Automatically Generating Models for Botnet Detection

Peter Wurzinger[1], Leyla Bilge[2], Thorsten Holz[1,3],
Jan Goebel[3], Christopher Kruegel[4], and Engin Kirda[2]

[1] Secure Systems Lab, Vienna University of Technology
pw@seclab.tuwien.ac.at
[2] Institute Eurecom, Sophia Antipolis
{bilge,kirda}@eurecom.fr>
[3] University of Mannheim
{holz,goebel}@informatik.uni-mannheim.de
[4] University of California, Santa Barbara
chris@cs.ucsb.edu

Abstract. A botnet is a network of compromised hosts that is under the control of a single, malicious entity, often called the botmaster. We present a system that aims to detect bots, independent of any prior information about the command and control channels or propagation vectors, and without requiring multiple infections for correlation. Our system relies on detection models that target the characteristic fact that every bot receives commands from the botmaster to which it responds in a specific way. These detection models are generated automatically from network traffic traces recorded from actual bot instances. We have implemented the proposed approach and demonstrate that it can extract effective detection models for a variety of different bot families. These models are precise in describing the activity of bots and raise very few false positives.

1 Introduction

As the popularity of the Internet increases, so does the number of miscreants who abuse the net for their nefarious purposes. A popular tool of choice for criminals today are *bots*. A bot is a type of malware that is written with the intent of compromising and taking control of hosts on the Internet. It is typically installed on the victim's computer by either exploiting a software vulnerability in the web browser or the operating system, or by using social engineering techniques to trick the victim into installing the bot herself. Compared to other types of malware, the distinguishing characteristic of a bot is its ability to establish a command and control (C&C) channel that allows an attacker to remotely control or update a compromised machine [9]. A number of bot-infected machines that are combined under the control of a single, malicious entity (called the *botmaster*) are referred to as a *botnet*. Such botnets are often abused as platforms to launch denial of service attacks [22], to send spam mails [17,26], or to host scam pages [1].

To complement host-based analysis techniques (such as anti-virus (AV) software), it is desirable to have a network-based detection system available that can monitor network traffic for indications of bot-infected machines. So far, work to detect bots at the network-level has proceeded along two main lines: The first line of research uses *vertical correlation* techniques. These techniques focus on the detection of individual bots,

M. Backes and P. Ning (Eds.): ESORICS 2009, LNCS 5789, pp. 232–249, 2009.

typically by checking for traffic patterns or content that reveal command and control traffic or malicious, bot-related activities. These systems require prior knowledge about the command and control channels and the propagation vectors of the bots that they can detect. The second line of research to detect bots uses *horizontal correlation* approaches to analyze the network traffic for patterns that indicate that two or more hosts behave similarly. Such similar patterns are often the result of a command that is sent to several members of the same botnet, causing the bots to react in the same fashion (e.g., by starting to scan or to send spam). The drawback of these approaches is that they cannot detect individual bots. That is, it is necessary that at least two hosts in the monitored network(s) are members of the *same* botnet.

In this paper, we propose a detection approach to identify single, bot-infected machines without any prior knowledge about command and control mechanisms or the way in which a bot propagates. Our detection model leverages the characteristic behavior of a bot, which is that it (a) receives commands from the botmaster, and (b) carries out some actions in response to these commands. Similar to previous work, we assume that the command and response activity results in some kind of network communication that can be observed.

The basic idea of our system is that we can generate detection models by observing the behavior of bots that are captured in the wild. More precisely, by launching a bot in a controlled environment and recording its network activity (*traces*), we can observe the commands that this bot receives as well as the corresponding responses. To this end, we present techniques that allow us to identify points in a network trace that likely correlate with response activity. Then, we analyze the traffic that precedes this response to find the corresponding command. Based on the observations of commands and responses, we generate detection models that can be deployed to scan network traffic for similar activity, indicating the fact that a machine is infected by a bot. Our approach produces specific detection models that are tailored to bot families or groups of bots related by a common C&C infrastructure. Because the system is automated, it is easy to quickly generate new models for bots that implement novel commands and responses. This is independent of any prior knowledge of the protocol or the commands that the bot uses.

