Detecting Botnets Using Hidden Markov Models on Network Traces

Wade Gobel
371 Fairfield Way, Unit 2155
Dept. of Computer Science and Engineering
University of Connecticut
Storrs, CT 06269, USA

Abstract. One of the most prevalent problems in modern internet security is the botnet – large numbers of computers running the same malicious, self-propagating program without their users' knowledge. Bot programs communicate with their (human) botmaster, who can command them to stage distributed denial of service attacks, send spam, commit click fraud, send back user passwords, or any number of other illicit actions [1, 6, 10, 11, 14]. The analysis of bots and botnets is still a relatively new field. One may observe that if most of the bots in a botnet can be identified, and if the necessary communication with the botmaster can be blocked, the botnet loses its power [6]. This paper presents a new approach of identifying botnets using data from captured network packets by modeling the network with a Hidden Markov Model (HMM) and then comparing HMMs generated this way to detect covert coordination between computers.

1 Introduction

Bots and botnets pose a challenging problem to the security of any computer connected to the internet. A bot program is able to transmit itself to other computers through security vulnerabilities in web browsers or by piggybacking on otherwise legitimate programs intended for download [3]. Bots are particularly insidious because unlike worms and viruses, they run in the background, taking part in malicious activities only intermittently – at the command of their botmaster – and thereby avoid detection. A successful bot program does not use enough of the computer's processing power to noticeably affect the speed of other programs, as the user could notice this and realize there is a problem.

When large networks of bot-infected computers are given the appropriate command, the effects can be devastating for individual users, online businesses, and entire corporations. Protecting computers against large-scale bot attacks, or working to mitigate attacks once they are underway, in no way removes the threat of botnets. Recent experiments in combating the botnet problem involve identifying the inter-computer coordination inherent in bot behavior. Since it is the combined and coordinated effort of a botnet that gives it power, removing the means of communicating – by blocking the necessary communication channels – is a significant step towards beating a botnet.

Of course, the members of a botnet must be identified first. Since the bots use the internet to communicate with their botmaster, it seems logical that network data should be collected and analyzed for signs of bot-like collaboration. Once the data are collected, there should hopefully be some way of measuring the similarity of two computers' network traces such that bot-run processes on different computers appear much more similar than human browsing behavior on those computers. This paper introduces an original process for identifying the collaborating members of a botnet: the events that generate some captured network traces are approximated using a mathematical construct called a Hidden Markov Model (HMM), and the "distance" between two HMMs can then be calculated to quantify the similarity of network traces run on different computers.

The rest of this paper is organized as follows: section 2 presents a sampling of previous work related to Hidden Markov Models and botnet detection, section 3 describes the behavior of bots and botnets in greater detail, section 4 introduces Hidden Markov Models and the Baum-Welch algorithm, and section 5 describes a possible procedure for collecting the data necessary to test the process described here.
2 Related Work

Although the botnet problem is still relatively new (and unsolved), a number of other research projects have already been dedicated to the detection, identification, and potential disruption of botnets, as well as projects analyzing the security-related uses of Hidden Markov Models.


The program BotSniffer is discussed in [6]: this program strives to identify bots and botnets by examining the similarity of different computers' responses to IRC messages, using both the times when the responses are generated and the type of response that each machine gives. Similarly, [1] tracks the changes in the log file sizes of various computers, where similar changes in the log files of different machines indicate potential bot-coordinated activity. [10] tracks machines connected to servers with large amount of potential bot-generated traffic, and groups these machines into botnets based on a similarity heuristic. And [11] uses the artificial learning process of support vector machines to identify bot traffic, using different combinations of data from captured network traffic to try to identify the subset of data that best indicates the presence or absence of bot communication.

3 Bots

Even with today's well-established and powerful antivirus programs and well-patched browsers, much of modern anti-malware software is, at best, "one step behind" the malware producers [11]. Many of the programs meant to protect against malware use a signature-based approach [11], in which telltale signs of known security threats are taken as indicators of a security breach. Of course, software that works in this way must be constantly updated: as new brands of malicious software spread, the known signatures must be immediately updated, or else a computer will remain vulnerable until the update is made.

Although there exist several advanced anti-malware programs today, many users do not buy this software, or fail to update it regularly. Therefore, even with anti-virus products available, the number of vulnerable systems in existence is still large enough to allow for botnets of significant size – into the tens of thousands in some cases [1, 11, 14]. The larger the botnet, the more power it wields, and the exploitation of numerous individual security flaws leads to problems much larger than a stolen password or identity.

