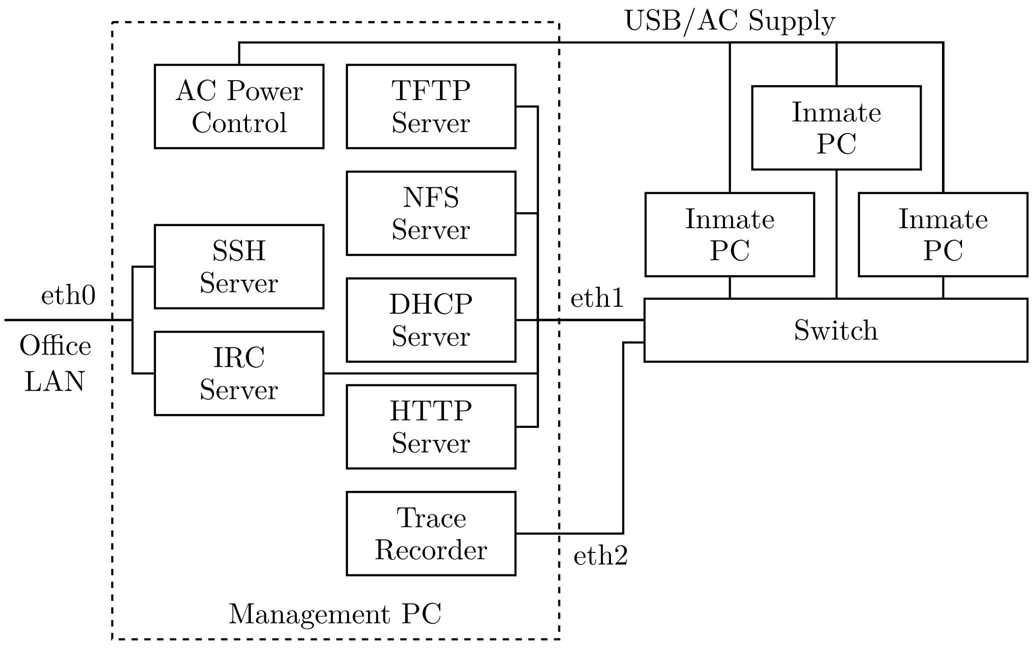


Figure 3.1: Diagram for the used setup, which is further described in appendix B.

The technical details needed for deeper understanding or reproduction are available in appendix B, but put briefly the setup consists of the following elements:

- Inmate PCs - the host on which bots are executed
- A switch - providing network connectivity for inmates and a spanning port mirroring all traffic.
- A management PC with the following services:
  - DHCP Server - for configuring inmate IP addresses and to enable automated disk imaging
  - TFTP and NFS Server - for automated disk imaging
  - SAMBA Server - for file transfer and storage
  - Trace Recorder - for recording network traffic traces
  - SSH Server - for convenient and secure remote access
  - AC Power Control - for hard resets by switching power supply
  - IRC server - for C&C channel
  - HTTP server - for binary downloads etc.

3.2.2 Bots used

Bot binaries are necessary in order to generate the data, and furthermore these need to be configured to use the infrastructure of the setup, such that e.g. the C&C channel can be established.

While obtaining bot binaries is easy, modifying and configuring them as needed is expected to be complicated, as they might employ obfuscation techniques, encryption etc. to avoid reverse engineering and attempts at modifying them to make them run in a research setting. This is to be expected for bot binaries because the bot malware authors have an interest in not sharing any information about a bots inner working with others, especially researchers. For this work the alternative approach of obtaining bot source code is selected, which provides freedom to configure the bot before building the binary.

Bot source code can be obtained by writing it, as done by Lu and Brooks [2012], which will not represent real bots very well if the researcher has incorrect ideas about how bots are implemented. Another approach is to buy bots source code from malware programmers, but this introduces ethical issues. For this work bot source code is obtained by searching the Internet for freely available bot source code, which has the disadvantage of being older than the state-of-the-art for bots.

To avoid the need for setting up multiple different C&C infrastructures only bots using a centralised IRC channel is considered from this point. As discussed in appendix C a set of bots with source code has been collected and the following is to be used:

- sdbot05b
- spybot1.3

3.2.3 Procedure

To generate a traffic trace an inmate is put through the phases of the life-cycle described in section 1.4.2 and presented on Figure 1.8, while traffic and phase is recorded. This section documents the design of the procedure to generate this traffic.

As seen from Figure 1.8 the life-cycle branches out and merges again at two points, corresponding to a local or remote exploit and a bot that autonomously performs attacks or not. These two branches, each with two options, results in 4 different paths from Clean to Active. The four paths are presented in tables 3.1.

Phase	Event
	Remedy
Clean	
	Scan Hit
Discovered	
	Remote Exploit
Infected	
	C&C connect
C&C Connected	
	Attack
Active	
	Remedy

(a) Remote exploit without Auto-active

Phase	Event
	Remedy
Clean	
	Scan Hit
Discovered	
	Remote Exploit
Infected	
	Attack
Auto-active	
	C&C connect
C&C Connected	
	Attack
Active	
	Remedy

(b) Remote exploit with Auto-active

Phase	Event
	Remedy
Clean	
	Local Exploit
Infected	
	C&C connect
C&C Connected	
	Attack
Active	
	Remedy

(c) Local exploit without Auto-active

Phase	Event
	Remedy
Clean	
	Local Exploit
Infected	
	Attack
Auto-active	
	C&C connect
C&C Connected	
	Attack
Active	
	Remedy

(d) Local exploit with Auto-active

Table 3.1: Tables of the four different paths through the life-cycle model.

All of the life-cycle events of tables 3.1 has a one-to-one mapping with a set of actions/observation that can be performed by the researcher, who for the procedure also has the role of bot master. The mapping is as follows:

- Remedy: Copying a clean image to the inmate.
- Scan Hit: Point a network or vulnerability scanner to the inmate which produces a hit.
- Remote Exploit: Use the exploit over the network.
- Local Exploit: Download and execute code or binary on the host and execute the exploit.

- C&C Connect: Observe a connection to the C&C channel controlled by the researcher.
- Attack: Initiate or observe malicious activity.

For simplicity the remainder of this work is delimited to Local Exploits and bots with no Auto-active phase, meaning data generated according to the path of table 3.1c is used. While this will have negative impact on the generality of the results, it is believed that the method and procedure is independent of this delimitation and can be applied to botnet infections utilising Remote Exploits and exercising an Auto-active phase as well.

With the bijective mapping of life-cycle events to actions/observations and a single life-cycle path the general procedure follows:

1. Copy image with a bot binary, no other infections, malicious services or executing bot code to the inmate.
2. Start recording traffic trace.
3. Boot inmate, record beginning of Clean phase.
4. Wait until Clean phase has lasted 2 minutes.
5. Execute bot binary, record beginning of Infected phase.
6. Wait until bot reports in on C&C channel, record beginning of C&C Connected phase.
7. Wait until Connected phase has lasted 2 minutes.
8. Order bot to attack, record beginning of Active phase.
9. Wait until Active phase has lasted 2 minutes.
10. Cut power to inmate, record end of Active phase.
11. Stop recording traffic trace.

This procedure is applied independently to each of the bots of section 3.2.2 to generate synthetic traffic traces for each. The mapping between procedure steps and the life-cycle events enables timestamped logging of the events. Using the timestamps for the life-cycle, it is easy to determine labels for the data. For most of the phases a duration must be selected in this procedure. A reasonable duration can be found through analysis of real bot traffic, which is not available thus this is not an option. For real bot infection life-cycles the durations are expected to be random. However, without access to real bot traces the distribution and parameters for a random value will have to be selected arbitrarily. Based on this an arbitrarily selected constant duration is acceptable. A phase duration of 2 minutes is selected, expecting it is long enough for a suitable number of alerts in each phase, while the duration of the entire procedure is still manageable.

3.3 Benign traffic

With the synthetic bot traffic filling the need for malicious samples, there is still a need for benign samples to form a real detection problem. The following advantages and disadvantages are acknowledged and discussed below:

- Advantages:
  - Ample amounts of data
  - Easy labelling
  - Realistic
- Disadvantages:
  - Need to define what is representative
  - Access to representative data
  - Privacy concerns
  - Need to assume or ensure that host are clean

As already discussed, e.g. in section 2.2 on the base-rate fallacy, benign traffic is assumed to be abundant. The labelling problem discussed earlier in this chapter also applies to benign traffic, but if it can be assured or assumed that all traffic is originating from clean hosts it follows that all traffic and the resulting alerts can be labelled as clean, making it much easier than the case of real bot traffic. By recording real benign traffic, with all hosts truly clean and representative of the real world, perfectly realistic data can be obtained.

While benign traffic is abundant, obtaining representative samples is not trivial, as it is hard to define exactly what is representative. Usage patterns for computer users are expected to vary with age, nationality, time of day and job position among others. As for reasons to this, the following purely speculative suggestions are made:

- Children play games while adults work,
- legislative restrictions limits people in some countries,
- people use the Internet for work during the day and recreational activities in the evening and
- a sales representative compared to a software developer will use completely different set of applications.

Such different usage patterns will lead to differences in the generated traffic. An obvious unfeasible and unrealistic solution is to “sample the world”. As this not is possible, a subset of all Internet users must be selected. For practical reasons, this could be the customers of an ISP, the users on a university or corporate campus or the residents in a residential building sharing their Internet connection.

To obtain the data, access to traffic is required, however the issue is much smaller than for real bot traffic, as all users produce ample amounts of benign traffic. The privacy and confidentiality issues when recording the traffic of individuals remains, but are also limited as much less users are necessary, and the users can be contacted to obtain permission to work on their data. Applying labels by assuming all traffic to be clean will add noise to the data if the assumption is not true. One approach to mitigate this is to inspect all host, which becomes unfeasible as the number of hosts grow. Another option is to apply other detection methods, possibly followed by manual inspection, to check if bots are present. Similar to the issue of making a selection of representative bots is the issue of making a selection of representative hosts. The easiest and least realistic procedure is to use ones own traffic, while one of the more difficult and possibly the most realistic is to tap an ISP network. With a large number of host the task of obtaining user consent and the task of validating that hosts are truly benign increases significantly.