For our evaluation, we generated detection models for 18 different bot families, 16 controlled via IRC, one via HTTP (Kraken), and one via a peer-to-peer network (Storm Worm). Our results indicate that our system is able to produce precise detection models that reflect well the command and response activity of the bots. These models allow us to identify bot-infected hosts on a network with a low false positive rate.

The contributions of this paper are as follows:

- We present a model to capture the command and response activity of bots in network traffic.
- We propose an automated mechanism to generate bot detection models by observing the actual behavior of bot instances in a controlled environment, without making assumptions about the C&C mechanisms.
- We demonstrate the feasibility of our approach by generating detection models for various bot families (including those controlled via IRC and HTTP, as well as P2P). These models are effective in detecting bots with few false positives.

An extended version of this paper is available as a technical report [33].

2 System Overview

This section provides an overview of our approach to generate network-based detection models to identify bot-infected machines.

The input to our system is a collection of bot binaries. These binaries are collected in the wild, for example, via honeynet systems such as Nepenthes [2], or through Anubis [5], a malware collection and analysis platform. The output of our system is a number of models that can be used to detect instances of different bot families.

The basic idea of our system is to launch a bot in a controlled environment and let it connect to the Internet. Then, we attempt to identify the commands that this bot receives as well as its responses to these commands. Afterwards, these observations are translated into detection models that analyze network traffic for symptoms of bot-infected machines. The two main questions that arise are: (a) how are detection models specified, and (b), how can we generate these models based on observing bot activity?

2.1 Detection Models

The goal of a detection model is to specify network traffic activity that is indicative of the presence of a bot-infected machine.

Stateful models. In our system, a detection model has two states. The first state of the model specifies signs in the network traffic that indicate that a particular bot command is sent. For example, such a sign could be the occurrence of the string `.advscan`, which is a frequently-used command to instruct an IRC bot to start scanning. Once such a command is identified, the detection model is switched into the second state. This second state specifies the signs that represent a particular bot response. Such a sign could be the fact that the number of new connections opened by a host is above a certain threshold, which indicates that a scan is in progress. When a model is in the second state and the system identifies activity that matches the specified behavior, a bot infection is reported. If no activity is found that matches the specification of the second state for a certain time period, the model is switched back to the first state. Note that we maintain a different (logical) model instance for each host that is monitored. That is, when a command is found to be sent to host x, only the model for this host is switched to the second state. Therefore, there is no correlation between the activity of different hosts. For example, when a scan command is sent to host x, while immediately thereafter, host y initiates a scan, no alert is raised.

We make use of a stateful model that only labels a host as bot-infected if the system detects that a command is sent to the host **and** it witnesses a response within a certain period of time. This directly reflects the characteristic behavior of bots, which remotely receive commands from a botmaster and react accordingly. A stateful model has the advantage that we can use less restrictive specifications to capture both the command and the bot response, without risking an unacceptably high number of false positives.

In our current system, we use content-based specifications (comparable to intrusion detection signatures) to model commands, and network-based specifications (comparable to anomaly detection) to model responses. This is a natural approach, where content signatures capture commands and network models reflect the network activities due to responses (such as scanning, mass mailing, or binary downloads).

2.2 Model Generation

Given our notion of detection models, the question is how these models can be generated automatically. As mentioned previously, we do this based on the observation of bot activity. More precisely, for each bot binary, we first record a trace of its network activity over a certain period of time. Based on a trace, we have to identify those points where the bot receives a command and responds appropriately.

Finding responses. Our key insight for being able to identify previously unknown commands in a network trace is that we attack the problem from the opposite side. That is, instead of checking the traces for commands, we first look for the activity that indicates that a response has occurred. The reason for this approach is that a response launched by a bot is often more visible in the network trace than an incoming command. While a bot is in an idle state (i.e., it is not fulfilling requests of its botmaster), the network activity is typically limited to the traffic required to participate in the botnet (e.g., by exchanging IRC information or by polling web pages). However, when a command is issued, the bot has to act accordingly. This action almost always leads to additional network activity, for example, because the bot engages in scanning, downloads additional components, or sends mails. This activity stands out from the background noise and can be detected as an anomaly.