3.1 Bot Creation

Bots are astoundingly easy to write. For the sample network traces described in section 5, a Python bot called blackenergybot.py (written by David Walluck with modifications by Wade Gobel) was able to regularly request a URL and discern the command specified by the botmaster, in under 50 lines. Those who write malware occasionally make their code publicly available, allowing other malware programmers to add their own features and security exploits. By the time a malicious program infects a computer, it can have had any number of authors [3, 11, 14].

Once a botmaster has discovered a browser's security vulnerability (or decided to continue exploiting an old one), he can write a bot that, once installed on a user's computer, connects back to the botmaster's machine, accepting commands and relaying information as directed. Bots were initially created to regulate Internet Relay Chat (IRC) channels, allowing for many more users than if each conversation required a human monitor [3]. In fact, many malicious bots use IRC as their primary means of communication because of its speed, its ability to deal with large numbers of users, as well as protocols that provide a level of anonymity [3]. Once an IRC bot is installed on a computer, it must keep an IRC channel open to its botmaster while waiting for commands. It is also possible for bots to connect to a website, and read the botmaster's commands from there. This approach does not require that the bots remain idle on an IRC
channel, and better mimics the habits of real user browsing (although not perfectly because of its regularity). Unless otherwise noted, the bots discussed in this paper (where communication method is relevant) are HTML-requesting bots, and not the standard IRC-bots.

### 3.2 Brief Explanation of Buffer Overflow

As a reference, this section explains the basic process behind using a simple but common browser security exploit: the buffer overflow.

When a program written in C is executed, the memory required to run the program is organized into three "chunks" for keeping track of the program's code, the variables currently in use, and the call stack [13]. The call stack is used to track the last unexecuted line of code after a method call, and allows for backtracking once a method finishes execution. Assume that a program allocates a certain amount of memory to some variable A (the "buffer"), and then copies information from variable B into A until all of the data has been transferred. If the amount of memory that B requires is much larger than the memory allocated for A, the copying process will overwrite other data in memory (the "overflow"). If the data in B are consistent with the way that C stores information in memory at runtime, this overwriting process could either cause the program to crash, or could launch a separate process and begin running a malicious script, or begin downloading software on the user's computer.

As many browsers are riddled with code that tacitly trusts data coming from an outside source, any one of these security vulnerabilities can be exploited to take malicious actions – with no warning to the user – and thereby successfully compromising the targeted machine.

### 3.3 Bot Propagation

Once a bot has been written, the botmaster can integrate the bot code into a program that exploits a browser's security vulnerability. However, if the only process by which a bot can be downloaded is through a single website, the spread of the bot would be very limited, as it is highly unlikely that a non-malicious site would allow bots, and a malicious site may not receive sufficient traffic to allow for a botnet of any significant size. Instead, the bots accelerate their own propagation by scanning for other computers on a network, and sending the bot code to vulnerable systems for download. This way, each new infected machine increases not only the size but also the range of the botnet.

It is important to keep in mind that not only are users' machines vulnerable, but servers and other devices meant to regulate the internet are as well. If a server becomes compromised, the botmaster can use it to expand the range of his botnet even further. In the case of IRC bots, if one or more IRC servers become compromised, the botmaster can relay his commands through several layers of machines, making it exceedingly more difficult to track the message back to its source. In fact, botmasters controlling IRC bots frequently change the path of communication they use so as to further confound tracking efforts [14].

### 3.4 Bot Attacks

Once sufficient numbers of bots are installed on servers and personal computers, and are actively communicating with their botmaster, he can command them to take a number of malicious actions. The botmaster can even command select groups of bots to take different actions, covering multiple bases simultaneously. As mentioned in section 3.3, bots can be told to scan for other vulnerable computers, downloading and installing the bot program if possible. The truly malicious actions that bots can perform are their various modes of attack, such as sending spam, staging DDoS attacks, committing click fraud, or sending back user information for identity theft. The following is a brief explanation of each of these modes of attack.

- **Spam.** One of the services bots can offer their botmaster is the sending of unsolicited e-mails to random e-mail accounts. This allows botmasters to "rent out" the service of their botnet to individuals or covert

---

1 Large numbers of idle users, or users that respond simultaneously and in a similar fashion to IRC messages, may warn IRC monitors of a potential botnet.
organizations that wish to send spam. Spam is therefore a profitable option for the botmaster; and the more computers in the botnet, the more processing power the botnet has, and the more spam it can send.

- **DDoS.** Distributed denial-of-service attacks occur when a large number of computers send multiple requests to the same website or server, thereby overloading its capacity and causing the online service to come to a standstill. If a company relies on a website in order to conduct business, temporarily blocking the website in this way effectively prevents the company from making money. The botmaster can then demand a ransom from the company before calling off the bots.