Overall it is deemed feasible to record a moderate set of traces from a limited number of users, tolerating that the selected users might not be completely representative of Internet users in general.

3.3.1 Setup

To record benign traffic a setup as seen on Figure 3.2 is used. The traffic is recorded by putting a tap on the connection between the office of four students at the department of Electronic Systems, Aalborg University and the remainder of the campus network and the Internet. As the bots selected in section 3.2.2 are using IRC, it is ensured that the benign traffic also includes benign IRC traffic. Without this detection will be trivial, as the easily recognisable IRC traffic is certain to be C&C traffic. Traces are recorded during workings days from 7:00 to 17:00. Further details are available in appendix E.

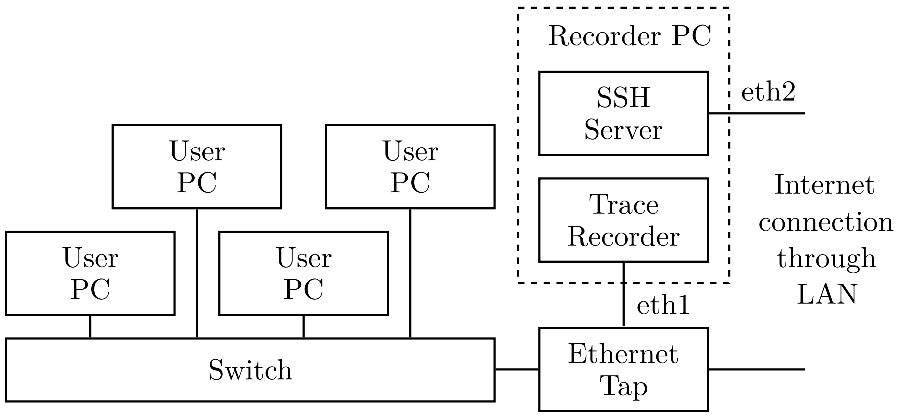


Figure 3.2: Diagram for the used setup, which is further described in appendix E.

3.4 Mixed traffic

Evaluating the method on benign and synthetic bot traffic has a serious shortcoming as the life-cycle phase estimation is done on a per host basis. If the host addresses in the two types of data does not overlap, all life-cycle estimates will be on either background noise, i.e. false alerts, during the clean phase or on synthetic bot traffic without any realistic background noise. If the two types of data are recorded at different times, but somehow overlapping in host addresses, the alert sequences will contain two clearly separated sub-sequences, one originating from the synthetic bot traffic and one from the benign. This is not acceptable, as user and bot activity might occur on the same hosts and at the same, which is expected to be harder than the above settings. To cope with this, a third type of data is generated by mixing synthetic bot traffic and benign traffic to match in both host address space and time. This approach has the following advantages and disadvantages:

- Advantages:
  - Provides a harder detection problem
- Disadvantages:
  - Reuse of data

The advantage is that the conservative approach of solving a more difficult problem serves as a better support for the usefulness of the proposed method. The drawback is that benign and synthetic bot traffic is reused, which can falsely make the results appear to be based on a more diverse selection of bots and users, while this is not the case. This issue can be solved by obtaining new bots and new benign traffic to use only for mixed traffic.

3.4.1 Procedure

To generate a mixed traffic trace the following procedure is applied:

- Pick one benign and one synthetic trace
- Replace all IP Media Access Control (MAC) addresses in the benign trace with random, unique ones.
- Choose 10 random hosts, identified by IP address, in the benign trace.
- For each of the 10 hosts, create a duplicate of the one synthetic trace.
- Rewrite the MAC and IP address of the bot to that of the host in each of the 10 synthetic traces
- Rewrite the timestamps of the synthetic traces, by shifting them to a starting time uniformly distributed in the time span of their matching host in the benign trace.
- Merge the ten rewritten duplicates of the synthetic trace into the rewritten benign trace.
- Store the result as a mixed trace

For details on the obtained data the reader is referred to appendix F.

4 Results

In this chapter the evaluation results for the proposed method are presented. Initially the used metrics are defined, before presenting the obtained results. The results are considered at two different levels of detail. The first is concerned with the detection of bot infections on hosts while completely ignoring individual phase estimation errors. This serves to indicate how suitable the method is for deployment in production, as it all comes down to detecting bots. For the second part life-cycle phase estimation performance is considered. By analysing the life-cycle estimation errors, a deeper understanding of the performance is obtained, providing valuable information for future improvements. Finally the results for evaluating on the training data is presented, as this gives an impression of the robustness.

For evaluation the counts of all positive and all negative detection outcomes are introduced. The counts are the number of hosts with respectively positive or negative detection outcome. The outcome is determined by applying the detection procedure defined in section 2.3.3 to the state estimate sequences (Q').

$$P: \text{ Positives, count of hosts detected as infected} \tag{4.1a}$$
$$N: \text{ Negatives, count of host not detected as infected} \tag{4.1b}$$

The two are both divided according to whether the detection outcome is true or false, i.e. in agreement with the outcome of applying detection to the known true state (Q).

$$TP: \text{ True Positives, count of positives that are truly infected} \tag{4.2a}$$
$$FP: \text{ False Positives, count of positives that are truly not infected} \tag{4.2b}$$
$$TN: \text{ True Negatives, count of negative that are truly not infected} \tag{4.2c}$$
$$FN: \text{ False Negatives, count of negative that are truly infected} \tag{4.2d}$$

These numbers expose information about the detection performance as FN and FP are the detection errors, referred to as errors of type I and type II respectively. Based on these counts, the following rates are introduced, as the performance metric for detection:

$$TPR = \frac{TP}{P} \quad : \text{True positive rate} \tag{4.3a}$$
$$FPR = \frac{FP}{N} \quad : \text{False positive rate} \tag{4.3b}$$

They indicate the probability of detecting a bot infection (TPR, equation (4.3a)), and the probability of raising a false alert on a clean host (FPR, equation (4.3b)). The accuracy of the method describes how often the detection outcome is correct:

$$ACC = \frac{TP + TN}{P + N} \quad \text{: Accuracy, the rate of correct detections outcomes} \tag{4.4}$$

The life-cycle phase estimation has the characteristics of a classification problem and the above metrics do not apply, unless the data is truncated to e.g. clean phase against all the other phases. Instead counts on the correct and incorrect states can be considered, along with confusion matrices, also known as misclassification tables. The confusion matrices provides detailed information about what leads to detection errors.

4.1 Bot detection

The test data contains 760 hosts identified by unique IP addresses, of which 11 are infected with bot malware. To be conservative and guarantee a fair treatment of other work, only hosts for which alerts are raised by the IDS are included in the results. In accordance with the procedures of chapter 3, roughly half of the non-infected hosts are duplicates of the other half, and 10 of the infected hosts is the result of merging traffic from a non infected host with a duplicate of the one synthetically infected host.

All the infected hosts are detected resulting in a true positive rate of 100.000%, which is the best possible result. Other work, such as Shin et al. [2013] and Gu et al. [2007] achieve as good results in similar settings, namely synthetic bot traffic[1].

8 hosts that are not infected with malware are falsely detected as infected, resulting in a false positive rate of 1.068%. This is a god result in a range comparable to Shin et al. [2013] ($FPR = 0.68\%$) and Masud et al. [2008] ($FPR = 1.3\%$)[2]. As pointed out in section 2.2.1 this is an important metric, and the good result bodes well for the usefulness of the proposed method.

The achieved accuracy is also good ($ACC = 98.947\%$), as expected with good TPR and FPR results. This result is also comparable to that of Masud et al. [2008] ($ACC = 99.9\%$)[3].

Based on this it is concluded that a HMM can be applied to estimate the life-cycle phases of bot malware infections and thereby enable detection of the infection, which answers the second compositional question of the research question found in section 1.6. The proposed method is found to be approximately on par with the best performing methods known from the literature. With the different sets of available metrics, varying evaluation conditions and different types of samples (Host, network flows or processes) it is not possible to conclude with an absolute ordering of the methods. To do this all the methods should be applied to the same data, the evaluation should be done on samples of a "common denominator" such as per host and the same metrics should be calculated.

[1]Masud et al. [2008] do not present this metric.
[2]Gu et al. [2007] do not present this metric.
[3]Gu et al. [2007] and Shin et al. [2013] do not present this metric.

It is noted that Shin et al. [2013] relies on both host and network data, which is assumed to be an advantage over this work, in that only network data is used here. Masud et al. [2008] uses one bot in both training and testing, and applies a collection of different machine learning algorithms with slightly different outcomes. To avoid favouring the current work, the highest individual metric is used in the comparison above. Gu et al. [2007] performs evaluation in various settings, but only the one with synthetic generated data is used, in order to be fair.

The counts of TP, FP, TN and FN is provided in table 4.1 for reference.

	Infected	Not Infected
Detected	11 (TP)	8 (FP)
Not Detected	0 (FN)	741(TN)

Table 4.1: Detection results with errors in number of hosts

4.2 Life-cycle estimation

The evaluation results for the life-cycle estimation is presented in this section. Among the known related work discussing life-cycles, none provide an explicit estimates of phases, hence these results are not compared to any related work. Instead these results provide an understanding of the errors made in the estimation, which propagates to the bot infection detection.

Overall 98.728% of the states are estimated correctly, which answers the last compositional question to the research question of section 1.6 - a HMM is suitable to model the life-cycle of bot malware infections, at least those considered in this evaluation. For completeness the estimation must also include real bot traffic, but that is left for future work. Table 4.2 provides details of how the true phases, the estimates and the errors are distributed. Each column represents a phase which a host is known to have been in when an alert concerning the host was received from the IDS. Each row indicates the phase estimated by the proposed method when receiving an alert. The entries are the counts of estimates made, where the given host is known to be in the phase at the top of the column, while it is estimated to be in the phase denoted by the row. This provides details of how the method confuses different phases, hence the name confusion matrix. Ideally all entries, except the diagonal, is zero as that signifies that the method has not confused any phases. It is noted that the majority of the estimates are in the diagonal.