Once a bot response is identified, it is characterized by a *behavior profile*. More precisely, a behavior profile models various properties of the network traffic that are associated with a bot response. More details on recording bot traffic and locating responses are presented in Section 3.

Finding commands. By scanning the trace for network anomalies, we can identify those points in time at which a bot has demonstrated a response. As a result, the network traffic before this point must contain the command that has caused this response. Thus, before each point at which a significant change in traffic behavior is detected, we extract a *snippet*, a small section of the network trace.

Typically, different commands will lead to responses that are different. Therefore, in a next step, we cluster those traffic snippets that lead to similar responses, assuming that they contain the same command. Once clusters of related network snippets have been identified, we search them for sets of common (string) tokens. As our results demonstrate, these tokens frequently represent the bot commands and can be used for detection. Section 4 provides more details on the way in which traffic snippets are clustered and analyzed for common bot commands.

Putting it all together. Extracted tokens can be directly used to represent the bot command in the first state of the detection model. For the second state (i.e., to specify the response), we leverage the network behavior profiles that characterize bot response activity. Thus, in our current system, a bot detection model consists of a set of tokens that represent the bot command, followed by a network-level description of the expected response. These models can be readily deployed on the network and can identify an infected host once this host receives a known command and responds as expected.

Bot families. To provide sufficient quantity and diversity of command-response pairs for our system to generate meaningful signatures, it is desirable to combine samples from different botnets into bot families, as long as they use the same C&C mechanism.

The partitioning of samples into bot families can be performed either manually, based on malware names assigned by anti-virus scanners, or based on behavioral similarities. For example, previous work has introduced host-based analysis systems that can find similar malware instances based on the system calls that these malware programs invoke [3,6,28]. Moreover, the partitioning step does not need to be perfect. Our system can tolerate the case in which the pool of bot network traces is polluted.

For the following discussion, we assume that the set of bot samples has already been divided into consistent groups. Of course, the system is neither provided with any information about the way in which commands are exchanged, nor how and when responses are launched.

3 Analyzing Bot Activity

As a first step to creating bot detection models, our system requires captures of the network traffic that the bot-infected machines create. To this end, we run each bot binary in a controlled environment with Internet access for a period of several days. The goal is to let the bot connect to its C&C mechanism and keep it running long enough to observe a representative collection of the different bot commands and the activities they trigger. The observed traffic should contain the most frequently used commands, since these are the most helpful detection targets. On the other hand, the absence of rarely used commands is acceptable, since detection models targeting these commands would also rarely trigger when deployed. A more detailed description of our bot trace collection environment can be found in the technical report [33].

3.1 Locating Bot Responses

Once a network trace is collected, the next step is to locate the points within this trace where the bot executes responses to previously received commands. We do this by checking for sudden changes in the network traffic (e.g., a surge in the number of packets, or the fact that many different hosts are contacted). The assumption is that such changes indicate bot activity that is launched when a command is received. Of course, this implies that we can only detect bot responses (and hence, commands) that lead to a change in network behavior. However, most current bot responses, such as sending spam mails, executing denial of service attacks, uploading stolen information, or downloading additional components, fall into this category.

Of course, it is possible that there are changes in the traffic that are not caused by commands. For example, a scan might end when the list of victims is exhausted. Our system will also consider the end of the scan as a potential response, and mark the location appropriately. Fortunately, this is of little concern, because it is likely that the subsequent analysis will fail to find an appropriate command for this (inexistent) response. Sometimes, however, interesting detection models can be generated in such cases. For example, once a bot has finished scanning, it often sends a status notification to the botmaster, which our system can extract as a content signature.

Locating bot responses in a network trace can be treated as a change point detection (CPD) problem. CPD algorithms operate on time series, that is, on chronologically

Table 1. Network features to characterize bot behavior

Number of packets	Number of non-ASCII bytes in payload
Cumulative size of packets (in bytes)	Number of UDP packets
Number of different IPs contacted	Number of HTTP packets (destination port 80)
Number of different ports contacted	Number of SMTP packets (destination port 25)

ordered sequences of data values. Their goal is to find those points in time at which the data values change abruptly. Change point detection has been used previously to recognize spreading worms [34] and denial of service attacks [32]. However, we are not aware of any prior work that used it in the context of botnet detection.