- **Click fraud.** Since bots are able to access the internet, one of the online actions they can simulate is clicking an advertisement on the botmaster's website. Companies that post online advertisements usually pay by the number of clicks on their ad, as this is a quantity that can be easily tracked, and it encourages the owner of the website to host appealing online material. However, as long as bots do not click the ads much more frequently than a normal user would, there is no way to detect whether a given click originated from a human or a bot. Click fraud is therefore another avenue by which botmasters can profit.

- **Identity theft.** Some bots are tailored to send back user account information, such as passwords or PayPal account numbers. Bots can register usernames and passwords by installing what is known as a keylogger – a program that registers the buttons pressed on the keyboard – and by keeping track of the strings of characters entered in different fields on webpages. Using this information, botmasters can gain access to bank account numbers and other financial services, thereby further funding their botnet.

Therefore, bots are widespread, difficult to detect before they begin spreading, and popular because of their profitability.

### 4 Hidden Markov Models

A Hidden Markov Model, or HMM, is a finite-state machine that transitions probabilistically between its states at discrete time steps. An HMM always has exactly one of its finite states selected as the "current" state. At each time step, the HMM transitions to another state, and the new current state generates some observable output – an "observation". What makes the model "hidden" is that an observer, given only the observations generated by the HMM, cannot (usually) deduce the current state with absolute certainty.

To define an HMM, three elements must be specified: 1) the start-state probabilities, which specify the chances of starting in each of the HMM's states on the initial time step; 2) the transition probabilities, which specify the probability of transitioning from any state of the HMM to any other state; and 3) the observation probabilities, which specify the probability of making any observation from any current state.

Each of the transition probabilities can be read, "If the current state is x, the probability of transitioning to state y on the next time step is <value>". It is possible to transition to the same state on consecutive time steps: in this case, x = y. It is also possible that some transitions between states are guaranteed never to happen: in this case, the probability specified by <value> would be 0.

It must always be possible to start in one of the HMM's states on the initial time step, as it must always be possible to transition to a new current state and generate a single observation. Therefore, there exists a one-to-one correlation between the generated observations and the various current states that the HMM "passes through". Using this one-to-one correlation, it is possible to construct an HMM that has a (locally) maximized probability of generating a given sequence of observations.

The Baum-Welch algorithm takes a sequence of observations and the specification of an HMM, and generates the specification of a new HMM with a greater probability of producing the given sequence. This process can then be repeated with the new HMM until the values in the HMM specification converge – i.e. they do not change enough on consecutive iterations of the algorithm to significantly affect the behavior of the HMM. At the point of convergence, the resulting HMM has a maximized probability of generating the given sequence (given the original HMM). Baum-Welch never changes the number of states in an HMM – this number must be provided in the original HMM. An explanation of the Baum–Welch algorithm follows.
4.1 The Baum-Welch Algorithm

First, some definitions:

- Enumerate the states of the HMM with integer values 1, 2, \ldots, N.
- Enumerate the observations that the HMM can generate with integer values 1, 2, \ldots, M.
- Let the initial time step be 0, and let T be the (positive integer) time step when the final observation in the sequence was generated.
- Define the start-state probability \( s_i \) as the probability of starting in state i at the initial time step.
- Define the transition probability \( a_{ij} \) as the probability of transitioning to state j at time step \( t + 1 \) if the current state is state i at time step t.
- Define the observation probability \( b_{ij} \) as the probability of making observation j if the current state is state i.

With these definitions in hand, it is now necessary to define the four functions \( \alpha_t(j) \), \( \beta_t(i) \), \( \gamma_t(i) \), and \( \xi_t(i,j) \) before defining the algorithms used to create the new, updated HMM.

The forward variable \( \alpha_t(j) \) represents the probability of making the first \( t \) observations in the given sequence, and ending in state j. If \( t = 0 \), \( \alpha_0(j) \) is merely the probability of starting in state j and making the first observation. If the first observation is observation k, \( \alpha_0(j) = s_j \cdot b_{jk} \). If \( t > 0 \), then \( \alpha_t(j) \) is the probability of passing through some series of states, ending in any state i at time step \( t - 1 \), transitioning to state j at time step t, and generating the observation made at time step t. The first step of this path – transitioning to any state i by time step \( t - 1 \) – is merely the sum of \( \alpha_{t-1}(i) \) for all values of i. The subsequent transition from state i to state j is provided by the transition probability \( a_{ij} \), while the probability of making the observation k at time t (given that the current state is state j) is \( b_{jk} \). Therefore, \( \alpha_t(j) \) can be defined recursively as follows, assuming the observation made at time t is observation k:

\[
\alpha_t(j) = \begin{cases} 
  s_j \cdot b_{jk} & \text{if } t = 0 \\
  \sum_{i=1}^{N} \alpha_{t-1}(i) \cdot a_{ij} \cdot b_{jk} & \text{if } t > 0 
\end{cases}
\]  