		Truth			
		Clean	Infected	C&C Conn.	Active
Estimate	Clean	34205	10	0	3
	Infected	10	26	0	0
	C&C Connected	406	0	22	12
	Active	0	0	0	1
	Sum	34621	36	22	16

Table 4.2: Misclassification of life-cycle phases. Count by phases.

As there is a heavy class imbalance for the Clean phase, table 4.3 is provided with normalised values. This provides a better picture of how likely a certain error is.

		Truth			
		Clean	Infected	C&C Conn.	Active
Estimate	Clean	98.798%	27.778%	0.000%	18.750%
	Infected	0.029%	72.222%	0.000%	0.000%
	C&C Connected	1.173%	0.000%	100.000%	75.000%
	Active	0.000%	0.000%	0.000%	6.250%
	Sum	100.000%	100.000%	100.000%	100.000%

Table 4.3: Misclassification of life-cycle phases. Normalised vertically.

The results for bot infection detection in the previous section only left room for improvement on the FPR metric. Considering the rule applied for detection, it is clear that false positives only can occur when a host at some point is estimated to be in a phase different from Clean, when it truly is Clean. These kinds of errors are seen in the column labelled Clean, in both tables 4.2 and 4.3. It is noted that the rate of error for the clean state is higher than the FPR. This can occur because multiple estimation errors for a host can only cause one error in the per host detection outcome. C&C Connected is the phase that is most often confused with the Clean, leading to errors. This might in part be due to one of the hosts contributing to the benign trace also being connected to the IRC server used for C&C. If this is the case, some false positives might be eliminated by ensuring proper separation of the setups for producing synthetic and benign data. Another possibility is that the methods trains the model to rely on IRC communication, as the training data only contains an IRC bot. In that case the error is more tightly coupled with the method, and hence a bigger concern. One approach to overcome this is to review and improve the relevant rules for generating alerts in the IDS.

The Infected phase is often confused for being Clean. In section 3.2.3 it is decided to only use local exploits, which means that the only event expected to be seen on the network during the Infected phase is a download of the bot binary via HTTP. As a results of practical considerations the Clean phase also includes a download of a file by HTTP. This might be a source of noise that can be eliminated by improving the implementation of the data generating procedures.

The smallest diagonal entry, by far, is that of the active phase. This is surprising if it is assumed that some very noisy activity is performed, because any IDS is expected to recognise e.g. a port scan. However, by inspecting the alerts on the synthetic data it is found that Snort produces two different types of alerts for test and training, although both perform port scanning. With no recognisable alerts in the test data for the Active phases the model can only be expected to perform very badly. As the C&C connection remains established it is also to be expected that Active easily is confused with the C&C Connected phase. This result highlights both a weakness and a strength in the proposed method. The weakness is that it gains nothing from the kind alerts which did not appear in the training data. For the best results, the training and test data should be more alike, but this is always the case. A solution is to use more bot traffic with different bots and behaviours in the Active phase. The strength lies in the ability to function remarkably well despite the significant difference in test and training data.

4.3 Robustness

Beyond the metrics presented and discussed above, it is also important to consider how robust the method is, i.e. how well it performs when presented to new data. Ideally evaluation relies on cross-validation on multiple independent data sets to estimate statistics of the metrics, such as mean and variance. Due to the amount of work involved with obtaining data sets, cross-validation is left for future work. Still, the question on how robust the method is still needs to be discussed. By applying the detection procedure and calculating the performance metrics on the data used for training, the robustness between test and training data can be evaluated. Table 4.4 presents the results of such an evaluation.

	Training	Test	Change
TPR:	100.000%	100.000%	0.000%*point*
FPR:	0.000%	1.070%	1.070%*point*
ACC:	100.000%	98.946%	−1.054%*point*
State est. error:	0.057%	1.272%	1.215%*point*

Table 4.4: Comparison of selected metrics between evaluation on test and training data

It is evident from table 4.4 that the model fits better to the training data than to the test data. Relying on a single bot and in practice one instance of its execution comes with a risk of over-fitting, and as seen the metrics change remarkably from training to test data, indicating over-fitting. Keeping the model size fixed and using larger, more diverse data for training will make over-fitting less likely.

Another concern stems from the choice of only using IRC bots. The results provide no indication of how the method handles bot malware implemented with other protocols or topologies, as such are not present in the data. In particular C&C traffic through HTTP is expected to be hard to discern from the abundance of HTTP caused by ordinary, benign usage. Encrypted C&C traffic as well as novel exploits and malicious activities poses similar problems. These problems can be mitigated by adding sensors to generate alerts by remote host reputation, network information trade rate and payload statistics[Shin et al., 2013][Gu et al., 2007].

By relying only on synthetic traffic, in particularly for the testing data, the generality of the results is limited. Assuming that the infection life-cycle is influenced by how a bot master uses his botnet, the performance on real bot traffic can differ significantly from the found results. To obtain more general results the evaluation must be performed on real bot traffic. Asymmetric test and training data sets, with real bot traffic only in the testing data, and evaluation on detection performance alone, eliminates the need for life-cycle phase labelling. If the method performs well under such conditions, it can be concluded that the synthetic bot traffic is a good substitute for the real bot traffic and the designed procedure captures the behaviour of a bot master reasonably well.

5 Future work

In this chapter multiple interesting issues found in the project are discussed. Ideas for future work to remedy the problems or expand the work are presented. The chapter is organised by grouping similar and related issues in sections, with no particular ordering of the sections.

5.1 Finalise the proof-of-concept implementation

While the current implementation is sufficient for an initial investigation on the possibilities of detection with the proposed method, it is delimited from what is described up to this point. Other ideas are introduced but not actually implemented. This section covers these aspects, which are good candidates for what to do next to bring the implementation further.

It has been proposed to make use of a probabilistic ranking of hosts to determine those most likely to be infected with bot malware. This has obvious benefits in real applications, as efforts can be targeted to the hosts detected with highest certainty. With a detection threshold relying on the same probabilities, it is possible that detection performance can be improved further. Recall that no infected hosts are missed ($TPR = 100.000\%$) and all the errors made in detection are false positives ($FP = 8$, $FPR = 1.068\%$). If the false positives and true positives are separable by the probability, the errors can be corrected.

In the current implementation and evaluation the detection outcome is per host. The host is identified by an IP address and considered to be paired with a MAC address. It is implicitly assumed that the IP addresses are unique identifiers and that they map to one MAC address. Considering the duration of the traffic traces used this might hold, but a better solution is desired. MAC addresses are designed to be unique and therefore a better identifier for hosts. Some issues do however remain, as MAC addresses can be spoofed and in some situations a host might have multiple Network Interface Cards (NICs).

As discussed in section 2.3.3, the current implementation provides one detection outcome per host. Considering how minor a change is required, it is reasonable to modify the proof-of-concept to provide a detection outcome for a host on every alert concerning that host.

Section 2.3.3 describes how the most likely individual state is used for state estimation, as the underlying algorithm do not rely on future data and is computationally cheap. The current implementation of life-cycle state estimation does actually rely on the Viterbi path found with the Viterbi algorithm. The implementation should be updated to use the individual state estimation. A comparison of the two, and possibly other estimation methods, can potentially yield improvements of the method.

5.2 Minor improvements and tweaks

Some variations in the implementation have been considered but not implemented or thoroughly investigated. The variations have the potential of improving the results, but any small impact can possibly be hidden by noise. As a consequence, cross-validation can be necessary to evaluate the variations.

One variation is to discern between alerts with a host as source and as destination. The rationale behind this is that the direction of traffic causing an alert is relevant. As an example a host is expected to be in the Active phase if it is the source of a port scan, but any host can be targeted. While potentially doubling the alphabet size in the trained HMM, this a expected to have negligible impact, as only one model is trained and stored.

The initial state probability (π) is chosen such the model is certain that any host initially is assumed to be Clean. This suppresses the first alert of any host, being false or true, with the advantage that no false detections occur from a single alert and the disadvantage that the detection reacts a bit slower. If response time is critical, this can be changed, keeping in mind that suppressing the first alert has little negative impact if the subsequent alerts follows quickly when infections occur. An alternative is to make an empirical estimation of π as done with A and B, to manually pick some values or some combination of the former two.

For the empirical estimation of state transition probabilities (A) and symbol emission probabilities (B) a very small number (10^{-20}) is added to all counts of occurrences, such that none are zero. This makes the model capable of handling rare transition, not seen in the training data. The value is selected arbitrarily, so it is sensible to investigate if a better value can be used.

Another change concerned with the incompleteness of training data, evolves around how alert types only seen in test data is handled. Currently all such alerts are truncated to a single "any other" symbol, for which no information is used. Alternatively such alerts might be ignored completely, as the low emission probability will cause a big drop in the probabilities used for ranking as discussed in section 5.1 above. Alternatively groups of similar alerts can be considered, in order to cover the holes of missing alert types in the training data.

The implementation and the evaluation considers all hosts found in the data. Some of the traffic is across the campus network infrastructure and the Internet to hosts for which only a little of their network activity is visible. In a deployment scenario the ambition is likely to protect the hosts in the Local-Area Network (LAN), and the method must be modified to reflect this. It is assumed that hosts for which all network traffic is available are much easier to model and therefore a better result is expected.

5.3 Improved evaluation

While the evaluation results in chapter 4 serves to show that the proposed method can be used to detect bot malware infections under the given circumstances, more can be done to establish the robustness and generality of the method. Being able to provide a exact ranking of the proposed and other methods is also desirable.

As already mentioned cross-validation is useful for establishing how robust the method is. It has also been mentioned that more realistic and diverse data, along with more data is expected to give more useful and stable results. Both these areas of improvement are essentially dependent on obtaining more data. By repeating work already done and documented in chapter 3 more data, similar to the already used, can be obtained. Solving the issues of obtaining real bot traffic, as discussed in chapter 3, will provide more realistic and diverse data. It is possible to work around some of the issues by using real traffic for testing of false positives only. This will require some manual inspection to validate positive detections outcomes. The number of detections is expected to be low, assuming a low ratio of infected to non-infected hosts and considering the good FPR of section 4.1.