Before we can apply a CPD algorithm, we first have to convert a traffic trace into a time series. To this end, the network traffic is partitioned into consecutive time intervals of equal length (our choice of a concrete interval length will be discussed later). Then, we compute a numeric description in the form of a vector that represents the network traffic for each interval. For this, we extract a number of low-level features from the network traffic. Each feature captures a different aspect of the network traffic and translates into one element of the vector. Currently, we consider eight network traffic features:

Using the features shown in Table 1, we can characterize the bot's behavior during a given time interval. The characterization of bot activity is designed in a generic fashion, taking into account general features such as the number of packets, number of different machines contacted, or the number of (binary) bytes in network streams. In addition, we include two features that are derived from our domain knowledge of common bot responses: the numbers of SMTP and HTTP packets. The reason is that sending spam mails typically results in a surge of SMTP packets. The HTTP feature was initially considered as helpful to detect cases in which a bot downloads additional components via this channel. However, also currently unknown bot activity could be captured by our features, and it is certainly easy to add additional ones.

For every time interval, we calculate a vector that stores the absolute value for each feature. For example, when 50 packets are seen during a certain time interval, the corresponding element of the vector (number of packets) is set to 50. We call this vector a *traffic profile* of the bot for this time interval. To be able to compare behaviors obtained from different traces, this vector is normalized with regard to the maximum that was observed for the corresponding feature. This yields a value between 0 and 1 for all vector elements.

Change point detection. Once a network trace is converted into a sequence of traffic profiles, we apply a CPD algorithm to locate points that indicate interesting changes in the traffic. For this, we use CUSUM (cumulative sum), a well-known, robust algorithm that is known to deliver good results for many domains [4]. In principle, CUSUM is an online algorithm that detects changes as soon as they occur. Since we have the complete network trace (time series) available, we can leverage this fact and transform CUSUM into an off-line algorithm. This allows CUSUM to "look into the future" when a decision needs to be made, and thus, yields more precise results.

The algorithm to identify change points works as follows: First, we iterate over every time interval t, from the beginning to the end of the time series. For each interval t, we calculate the average traffic profile P_t^- for the previous $\epsilon = 5$ time intervals and the traffic profile P_t^+ for the subsequent ϵ intervals. Then, we compute the distance $d(t)$ between P_t^- and P_t^+. The distance between two traffic profiles is defined as the Euclidean distance between the corresponding vectors. More precisely:

$$P_t^- = \sum_{i=1}^{\epsilon} \frac{P_{t-i}}{\epsilon} \qquad P_t^+ = \sum_{i=1}^{\epsilon} \frac{P_{t+i}}{\epsilon} \qquad d(t) = \sqrt{\sum_{1}^{dim} \left|P_t^- - P_t^+\right|^2} \tag{1}$$

The ordered sequence of values $d(t)$ forms the input to the CUSUM algorithm. Intuitively, a change point is a time interval t for which $d(t)$ is sufficiently large and a local maximum.

The CUSUM algorithm requires two parameters. One is an upper bound ($local\_max$) for the normal, expected deviation of the present (and future) traffic from the past. For each time interval t, CUSUM adds $d(t) - local\_max$ to a cumulative sum S. The second parameter determines the upper bound ($cusum\_max$) that S may reach before a change point is reported. To determine a suitable value for $local\_max$, we require that each individual traffic feature may deviate by at most $allowed\_avg\_dev = 0.04$. Based on this, we can calculate the corresponding value $local\_max = \sqrt{dim \times allowed\_avg\_dev^2}$. For $cusum\_max$, we use a value of 0.25. We empirically determined the values for $allowed\_avg\_dev$ and $cusum\_max$. However, note that these values are robust and yield good results for a large variety of traffic produced by hundreds of different malware instances that belong to different bot types (IRC, HTTP, and P2P bots).