(1)

The backward variable \( \beta_t(i) \) can be calculated recursively like the forward variable, but it includes all the states and observations not covered by \( \alpha_t(i) \). That is, \( \beta_t(i) \) represents the probability of transitioning to any state j from state i and generating the final \( T - t \) observations in the given sequence, disregarding the transition into and the observation made from state i. In the simple case, \( t = T \), and \( \beta_T(i) \) represents the probability of starting in state i and making no further transitions or observations, so \( \beta_T(i) \) can simply be defined as 1. If \( t < T \), \( \beta_t(i) \) is the probability of transitioning to any state j from state i, generating the observation at time step \( t + 1 \), and then making the last \( T - t - 1 \) observations from state j. As with the forward variable, this last step – making the last \( T - t - 1 \) observations – is a rephrasing of the backward variable with different parameters: \( \beta_{t+1}(j) \) (summed over all values of j). Similarly, the probability of transitioning from state i to any state j and then making observation k at time step \( t + 1 \) is the sum of \( a_{ij} \cdot b_{jk} \) over all values of j. The backward variable is defined as follows:

\[
\beta_t(i) = \begin{cases} 
  1 & \text{if } t = T \\
  \sum_{j=1}^{N} \beta_{t+1}(j) \cdot a_{ij} \cdot b_{jk} & \text{if } t < T, \text{ and the observation made at time step } t + 1 \text{ is observation } k 
\end{cases}
\]  

(2)

The variable \( \gamma_t(i) \) represents the probability that the current state is state i at time step t, given all the observations in the sequence. Conceptually, \( \gamma_t(i) \) can be calculated by dividing the probability of being in state i at time step t by the probability of being in any state at time step t. Using the forwards and backward variables, note that \( \alpha_t(i) \cdot \beta_t(i) \) is the probability of generating the first \( t + 1 \) observations of the given sequence and ending in state i, then transitioning out of state i and generating the last \( T - t \) observations. In order to find the probability of being in any state at time step t, the product of \( \alpha_t(j) \) and \( \beta_t(j) \) must be

---

2 Since the end of the sequence of observations has been reached when \( t = T \), it is considered possible to remain in any state of the HMM, thereby generating no observations while maintaining consistency with the observations generated.
calculated for every possible value of \( j \). The value of \( \gamma(i) \) can now be explicitly defined in terms of calculable values:

\[
\gamma(i) = \frac{\alpha(i) * \beta(i)}{\sum_{j=1}^{N} \alpha(j) * \beta(j)} .
\]

Note that

\[
\sum_{i=0}^{T} \gamma(i)
\]

is the expected number of times that the given HMM transitions out of state \( i \) in the given observation sequence, as each state generating observations at time steps 0 to \( T - 1 \) has a following state.

Also note that

\[
\sum_{i=0}^{T} \gamma(i)
\]

is the expected number of times that the given HMM reaches state \( i \) in total. These values will be used when updating the HMM on each iteration of the Baum-Welch algorithm.

The variable \( \xi(i,j) \) is the probability of arriving in state \( i \) at time step \( t \), then transitioning to state \( j \) at time step \( t + 1 \), given all the observations in the sequence. In other words, it is the probability of generating the first \( t + 1 \) observations in the given sequence, ending in state \( i \) at time step \( t \), transitioning to state \( j \) from state \( i \), generating observation \( k \) at time step \( t + 1 \) from state \( j \), then transitioning out of state \( j \) and making the last \( T - t - 1 \) observations. Similar to the calculation of \( \gamma(i) \), the probability of this sequence of events occurring for a particular \( i-j \) state pair must be divided by the probability of the same sequence of events occurring for any \( i-j \) state pair. For a given \( i-j \) state pair, the probabilities of each of the various steps described above are, respectively, \( \alpha(i) \), \( a_{ij} \), \( b_{jk} \), and \( \beta(j) \). The equation specifying \( \xi(i,j) \) is therefore as follows, assuming the observation made at time step \( t + 1 \) is observation \( k \):

\[
\xi(i,j) = \frac{\alpha(i) * a_{ij} * b_{jk} * \beta(j)}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha(i) * a_{ij} * b_{jk} * \beta(j)} .
\]

Note that

\[
\sum_{i=0}^{T} \xi(i,j)
\]

is the expected number of times that the HMM transitions from state \( i \) to state \( j \) in the entire observation sequence.