The length of the state estimation sequences varies from host to host, as varying numbers of alerts are raised. For sufficiently many, sufficiently long sequences the HMM implies that any transition with non-zero probability is made eventually. With many, long sequences of hosts in the Clean phase the effect can lead to the method falsely estimating another state than Clean, resulting in false positives in the detection. To determine if this effect influences the results, the hosts can be put in bins according to the length of state estimation sequences. If the performance for the bins with longer sequences are worse than those with the shorter, this can be due to the above. Is this an issue it can easily be solved by only considering the last X alerts in the state estimation, where X is some number.

Assuming that the detection threshold, discussed in section 5.1 above, is implemented, Receiver Operating Characteristic (ROC) curves can be included in the evaluation. While these curves do not directly allow a numeric comparison of different methods, they are commonly employed to present the performance of detectors with respect to a trade off between false positives and true positives[Lu and Brooks, 2011][Cho and Park, 2003][Ourston et al., 2004]. The area under a ROC curve can be estimated and used for a numeric metric.

The traffic used to represent benign traffic covers the working day activities of four students at the department of Electronic Systems, Aalborg University, as specified in section 3.3.1. As discussed in section 3.3 this is expected to have a negative impact of how representative the traffic is for all benign traffic. To improve on the issue benign traffic recorded under different circumstances can be included in the evaluation. By comparing result for different kinds of benign traffic the impact and relevance of this issue can be investigated.

As evident in chapter 6, a fair comparison with the results of other work is extremely difficult. To overcome this it is an option to obtain e.g. BotHunter presented by Gu et al. [2007], and apply this method to the data used in this work. Due to the specific labelling requirements of this work and confidentiality obtaining the data used in the related work is more laborious. Another way to obtain more reliable results is to deploy the proposed method in real life to investigate the FPR.

In section 3.2.3 it is selected to disregard that a bot might be Active before establishing a C&C channel and that infections might occur as Remote exploits, rather than just Local exploits. While it is assumed that this has no impact on the functioning of the HMM model - it is simply an additional state - this must be investigated and evaluate to discover if there is an impact on the performance.

5.4 Implementation for production

Before applying the method in a real, production environment some changes are necessary and some additional features are relevant to consider.

The present implementation executes off-line, such that all data is recorded and each step of the proposed method is applied on its own. In a real deployment it will be necessary to run on-line. To do this, the different steps must be performed in real-time with appropriate throughput. According to Sourcefire [2013] a commercial IDS solution, processing traffic at 40*Gbps*, is available. Based on this it is assumed that Snort can handle large amounts of network traffic in real-time. The life-cycle state estimation through the HMM is lightweight and needs to process much less data than the IDS. Furthermore the estimation and detection can be split at host granularity and distributed to other hardware, meaning that no bottlenecks are expected, compared to an IDS alone. This implies that the method readily can be run on-line.

The sketches of a deployment scenario on Figure 2.4 in section 2.3 presents a setup with a tap and the IDS operating passively on the mirrored traffic. It is also an option to put the IDS in the place of the tap to actively react on the traffic. An active solution has the advantage that infection prevention is possible, while a passive solution requires reaction and interaction through other channels, presumably with a delay so high that it is a matter of infection detection, without ambitions of completely avoiding damage. An active solution has much stricter requirements for false positives, as the impact can be that a user is blocked from performing his or her tasks, rather than just an extra detection to analyse. Furthermore, an active solution introduces a delay across the IDS, as packets must be hold until a decision for an action is made. The access to the Internet will also depend on the IDS operating correctly.

5.5 Multiple and improved sources of alerts

While the current implementation relies only on Snort with a basic configuration this is not necessarily the best solution. The input to the proposed method is symbols, based on alerts, the IP address of the host to which the symbol pertains and a time stamp to allow for ordering. For training and testing the true life-cycle phase of the host at the time of every symbol must naturally also be known. The symbols might originate from multiple different modules, as seen in the work of Shin et al. [2013], or from a modified version of Snort with additional plug-ins targeting botnets, as done by Gu et al. [2007]. In fact the modules and plug-ins of the two are expected to provide inputs with additional information on the infection life-cycle. This leads to the expectation that they can provide improvements if added to the current work. Due to the simple, similar interfaces the integration of the modules and plug-ins are expected to be reasonably straightforward.

6 Conclusion

With this chapter conclusions are made on the work documented in this report. First an outline of the report is made, pointing out important details. This leads to a conclusive answer for the research question. Finally thoughts on the process are presented.

This report documents an understanding of malware, botnets and the botnet problem. In particular the financial impacts of botnets, described in the introduction, are highlighted. Through spam campaigns, botnets are estimated to cost at least 50 billion USD a year and a single botnet is known to have facilitated theft of 47 million USD. Through related work a model for a host moving through the life-cycle of a bot malware infection is established. A detection method, relying on a Hidden Markov Model (HMM) and the life-cycle model in combination, is proposed. To structure the remainder of the work a research question regarding the evaluation of the method is stated:

Can the framework of HMMs be used to model the bot infection life-cycle of a host, based on IDS alerts, and thereby provide improved detection of bot infections?

Relevant knowledge on HMMs is described, where an important detail is that the needed calculations are computationally cheap for small models, such as the life-cycle model used here. IDSs are discussed to provide an understanding of these existing solutions, which is used as a part of the method proposed in this work. Applying the above knowledge of HMMs and IDSs, the method is defined with more details: A single HMM is trained and then applied individually to each host on which infections must be detected. The unobservable states of the HMM is equivalent to the life-cycles phases of the host and the alerts of an IDS forms the observable symbols emitted from the HMM of the host.

A substantial amount of time has been invested in obtaining data for evaluation, in particular because the proposed method has the unique requirement that the alerts of the IDS must be labelled with the true life-cycle phase at the time the alert is raised. Mainly as a result of this unique requirement, the data used in this work is synthetic bot traffic, i.e. generated by running a bot in a completely controlled network. Real benign traffic recorded from everyday usage is also used. For this benign traffic it is important to keep in mind that it only covers so many users, and the results is delimited from the big discussion on what truly represents benign traffic. To challenge the proposed method, synthetic bot traffic and benign traffic is mixed. The mixing is done such that bot traffic and benign traffic appears to be from the same host and occurring at the same time.

The obtained data is used to evaluate how suitable the proposed method is for detecting bot malware infections. A false positive rate of 1.068%, a true positive rate of 100.000% and an accuracy of 98.947% is achieved. These are good results, comparable to the state-of-the-art as found in the work of Gu et al. [2007], Shin et al. [2013] and Masud et al.

[2008]. It is not sensible to make an exact comparison of any metrics, as the conditions vary among the four evaluations. The performance of the life-cycle estimation is also evaluated in detail and found to be good in general and for all states except one. A high error rate is found for estimations of the Active phase in the life-cycle. After further analysis the errors can be explained by confirmed differences among test and training data. The robustness of the proposed method is considered and found to be reasonable, with room for improvement. Larger amounts and more diverse data on especially bot traffic is found to be needed for a more stable detection performance. cross-validation of the results must be added in future work to provide solid metrics of the stability of the method. This does however required much more bot data, which is cumbersome to obtain. The evaluation has its limits in that only synthetic traffic and two selected IRC bots are used. It is left for future work to evaluate the performance on other bots or bots executing under any other conditions than those described in chapter 3.

The conclusion is that the framework of HMMs can be used together with the alerts of an IDS to model the life-cycle of bot malware infections, for the bot malware used in this work and with the life-cycle of the synthetic data. By aggregating alerts per host and eliminating many false alerts, the detection is improved over that of the IDS. The detection performance is deemed to be good, as it is on par with state-of-the-art bot infection detection methods.

Taking a step back to consider the process of the project, it is has been a great and educating experience. To summarise the process: The idea of the proposed method rose from a knowledge of HMMs and IDSs, a solid comprehension of networks, some comprehension of botnets and thorough studies of related work, where the concept of an infection life-cycle appeared. With the abstract idea as a starting point the research question was defined to guide the further effort. To determine if the method was at all possible a good understanding of HMMs was obtained. For IDSs more knowledge was also needed. Despite considering it a black box or third party component in this report, setup and configuration of the used IDS required both knowledge and time. The difference between the abstract proposal in section 1.5 and the design in section 2.3 demonstrates how the level of understanding was increased. The writing on HMM and IDS in between, documents what has been learned, and provides the reader with the facts and references used.

The part of the process concerned with obtaining data involves many considerations, as discussed in chapter 3. It also involves the most tedious and trivial work in the project, and a lot of it, as different setups had to be made. The advantage is that practical experience with the IDS Snort was gained, along with more routine on configuring common network services and the used solution to automate deployment of host images across a LAN. Reaching the critical part of the process where results are obtained, it was astonishing, and somewhat a surprise, to see a performance comparable to the state-of-the-art. Of course the evaluation has its limitations, as discussed, but the results are good and solid enough to justify a further effort to evaluate and develop the method. The main threat to the method is the need for training data, labelled with the life-cycle phase of the relevant host, which is even harder to obtain than ordinary labelled bot traffic. Overall the outcome of the project is as good as one could hope, as much has been learned and what was learned turned out to be useful in practice.

Bibliography

Axelsson, S. (2000). The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security (TISSEC)*, 3(3):186–205.

Bailey, M., Cooke, E., Jahanian, F., Xu, Y., and Karir, M. (2009). A survey of botnet technology and defenses. In *Cybersecurity Applications & Technology Conference For Homeland Security - CATCH.*

Bauer, J. M., van Eeten, M. J. G., Chattopadhyay, T., and Wu, Y. (2008). Itu study on the financial aspects of network security: Malware and spam. Technical report, International Telecommunication Union.

Cho, S.-B. and Park, H.-J. (2003). Efficient anomaly detection by modeling privilege flows using hidden markov model. *Computers & Security*, 22(1):45–55.

Elisan, C. (2012). *Malware, Rootkits & Botnets A Beginner's Guide.* Beginner's Guide. McGraw-Hill Education.

Elkan, C. (2000). Results of the kdd'99 classifier learning. *ACM SIGKDD Explorations Newsletter*, 1(2):63–64.

Gu, G. (2008). *Correlation-based botnet detection in enterprise networks.* ProQuest.