It is possible that the cumulative sum S exceeds $cusum\_max$ for a number of consecutive time intervals. To locate the actual change point in this case, we take that interval for which $d(t)$ is maximal (since it is the time interval with the greatest discrepancy between past and future traffic composition). The precision with which a change point can be located also depends on the length of the time intervals. Shorter intervals increase the precision. Unfortunately, they also increase the probability that small traffic variations (e.g., bursts) are misinterpreted as a change point. This could introduce unwanted noise into the subsequent model generation process. To find a suitable length for the time intervals, we experimented with a variety of values between 20 and 100 seconds. An interval of 50 seconds delivered the best results in our tests.

3.2 Extracting Model Generation Data

We assume that each change point indicates the time when a bot has received a command and initiated the corresponding response. Based on this assumption, we leverage change points to extract two pieces of information that are needed for the subsequent model generation step.

First, we extract a snippet of the traffic that is likely to contain the command that is responsible for the observed change. Clearly, the snippet must contain the traffic within the time interval where the change point is located. Moreover, we take the first 10 seconds of the following interval. The reason is that when a change point occurs close to the boundary between two intervals, the CPD algorithm might select the wrong one. To compensate for this imprecision, the start of the subsequent traffic interval is included. Finally, we include the last 30 seconds of the previous interval to cover typical command response delays. As a result, each snippet contains 90 seconds of traffic.

The second piece of information required for creating a detection model is a description of the response behavior. To this end, we extract a behavior profile, which captures the network-level activities of the bot once a command is received. This profile consists of the average of the traffic profile vectors over the complete period where the bot carries out its response. This period is considered to be the time from the start of the current response to the next change in behavior. That is, once the network traffic changes again, we assume that the bot has finished its task or received another command.

4 Generating Detection Models

Given a set of network traffic snippets, together with their associated response behavior profiles, we automatically generate suitable detection models. Recall that detection models should embody the correlation of two events: The appearance of a command in the network traffic, and the appearance of a subsequent response. The patterns that each of the two events have to match are represented separately in our model.

At this point, the set of snippets contains a mix of network traffic that consists of different commands and some contents that are specific to the C&C protocol. For subsequent processing performed by the token extraction algorithm, we require a two-phase clustering: First, we arrange snippets such that those are put together in a cluster that likely contain the same command. Afterwards, we group the contents of the snippets in each cluster such that elements in a group share commonalities that can be leveraged by the token extraction algorithm.

First, to cluster similar snippets, we make the following observation: The network traffic of a bot responding to a certain command will look similar to the traffic generated by this bot executing the same command at some later time. On the other hand, the same bot executing a different command will generate traffic that looks different. That is, there is a correspondence between the command that is sent and the response that is invoked. This observation can be leveraged by clustering the snippets according to the behaviors that we believe to be a response. That is, the goal is to find *behavior clusters*, where each such cluster represents a certain bot activity, such as a scanning period or any other kind of distinguishable network activity. Once such clusters have been found, we can expect that most snippets that are part of the same cluster contain common parts that are either directly responsible for triggering the bot reaction (the command itself), or at least always appear in order for a bot to react that way.

To identify behavior clusters, we perform hierarchical clustering [10] based on the normalized response behavior profiles. After the clustering step, each cluster holds a set of snippets that likely contain a command that has led to the same response. These snippets are used to extract the model of the bot command (as described in Section 4.1). The response behavior profiles associated with the snippets are then used to model the response activity (as discussed in Section 4.2).

4.1 Command Model Generation

The objective of the command model generation step is to identify common elements in a set of network snippets that belong to a particular behavior cluster. In particular, we are interested in finding character strings that appear frequently in the traffic snippets, since there is a chance that they encode bot commands.

To extract likely bot commands from network traces, we use a signature generation technique that produces *token sequences*. A token sequence consists of an ordered set of tokens. That is, the tokens have to appear in a certain order, but there can be arbitrary characters between each token. Token sequences can be easily encoded as regular expressions (which can serve directly as input to a network intrusion detection system).

To find common tokens, we use the longest common subsequence algorithm (based on suffix arrays). Since the algorithm outputs a token sequence only if it is present in all network traces, we cannot apply the algorithm directly. The reason is that different commands may lead to similar responses which may be clustered together. Furthermore, an incorrectly detected change point can cause an unrelated snippet to become part of a cluster. Therefore, we require a second clustering refinement step that groups similar network packet payloads within each behavior cluster. For the second clustering step, we employ a standard complete-link, hierarchical clustering algorithm to find payloads that are similar.