With all the above functions, the features of the new HMM can now be defined. First, each start-state probability is updated to be the expected number of times that the HMM starts in that state:

\[
s_{i}^{\text{new}} = \gamma(i)
\]

for all \( i \).

Next, the probability of transitioning from state \( i \) to state \( j \) is updated to be the expected number of times that the HMM transitions from state \( i \) to state \( j \) divided by the expected number of times that the HMM transitions out of state \( i \) in total:

\[
a_{ij}^{\text{new}} = \frac{\sum_{i=0}^{T} \xi(i,j)}{\sum_{i=0}^{T} \gamma(i)} .
\]

for every \( i-j \) state pair.
Finally, the observation probabilities of the new HMM are defined as the expected number of times that each observation $j$ is made from a given state $i$, divided by the expected number of times that the HMM reaches state $i$ in total:

$$b_{ij}^{\text{new}} = \frac{\sum_{t=0}^{T} \gamma(i)}{\sum_{t=0}^{T} \gamma(j)}.$$  

(10)

Note that the values in the specification of the old HMM are used throughout. Only once all features of the new HMM are calculated is the new HMM used.

### 4.2 Proposed Experiment

A necessary final element to this analysis of botnet detection is the ability to test the similarity of two HMMs. Luckily, there exists an algorithm that can quantify the "distance" between two HMMs, reflecting the similarity of their various states, inter-state transitions, and observations. The thought process behind the proposed bot-detecting experiment is as follows:

- Bots often communicate with their botmaster at regular intervals, and bots on different machines often communicate at the same time. If it is possible to detect this coordinated communication, it should be possible to detect which machines host bots and are part of a botnet.
- Bot behavior differs from normal user browsing behavior, and may even generate noticeable differences in network traffic when a user is not browsing. Given two computers, either representing normal user browsing behavior or an idle internet connection, coordinated network events on both machines could indicate the presence of bots.
- A Hidden Markov Model can simulate the underlying states of a computer that generate the sequence of packets from a particular internet session. Some single data element can be extracted from network packets generated by that session to represent the sequence of observations. Using Baum-Welch, a maximized-probability HMM can then be constructed to reflect these observations.
- Given HMMs for two different computers, it is possible to measure the distance between these HMMs. Hopefully, computers running identical bots will show a much smaller distance than uninfected computers, because of their coordinated action and the similar frequency of some of their states.

### 5 Future Work

A Windows-compatible program that is well-suited to capturing network packets is Wireshark. A safe reimplementation of a simple HTML-requesting bot can be run while Wireshark is collecting packets. The bot's update interval – the frequency with which it polls the botmaster's website – can be set to some regular time, say 1 or 5 minutes. The capture would have to be several times longer than the bot's update interval, to ensure multiple bot-initiated requests are collected. If only a limited number of computers are available, a random string (generated when the bot is initialized) can be concatenated onto the end of the computer's hostname in the bot's request. The bot must then be stopped and started anew at the end and beginning of each capture. This will make it appear that each capture is done from a slightly different host, and will provide some variance in the bot's messages.

In the preliminary stages of testing, it would be ideal for the bot to begin running at approximately the same time and with the same update interval in each capture, with no background user-initiated browsing, and for each capture to record the same number of bot-initiated requests. This will test if HMMs can detect bot synchronization in the simplest case. Cases that might be more difficult for HMMs to detect would include sessions in which the bots have constant but different update intervals, or random update intervals, or when sessions have different capture time spans, or when bots are started at dramatically different times.

---

3 It may be necessary to begin running the bot only after Wireshark has begun capturing packets, to ensure the entire first bot-generated packet is captured.
in the capture, or when only one bot-initiated request is captured but it syncs up with a request made by a bot with multiple requests. Aside from variance in the bot's behaviors, it would also be worthwhile to capture user browsing behavior, especially when the user or multiple users check the same website at regular or irregular intervals.

The success of HMMs to detect the similarity between two bot-controlled computers and the dissimilarity of normal browsing behavior to bot-directed regular behavior will determine the true power of this approach.

Acknowledgements. This research was funded by the National Science Foundation as part of the 2008 Bio-Grid Research Experience for Undergraduates at the University of Connecticut, directed by Professor Chun-Hsi Huang. This research was conducted under the advisement of Professor Aggelos Kiayias, with significant contributions from David Walluck.

6 References