Gu, G., Porras, P., Yegneswaran, V., Fong, M., and Lee, W. (2007). Bothunter: Detecting malware infection through ids-driven dialog correlation. In *Proceedings of 16th USENIX Security Symposium*, page 12. USENIX Association.

Gutmann, P. (2007). The commercial malware industry. In *DEFCON conference.*

Hachem, N., Mustapha, Y. B., Granadillo, G. G., and Debar, H. (2011). Botnets: life-cycle and taxonomy. In *Network and Information Systems Security (SAR-SSI), 2011 Conference on*, pages 1–8. IEEE.

Hoang, X. D., Hu, J., and Bertok, P. (2003). A multi-layer model for anomaly intrusion detection using program sequences of system calls. In *Proc. 11th IEEE Int'l. Conf.* Citeseer.

Holz, T., Steiner, M., Dahl, F., Biersack, E., and Freiling, F. C. (2008). Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm. In *First USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET.*

Joshi, S. S. and Phoha, V. V. (2005). Investigating hidden markov models capabilities in anomaly detection. In *Proceedings of the 43rd annual Southeast regional conference-Volume 1*, pages 98–103. ACM.

Kidmose, E., Jensen, S. A., Vejlø, M. H., and Andersen, N. F. (2013). Aau honeyjar - an automated tool for secure deployment and management of a malware test network. Technical report, Studyboard for Electronics and IT, Aalborg University.

Kruse, P. (2012). Csis frigiver gratis zeroaccess værktøj. *csis.dk blog*. Online: `http://www.csis.dk/da/csis/blog/3735/`.

Liu, L., Chen, S., Yan, G., and Zhang, Z. (2008). Bottracer: Execution-based bot-like malware detection. In *Information Security, 11th International Conference*.

Lu, C. and Brooks, R. (2011). Botnet traffic detection using hidden markov models. In *Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research*, page 31. ACM.

Lu, C. and Brooks, R. R. (2012). P2p hierarchical botnet traffic detection using hidden markov models. In *Proceedings of the 2012 Workshop on Learning from Authoritative Security Experiment Results*, pages 41–46. ACM.

Masud, M. M., Al-Khateeb, T., Khan, L., Thuraisingham, B., and Hamlen, K. W. (2008). Flow-based identification of botnet traffic by mining multiple log files. In *Distributed Framework and Applications, 2008. DFmA 2008. First International Conference on*, pages 200–206. IEEE.

Meister, M. (2013). Ddos er nu hverdag for nemid: Nyt angreb trætter nets. *Version 2*. Online: `http://www.version2.dk/artikel/nemid-ramt-af-nyt-ddos-angreb-51590`.

Messmer, E. (December 05, 2012). 'eurograbber' online banking scam netted $47 million. *Network World*.

Nazario, J. and Holz, T. (2008). As the net churns: Fast-flux botnet observations. In *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*, pages 24–31. IEEE.

Ollmann, G. (2009). Botnet communication topologies. Online: `https://www.damballa.com/downloads/r_pubs/WP_Botnet_Communications_Primer.pdf`.

Ourston, D., Matzner, S., Stump, W., and Hopkins, B. (2004). Coordinated internet attacks: responding to attack complexity. *Journal of Computer Security*, 12(2):165–190.

Pedersen, K. (2012). Koobface-programmør rejste også til danmark. *Computerworld*. Online: `http://www.computerworld.dk/art/208808/koobface-programmoer-rejste-ogsaa-til-danmark`.

Porras, P., Saidi, H., and Yegneswaran, V. (2007). A multi-perspective analysis of the storm (peacomm) worm. Technical report, Technical report, Computer Science Laboratory, SRI International.

Rabiner, L. (1989). A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286.

Schipka, M. (2009). Dollars for downloading. *Network Security*, 2009(1):7–11.

Shin, S., Xu, Z., and Gu, G. (2013). Effort: A new host–network cooperated framework for efficient and effective bot malware detection. *Computer Networks*, 57(13):2628–2642.

Silva, S. S. C., Silva, R. M. P., Pinto, R. C. G., and Salles, R. M. (2013). Botnets: A survey. *Computer Networks*, 57(2):378–403.

Sourcefire (2013). Sourcefire product datasheet. Marketing material, online: `https://na8.salesforce.com/sfc/p/#80000000dRH9My3BforYGEqBReQN10GcFZnLvOg=`.

Sourcefire and Snort community (2014). Snort documentation. Webpage, accessed 20. April 2014. Online: `http://snort.org/docs`.

Srivastava, A., Kundu, A., Sural, S., and Majumdar, A. K. (2008). Credit card fraud detection using hidden markov model. *Dependable and Secure Computing, IEEE Transactions on*, 5(1):37–48.

Stinson, E. and Mitchell, J. C. (2007). Characterizing bots remote control behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 89–108. Springer.

Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., Kruegel, C., and Vigna, G. (2009). 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.

Stringer, R. (2008). War in the wires. *Infosecurity*, 5(6):10.

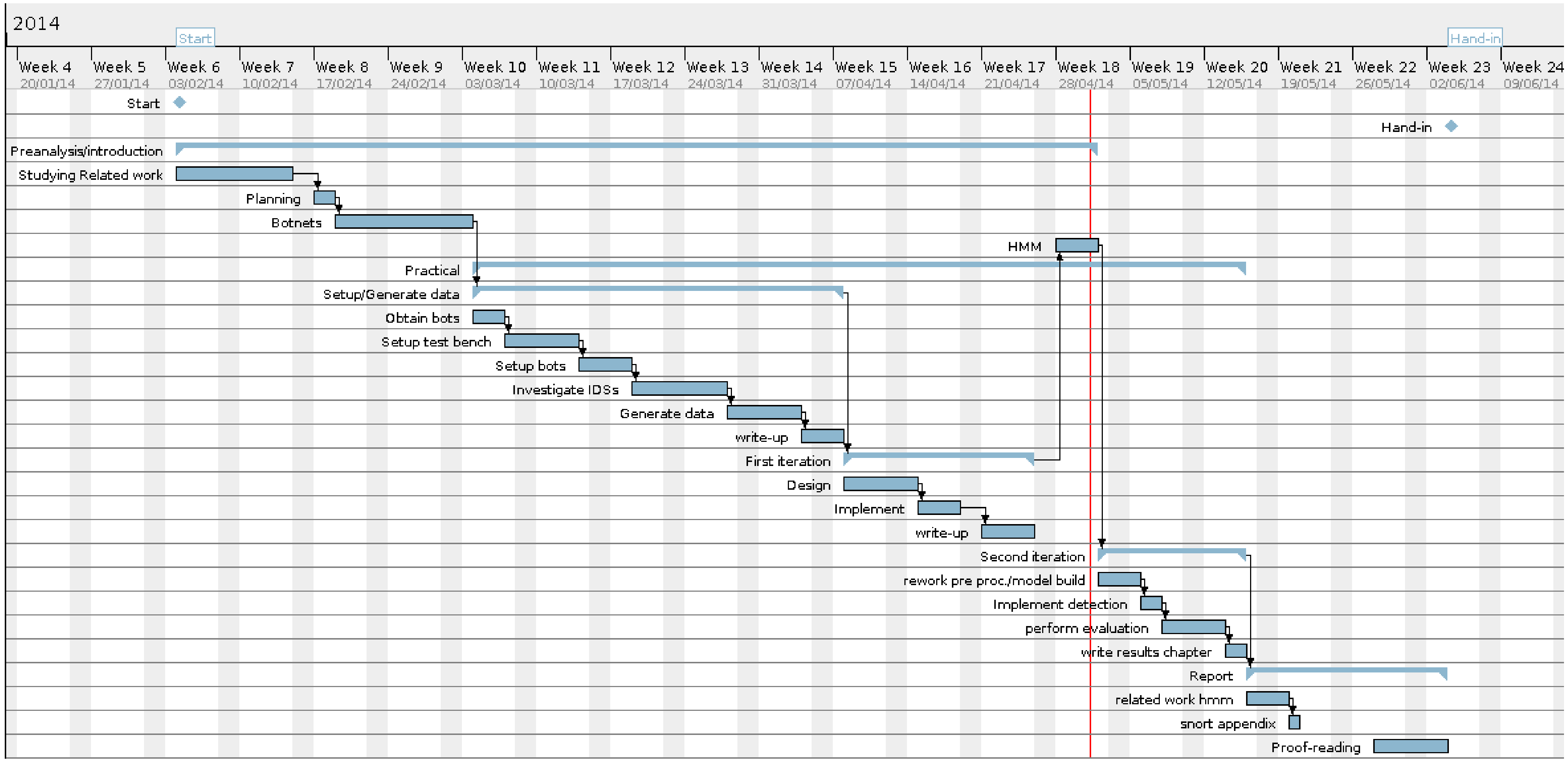
Thomsen, M. K. (2012). Netbanktyve bryder gennem nemid igen: Stjæler 700.000. *Version 2*. Online: `http://www.version2.dk/artikel/breaking-netbanktyve-bryder-gennem-nemid-igen-stjaeler-700000-43471`.

Venkatesh, G. K., Srihari, V., Veeramani, R., Karthikeyan, R., and Anitha, R. (2013). Http botnet detection using hidden semi–markov model with snmp mib variables. *International Journal of Electronic Security and Digital Forensics*, 5(3):188–200.

Zeng, Y. (2012). *On detection of current and next-generation botnets*. PhD thesis, The University of Michigan.

Zeng, Y., Hu, X., and Shin, K. G. (2010). Detection of botnets using combined host-and network-level information. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 291–300. IEEE.

A Project schedule


2014
Start
Hand-in
Week 4 20/01/14
Week 5 27/01/14
Week 6 03/02/14
Week 7 10/02/14
Week 8 17/02/14
Week 9 24/02/14
Week 10 03/03/14
Week 11 10/03/14
Week 12 17/03/14
Week 13 24/03/14
Week 14 31/03/14
Week 15 07/04/14
Week 16 14/04/14
Week 17 21/04/14
Week 18 28/04/14
Week 19 05/05/14
Week 20 12/05/14
Week 21 19/05/14
Week 22 26/05/14
Week 23 02/06/14
Week 24 09/06/14
Start
Hand-in
Preanalysis/introduction
Studying Related work
Planning
Botnets
HMM
Practical
Setup/Generate data
Obtain bots
Setup test bench
Setup bots
Investigate IDSs
Generate data
write-up
First iteration
Design
Implement
write-up
Second iteration
rework pre proc./model build
Implement detection
perform evaluation
write results chapter
Report
related work hmm
snort appendix
Proof-reading

B Installation and set-up instructions

This chapter provides instructions that should enable reconstruction of the used setup. Much inspiration and some direct copy of e.g. commands and configuration files are taken from appendix E of Kidmose et al. [2013].

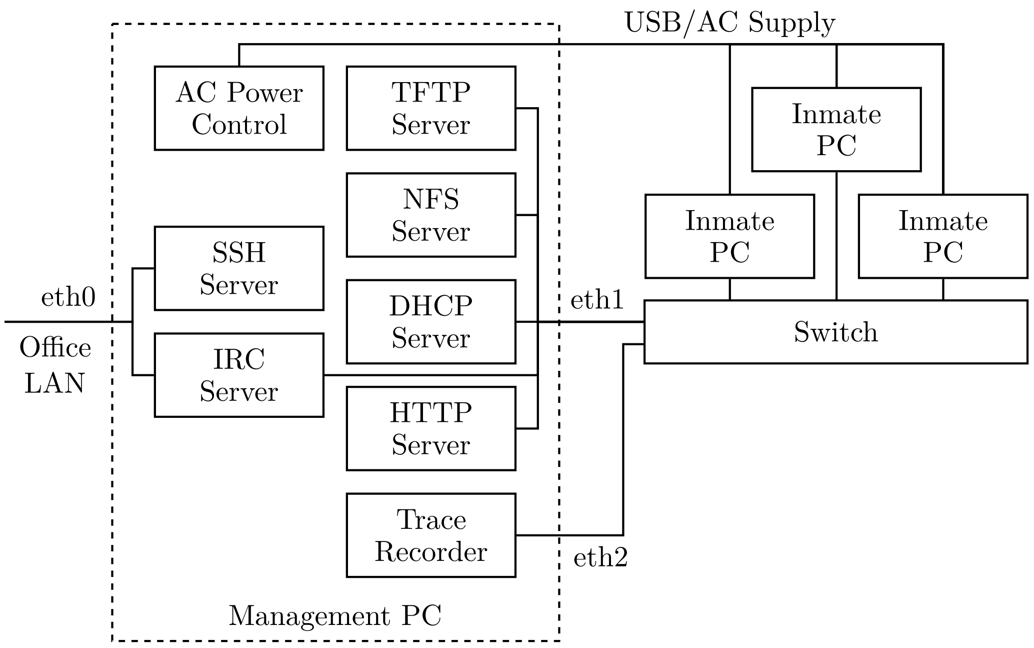


Figure B.1: Diagram for the used setup

As can be seen the setup consists of a management PC and multiple inmate PCs.

B.1 Inmate PCs

Nothing in particular is required from these, beyond a NIC, support for Preboot Execution Environment (PXE) and the option to bot after a power failure. Only identical PCs have been used and not doing so is expected to cause some complications with regards to disk imaging.

B.2 Management PC

The management PC provides the following services: DHCP server, TFPT server, NFS server, traffic monitoring/recording, switch console access and remote shell.

B.2.1 Administrativia

AAU no.:	64260
On-board NIC MAC address:	00-30-05-97-57-3A
Hostname registered with ES IT on MAC:	honeyjar.lab.es.aau.dk
IP resserved for MAC:	172.26.12.57
User:	“manager”
Password:	“1 White Bench”
Samba user:	“storage”
Samba password:	“Not To Be Used”
Installed Operating System (OS):	ubuntu-12.04.4-server-i386, ISO image was obtained from `http://www.ubuntu.com/download/server`.

sudo usermod -a -G dialout manager

B.2.2 Network interfaces

Listing B.1: Configure network interfaces

```
manager@honeyjar:~$ sudo cp /etc/network/interfaces /etc/network/interfaces.
    factory-defaults
manager@honeyjar:~$ sudo emacs /etc/network/interfaces
```

Listing B.2: /etc/network/interfaces

```
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth0
iface eth0 inet dhcp

# IF for DHCP Server in the closed network
auto eth1
iface eth1 inet static
      address 10.10.10.1
      netmask 255.255.255.0

# IF for traffic analysis
auto eth2
iface eth2 inet manual
      up ifconfig eth2 promisc up
```

```
        down ifconfig eth2 promisc down
```

Listing B.3: Configure network interfaces

```
manager@honeyjar:~$ sudo /etc/init.d/networking restart
```

B.2.3 SSH Server

Listing B.4: SSH Server Installation

```
manager@honeyjar:~$ sudo apt-get install openssh-server # install ssh server
# Generate a RSA key pair in case you haven't got some already
me@mymachine:~$ ssh-copy-id -i honeyjar # transfer public key
manager@honeyjar:~$ sudo cp /etc/ssh/sshd_config /etc/ssh/sshd_config.
    factory-defaults
manager@honeyjar:~$ sudo emacs /etc/ssh/sshd_config
```

Listing B.5: /etc/ssh/sshd_config

```
...
PermitRootLogin no
...
PasswordAuthentication no
...
```

Listing B.6: SSH Server Installation

```
manager@honeyjar:~$ sudo restart ssh
```

B.2.4 DHCP Server

Listing B.7: DHCP Server Installation

```
manager@honeyjar:~$ sudo apt-get install isc-dhcp-server
manager@honeyjar:~$ sudo cp /etc/default/isc-dhcp-server /etc/default/isc-
    dhcp-server.factory-defaults
manager@honeyjar:~$ sudo emacs /etc/default/isc-dhcp-server
```

Listing B.8: /etc/default/isc-dhcp-server

```
...
INTERFACES="eth1"
...
```

Listing B.9: DHCP Server Installation

```
manager@honeyjar:~$ sudo cp /etc/dhcp/dhcpd.conf /etc/dhcp/dhcpd.conf.
    factory-defaults
manager@honeyjar:~$ sudo editor /etc/dhcp/dhcpd.conf
```

Listing B.10: /etc/dhcp/dhcpd.conf

```
ddns-update-style none;
option domain-name "example.org";
option domain-name-servers ns1.example.org, ns2.example.org;
log-facility local7;

subnet 10.10.10.0 netmask 255.255.255.0 {
 range 10.10.10.10 10.10.10.30;
 range dynamic-bootp 10.10.10.40 10.10.10.50;
 filename "pxelinux.0";       # PXE image
 option routers 10.10.10.1;
 option broadcast-address 10.10.10.255;
 default-lease-time 600;
 max-lease-time 7200;
}
```

Listing B.11: DHCP Server Installation

```
manager@honeyjar:~$ sudo service isc-dhcp-server restart
```

B.2.5 TFTP Server

Listing B.12: TFTP Server Installation

```
manager@honeyjar:~$ sudo apt-get install tftpd-hpa
```

Files to be served by TFPT:

```
/var/lib/tftpboot/
 |__chain.c32
 |__pxelinux.0
 |__vesamenu.c32
 |__pxelinux.cfg/
 |   |__default
 |   |__01-aa-bb-cc-dd-ee-ff (Too match NIC with mach aa-bb-cc-dd-ee-ff)
 |   |__...
 |__clonezilla-live/
     |__filesystem.squashfs
     |__initrd.img
     |__vmlinuz
```

The syslinux version used in this project is syslinux-4.06. Newer versions seems to fail when using the *.c32 files. Make sure that the files have the right permissions (if not, then try: `chmod 755 /var/lib/tftpboot/ -R`).

B.2.6 NFS Server

```
manager@honeyjar:~$ sudo apt-get install nfs-kernel-server
```

```
/srv/
 |__clonezilla/
     |__clonezilla-live-2.1.1-25-i486.iso
     |__mount/ (mount point for the clonezilla-live-2.1.1-25-i486.iso)
     |__images/
```

```
│
│   └─2013-05-16-ubuntu-12-04-x86/ (As saved by Clonezilla)
└─debian/
```

Listing B.13: /etc/fstab

```
...
# Mount Clonezilla image
/srv/clonezilla/clonezilla-live-2.1.1-25-i486.iso /srv/clonezilla/mount udf,
    iso9660 user,loop 0 0
```

```
sudo mount -a # remount from fstab
```

Specify what to make available on NFS:

Listing B.14: /etc/exports

```
# /etc/exports: the access control list for filesystems which may be
    exported
...
/srv/clonezilla/mount *(ro,async,no_root_squash)
/srv/clonezilla/images *(rw,sync,no_root_squash,subtree_check)
```

```
manager@honeyjar:~$ sudo service nfs-kernel-server restart
```

B.2.7 Samba server

A samba server is used for storage accessible from the inmate PCs.

Listing B.15: Installing Samba server

```
# Create a user in OS
manager@honeyjar:~$ sudo adduser storage
# Install samba:
manager@honeyjar:~$ sudo apt-get install samba
# Create user in Samba
manager@honeyjar:~$ sudo sudo smbpasswd -a storage
# password used: "Let Me In"
# Create dir for content
manager@honeyjar:~$ sudo mkdir /var/lib/samba/usershares/bots
# add manager to sambashare group
manager@honeyjar:~$ sudo usermod -a -G sambashare manager
# verify
manager@honeyjar:~$ groups manager
# make folder accessible to all (Only Storage can login to Samba server)
manager@honeyjar:~$ sudo chmod 777 -R /var/lib/samba/usershares/bots/
# access config file
manager@honeyjar:~$ sudo cp /etc/samba/smb.conf /etc/samba/smb.conf.factory-
    defaults
manager@honeyjar:~$ sudo emacs /etc/samba/smb.conf
```

Listing B.16: /etc/samba/smb.conf

```
...
# Share for inmates
[bots]
  path = /var/lib/samba/usershares/bots
  available = yes
  valid users = storage
  read only = no
  browseable = yes
  public = no
  writable = yes
```

Listing B.17: Installing Samba server

```
manager@honeyjar:~$ sudo restart smbd
# Verify setup
manager@honeyjar:~$ testparm
```

B.2.8 HTTP server

sudo apt-get install apache2 sudo rm /var/www/index.html sudo emacs /etc/apache2/sites-available/default

B.2.9 Alternating Current Power Control

The Alternating Current (AC) power control is implemented with a Universal Serial Bus (USB) interface to an Arduino controlling the power pin of a female USB plug. Plugging an AC power cord controlled by USB power (Commonly reffered to as "USB sparreskinne" in Danish) into the Arduino, the power to the inmates can be cut from the management PC. Source code is available on the attached DVD, see appendix G.

B.3 Switch/network

A Cisco Catalyst 2950 is used to provide network connectivity. The switch is configured through a serial port and the console port on the switch. If the first serial port is used on the server it can be accessed with `sudo screen /dev/ttyS0` and closed with `c-a k y`(ctr+a, k, y). Beware of screen running in the background, `sudo killall screen` to kill all.

We clear any present configuration, disable the spanning tree protocol, and set up port SPAN to enable traffic sniffing. Spanning tree protocol must e disabled because it takes so long that DHCP requests times out for the boot on lan/PXE step, which then fails. [1]

Listing B.18: Switch configuration

```
Switch>enable
Switch#write erase
Switch#configure terminal
Enter configuration commands, one per line.  End with CNTL/Z.
```

[1]`http://docs.oracle.com/cd/E20881_01/html/E20884/glzbk.html` - `http://networkguy.de/?p=470`

```
Switch(config)# spanning-tree mode pvst
Switch(config)# spanning-tree portfast default
Switch(config)# monitor session 1 source interface Fa0/1 , Fa0/3 - 48
Switch(config)# monitor session 1 destination interface Fa0/2
# enter ctrl+z
Switch#exit
```

C Building and running bots

C.1 Obtaining bots

A list of bots used in a collection of papers has been compiled to organise knowledge about their C&C protocol, if the source code or binary was available to the researchers and if only bot or bot and C&C infrastructure was controlled in the research.

Based on this list of bots known from literature, seen in table C.1, a subset of the bots were selected because they are using IRC as C&C protocol, their source code have been available suggesting that they are among the easier to obtain, understand and configure, and other researchers have been able to execute them in a set-up where the whole botnet was controlled. The subset of bots selected as potentially useful for this work is listed in table C.2.

Agobot	Forbot
IRCBot	Jrbot
Phatbot	RBot
Reptilebot	Rxbot
SDBot	spybot
Wargbot	

Table C.2: Potentially useful bots

With the list of potentially usefull bots searches onWorld-Wide Web (WWW) was conducted, resulting in a collection of bots from `http://www.hackforums.net`, which contains the source code for the following bots:

- RBot
- Reptilebot
- Rxbot
- SDBot
- spybot

Name	Structure	Reference	Pkg.	SU
Agobot	IRC	Gu [2008], Liu et al. [2008]	SC	Inf.
Forbot	IRC	Liu et al. [2008]	SC	Inf.
IRCBot	IRC	Gu [2008]	SC	Inf.
Jrbot	IRC	Liu et al. [2008]	SC	Inf.
Phatbot	IRC	Gu [2008]	SC	Inf.
RBot	IRC	Gu [2008], Masud et al. [2008], Zeng [2012]	SC	Inf.
Reptilebot	IRC	Liu et al. [2008]	SC	Inf.
Rxbot	IRC	Gu [2008], Liu et al. [2008]	SC	Inf.
SDBot	IRC	Gu [2008], Liu et al. [2008], Masud et al. [2008]	SC	Inf.
spybot	IRC	Zeng [2012]	SC	Inf.
Wargbot	IRC	Gu [2008]	SC	Inf.
rbot	IRC	Zeng [2012]	Bin	Bot
spybot	IRC	Zeng [2012]	Bin	Bot
Gaobot	IRC	Gu [2008]	n/a	n/a
Gtbot	IRC	Gu [2008]	n/a	n/a
UrBot	IRC	Gu [2008]	n/a	n/a
UrXBot	IRC	Gu [2008]	n/a	n/a
JarBot	IRC	Shin et al. [2013]	SC	n/a
PhatBot	IRC	Shin et al. [2013]	SC	n/a
storm	hybrid P2P	Zeng [2012]	Bin	Bot
waledac	hybrid P2P	Zeng [2012]	Bin	Bot
Waledac	HTTP, P2P	Shin et al. [2013]	Bin	n/a
bobax	HTTP	Zeng [2012]	Bin	Inf.
Graybird	Custom	Liu et al. [2008]	SC	Inf.
BiFrost	Custom	Shin et al. [2013]	Bin	n/a
Cone	Custom	Shin et al. [2013]	Bin	n/a
Flux	Custom	Shin et al. [2013]	Bin	n/a
Lizard	Custom	Shin et al. [2013]	Bin	n/a
nuclearRat	Custom	Shin et al. [2013]	Bin	n/a
PanBot	Custom	Shin et al. [2013]	Bin	n/a
Penumbra	Custom	Shin et al. [2013]	Bin	n/a
Polip	Custom	Shin et al. [2013]	Bin	n/a
Sality	Custom	Shin et al. [2013]	Bin	n/a
SeedTrojan	Custom	Shin et al. [2013]	Bin	n/a
TBBot	Custom	Shin et al. [2013]	Bin	n/a
Nugache	P2P	Liu et al. [2008]	Bin	Bot
Storm/Peacomm	P2P	Shin et al. [2013]	Bin	n/a
agobot	n/a	Stinson and Mitchell [2007]	SC	Inf.
DSNXbot	n/a	Stinson and Mitchell [2007]	SC	Inf.
evilbot	n/a	Stinson and Mitchell [2007]	SC	Inf.
G-SySbot	n/a	Stinson and Mitchell [2007]	SC	Inf.
sdbot	n/a	Stinson and Mitchell [2007]	SC	Inf.
spybot	n/a	Gu [2008], Stinson and Mitchell [2007]	SC	Inf.

Table C.1: Table of bots found in literature. Pkg.: Packaging, SC: Source Code, Bin: Binary, SU: Set-Up, Inf.: Bot and C&C Infrastructure, Bot: Bot only.

C.2 SDBot

The folowing variations of sdbot was obtained: sdbot-b0rg-by-okasvi, sdbot-ntpass-codefix-nils-22.10.03, sdbot-syn-nbspread, sdbot04b, sdbot04b, sdbot05a, sdbot05a, sdbot05b-AE, sdbot05b-ago, SDbot05b-getadm, sdbot05b, sdbot05b[skbot]_mods_by_sketch, sdbot05b_skbot___mods_by_sketch, sdbot05b_syn_&_nick, sdbotconstant_nick_mod, sdboti3s, SDBOTNTSharesHardCorePrivM0D, sdbotvnc, Sdbot_Hardcore_Mod_By_StOner, sdbot_syn, sdbot_synx, sdbot_syn_secure_1, SDBot_with_NB spreader.

Among these `sdbot05b` and `SDBot_with_NB spreader` have been selected for further use. The first is selected due to the assumption that it is the most recent "official" release and the latter is selected arbitrarily.

Both come with windows bat scripts for building using either min-GW (`make-lcc.bat`) or lcc (`make-mingw.bat`). `lcc` is obtained from `http://www.cs.virginia.edu/~lcc-win32/lccwin32.exe` and used for compiling both.

Running `make-lcc.bat` fails because `icmp.lib` is missing in the compiler libraries, which can be solved using `pedump` and `buildlib` found in `c:\llc\bin\` in the following way: [1]

Listing C.1: Obtaining `icmp.lib`

```
cd c:\llc\lib\
pedump /EXP c:\windows\system32\icmp.dll > icmp.exp
buildlib icmp.exp icmp.lib
```

For `sdbot05b\make-lcc.bat -o` need to be replaced with `-O` or `-o sdbot05b.obj` for proper naming of the object file output by the compiler.

C.3 Spybot

version: spybot 1.3
make: `make spybot.bat`, using lcc
commands: `login botpass`, `info`

[1] `https://groups.google.com/forum/#!topic/comp.compilers.lcc/orye0QftgQI`

D Snort setup

To run Snort an engine and a rule set is required. The Snort engine is open source and can be downloaded from `http://snort.org/snort-downloads`. Rule can be written by one-self, which is asumed to be a big task, or rulesets can be downloaded and added. This task is commonly automated with the a script named pulledpork. Rules from SourceFire, the team behind Snort, and from `https://www.emergingthreatspro.com` is used.

D.1 Installing Snort engine

Listing D.1: Install prerequisites

```
sudo apt-get install gcc flex bison zlib1g zlib1g-dev libpcap0.8-dev
    libpcre3 libpcre3 libpcre3-dev tcpdump libdumbnet-dev
sudo su
cd /usr/local/src/
# download  daq-2.0.2.tar.gz
# from http://www.snort.org/snort-downloads
# to /usr/local/src
tar zxvf daq-2.0.2.tar.gz
cd daq-2.0.2/
./configure
make
make install
# update dynamic linker run-time bindings:
ldconfig -v /usr/local/lib
ldconfig -v /usr/local/lib64
```

Listing D.2: Install Snort engine

```
# download snort-2.9.6.0.tar.gz  to /usr/local/src
tar zxvf snort-2.9.6.0.tar.gz
cd snort-2.9.6.0/
./configure --prefix=/usr/local/snort --enable-sourcefire
make
make install
ln -s  /usr/local/snort/bin/snort /usr/local/bin/snort
cp -r etc/ ../../snort/
mkdir ../../snort/etc/rules
mkdir /usr/local/snort/lib/snort_dynamicrules
touch /usr/local/snort/etc/rules/snort.rules
touch /usr/local/snort/etc/rules/local.rules
touch /usr/local/snort/etc/rules/black_list.rules
touch /usr/local/snort/etc/rules/white_list.rules
touch /usr/local/snort/etc/rules/iplists
mkdir /usr/local/snort/so_rules
mkdir /usr/local/snort/preproc_rules
mkdir /var/log/snort/
mkdir /var/log/snort/
emacs /usr/local/snort/etc/snort.conf
```

The snort.conf used in this project is available on the attached DVD, see appendix G, in git-reps/nds10-code/data/cfg-copies/snort.conf.

D.2 Installing pulledpork

Listing D.3: Installing pulledpork

```
# install pulledpork
apt-get install subversion perl libcrypt-ssleay-perl liblwp-protocol-https-
    perl
cd /usr/local/src/
svn checkout http://pulledpork.googlecode.com/svn/trunk pulledpork
cd pulledpork/
ln -s /usr/local/src/pulledpork/pulledpork.pl /usr/local/bin/pulledpork
sudo chmod +x /usr/local/bin/pulledpork
cp -r /usr/local/src/pulledpork/etc /usr/local/pulledpork/
cd /usr/local/src/pulledpork/
emacs /usr/local/pulledpork/pulledpork.conf
```

The pulledpork.conf used in this project is available on the attached DVD, see appendix G, in git-reps/nds10-code/data/cfg-copies/pulledpork.conf.

D.3 Running Snort

Listing D.4: Commands for running pulledpork and Snort

```
# to update the ruleset (Or obtain them the first time):
sudo pulledpork -c /usr/local/pulledpork/pulledpork.conf -o /usr/local/snort
    /etc/rules/snort.rules

# To run snort on some.pcap:
snort -c /usr/local/snort/etc/snort.conf -r some.pcap -N -l .
```

E Recorder setup

Benign traffic is recorded with a Barracuda Ethernet Tap, for which a data sheet is provided on the attached DVD, see appendix G. A machine with five Gigabit NICs is used for the recorder. To capture traffic in both directions, two NICs are bonded to form one logical NIC, on which traffic can be recorded.

E.1 Bonding interfaces

The implementation of traffic mixing relies on the Python module scapy for processing network packets, which requires PCAP files of the older libpcap format rather than the newer pcap-ng. All known traffic capture utilities (tcpdump, tshark and dumpcap) can only store traces in the libpcap format if recording is done on a single interface. Hence bonding of NICs is applied.

Listing E.1: Installation of modules for NIC bonding

```
recorder@recorder:~/traces$ sudo apt-get install ifenslave-2.6
recorder@recorder:~/traces$ sudo cp /etc/modules /etc/modules.orig
recorder@recorder:~/traces$ sudo emacs /etc/modules
```

Listing E.2: contents of /etc/modules

```
# /etc/modules: kernel modules to load at boot time.
#
# This file contains the names of kernel modules that should be loaded
# at boot time, one per line. Lines beginning with "#" are ignored.

loop
lp
bonding
```

Listing E.3: Configuring bonding

```
recorder@recorder:~/traces$ sudo /etc/init.d/networking stop
recorder@recorder:~/traces$ sudo modprobe bonding
recorder@recorder:~/traces$ sudo cp /etc/network/interfaces /etc/network/
    interfaces.orig
recorder@recorder:~/traces$ sudo emacs /etc/network/interfaces
```

Listing E.4: /etc/network/interfaces with eth1 configure by DHCP, eth2 and eth3 in promiscous mode bonded as the bond0 interface.

```
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth1
iface eth1 inet dhcp

#eth2 is manually configured, and slave to the "bond0" bonded NIC
auto eth2
iface eth2 inet manual
      up ifconfig $IFACE promisc up
      down ifconfig $IFACE promisc down
bond-master bond0

#eth3 ditto, thus creating a 2-link bond.
auto eth3
iface eth3 inet manual
      up ifconfig $IFACE promisc up
      down ifconfig $IFACE promisc down
bond-master bond0

# bond0 is the bonding NIC and can be used like any other normal NIC.
auto bond0
iface bond0 inet manual
      up ifconfig $IFACE promisc up
      down ifconfig $IFACE promisc down
bond-mode balance-rr
bond-miimon 100
bond-slaves eth2 eth3
```

Listing E.5: Start networking again, stopped for configuration

```
recorder@recorder:~/traces$ sudo /etc/init.d/networking start
```

E.2 Capture utility started with cron job

The traces are recorded with dumpcap, a commandline tool which is part of wireshark. Cron jobs are used to automatically start dumpcap.

Listing E.6: Install Wireshark, cron

```
recorder@recorder:~$ sudo apt-get install wireshark-common # for dumpcap
recorder@recorder:~$ sudo apt-get install cron # for automated starup every
    morning
recorder@recorder:~$ touch traces
recorder@recorder:~$ touch log
```

Listing E.7: Add a cron tab

```
recorder@recorder:~$ crontab -e
# for a job running at 07:00 Monday through friday enter the following line:
00 07 * * 1-5 ./record.sh
```

The following script is called by cron to start recording for 10 hours:

Listing E.8: ./record.sh - shell script for recoding traffic

```
#!/bin/bash

ts=$(date +%Y-%m-%d_%H-%M-%S)
pcap="traces/"$ts"_benign.pcap"
log="logs/"$ts".log"

#/usr/local/bin/tshark -i bond0 -w $pcap -a duration:36000 > $log 2>&1
/usr/local/bin/dumpcap -P -i bond0 -w $pcap -a duration:36000 > $log 2>&1
```

F Data summary

This section provides an overview of the data recorded for training and testing the proposed method. The test and training data are obtained by applying the same procedures, but with different bots for the synthetic traces and on different days for the benign.

The synthetic bot traffic is everything observed on the inmate network, while a host running vanilla windows XP is being put through the infection life-cycle according to the procedure described above in section 3.2.

The benign traffic is any wired traffic to or from the PCs of four students during an ordinary day at Aalborg University, as further discussed in section 3.3 above.

The mixed traffic is generated by merging synthetic bot traffic and benign traffic in time and used host addresses, as described in section 3.4. For test and training data the respective synthetic and benign data is reused.

F.1 Training data

F.1.1 Synthetic bot traffic

- Bot: sdbot05b
- Start time: Thu Apr 17 13:32:40 2014
- End time: Thu Apr 17 13:40:50 2014
- Traffic trace, file size: 329,593 bytes
- Traffic trace, packets: 1,349
- Alerts, file size: 3,475 bytes
- Alerts, count: 16

F.1.2 Benign traffic

- Start time: Mon May 12 07:00:05 2014
- End time: Mon May 12 17:00:04 2014
- Traffic trace, file size: 1,209,602,363 bytes
- Traffic trace, packets: 2,544,906

- Alerts, file size: 630,042 bytes
- Alerts, count: 4,708

F.1.3 Mixed traffic

- Benign input: See above
- Synthetic input: See above
- Start time: Mon May 12 07:00:05 2014
- End time: Mon May 12 17:00:04 2014
- Traffic trace, file size: 1,214,033,235 bytes
- Traffic trace, packets: 2,548,536
- Alerts, file size: 630,042 bytes
- Alerts, count: 3,075

F.2 Test data

F.2.1 Synthetic bot traffic

- Bot: spybot1.3
- Start time: Mon Apr 28 09:40:53 2014
- End time: Mon Apr 28 09:49:01 2014
- Traffic trace, file size: 193,806 bytes
- Traffic trace, packets: 1,309
- Alerts, file size: 845,148 bytes
- Alerts, count: 9

F.2.2 Benign traffic

- Start time: Tue May 13 07:00:06 2014
- End time: Tue May 13 17:00:05 2014
- Traffic trace, file size: 4,782,381,604 bytes
- Traffic trace, packets: 5,504,054
- Alerts, file size: 2,938,146 bytes
- Alerts, count: 13,990

F.2.3 Mixed traffic

- Benign input: See above
- Synthetic input: See above
- Start time: Tue May 13 07:00:06 2014
- End time: Tue May 13 17:00:05 2014
- Traffic trace, file size: 4,785,578,479 bytes
- Traffic trace, packets: 5507314
- Alerts, file size: 845,148 bytes
- Alerts, count: 4,068

G DVD Content

A DVD, labelled “14gr1021” is attached to the report. The general structure is as follows, with important content highlighted:

- master-thesis-egon-kidmose.pdf: This report.
- references: Copies of used references, where available.
- git-reps: All code, including LaTeX for this report, under version control with Git
  - nds10-bots: Bot source code
  - nds10-code: All the implemented code of this project.
    - data-generation: Procedures for generating synthetic traffic.
      - procedure.py: Procedure for SDBot.
      - spybotprocedure.py: Procedure for SpyBot.
    - data; Data processing and implementation of the proposed method,
      - buildmodel.py: Procedure for building a model.
      - detection.py: Procedure for performing detection with a model.
      - mixtraffic.py: Procedure for mixing benign and synthetic malicious traffic.
  - nds10-report: Latex source for this report.
- synthetic-traces: Synthetic traffic traces.[1]

[1]Benign and mixed traffic traces are omitted due to their file sizes being larger than DVD capacity.

H Used acronyms

AC Alternating Current

C&C Command and Control

CSV Comma Separated Values

DDoS Distributed Denial of Service

DGA Domain Generation algorithm

DHCP Dynamic Host Configuration Protocol

DNS Domain Name System

DoS Denial of Service

DPI Deep Packet Inspection

HIDS Host-based Intrusion Detection System

HMM Hidden Markov Model

HsMM Hidden semi-Markov Model

HTTP Hypertext Transfer Protocol

IDS Intrusion Detection System

IP Internet Protocol

IRC Internet Relay Chat

ISP Internet Service Provider

LAN Local-Area Network

MaaS Malware-as-a-Service

MAC Media Access Control

MIB Management Information Base

NAT Network Address Translation

NFS Network File System

NIC Network Interface Card

NIDS Network-based Intrusion Detection System

OCSVM One Class Support Vector Machine

OS Operating System

P2P Peer-to-Peer

PCAP Packet Capture

PXE Preboot Execution Environment

ROC Receiver Operating Characteristic

SaaS Software-as-a-Service

SNMP Simple Network Management Protocol

SSH Secure Shell

SVM Support Vector Machine

TCP Transmission Control Protocol

TFTP Trivial File Transfer Protocol

TTL Time To Live

UDP User Datagram Protocol

USB Universal Serial Bus

WWW World-Wide Web