The longest common subsequence algorithm is applied to each set of similar payloads, generating one token sequence per set. Recall that the second clustering step is performed individually for each behavior cluster. Thus, it is possible (and common) that multiple token sequences are associated with a single behavior cluster. Each of these token sequences represents a potential command that leads to network activity that the corresponding response behavior profile captures.

Precision optimizations. Some of the generated token sequences may be overly generic, i.e., they are likely to match on benign traffic frequently. We want to identify and remove these token sequences to improve the precision of our detection models. This can be done in an automated way by matching all generated token sequences against known benign traffic: every match is clearly undesirable and suggests to discard the token sequence. We recorded the traffic at the Secure Systems Lab, a well administrated network, for a duration of one day. It is save to assume that all traffic is benign. Furthermore, we remove all token sequences whose longest token is shorter than five bytes. This is done because token sequences consisting only of very short tokens will trigger frequently just by chance, e.g., when large amounts of binary data are transmitted.

4.2 Response Model Generation

The second part of our detection model consists of a network-based description of the bot response. This description should capture the kind of network activity that is expected to be shown by a bot after the command has been received.

The input to this step is a behavior cluster. Recall that a behavior cluster is created by grouping similar response behavior profiles and their associated snippets. We generate the bot response model for a behavior cluster by computing the element-wise average of the (vectors of the) individual behavior profiles. The result is another behavior profile vector that captures the aggregate of the behaviors combined in the respective behavior cluster. As such, this behavior profile is suitable to model the expected bot response behavior associated with the bot commands that are described by the content-based models extracted from the snippets.

Precision optimizations. In some cases, the behavior profile of a bot response can be exceeded by sending only a few HTTP packets or by contacting two other hosts. Clearly, such traffic is easily produced by regular users (e.g., surfing the web or using an instant messaging client). Thus, we introduce minimal bounds for certain network features. In particular, we define a threshold of 1,000 for the number of UDP packets that are sent within one time interval (50 seconds), 100 for HTTP packets, 10 for SMTP packets, and 20 for the number of different IPs. When a response profile exceeds *none* of these thresholds, the corresponding behavior cluster (and its token sequences) are not used to generate a detection model. This technique removes a small number of weak profiles that could potentially result in a large number of false positives.

4.3 Mapping Models into Bro Signatures

Bro is a network intrusion detection system designed to monitor network activity for suspicious or irregular events [24]. One of its key features is the integrated policy and signature scripting language, which enables custom rules for intrusion detection. Due to its flexibility, Bro is an appropriate platform to implement our detection models.

To map a detection model into a Bro specification, we have to encode the model's set of token sequences as well as its behavior profile. For each token sequence, one Bro signature is generated. The signature consists of the concatenation of the individual tokens of a token sequence, using the '`.*`' regular expression operator. Also, each signature is restricted to match only on inbound or outbound traffic, depending on the bot traffic it had been generated from.

When a token sequence matches, the corresponding detection model is advanced to the second state. At this point, Bro starts to record the traffic of the host that triggered a signature. This is done for a duration of 50 seconds. Then, the system creates a profile from the recorded traffic, using the following four features: number of UDP packets, number of HTTP packets, number of SMTP packets, and number of unique IP addresses. When the observed traffic exceeds, for at least one of these four features, the corresponding value stored in the response profile, we consider this a match. In that case, the host is considered to be bot-infected, and an alert is raised.

5 Evaluation

The purpose of the evaluation is to demonstrate that our system generates detection models that are capable of detecting bot-infected hosts with a low false positive rate.

In a first step, we collected a set of 416 different (based on MD5 hash) bot samples. We obtained these malware programs through Anubis, a public malware analysis service [5]. Thus, the samples originate from a wide range of sources and include bots manually submitted by users, binaries collected with the help of honeypots and spam traps, as well as contributions from malware analysis organizations (such as

Table 2. Number of detection models (DM) and token sequences (TS) for each bot family

Bot family	#DM	#TS	Bot family	#DM	#TS	Bot family	#DM	#TS
IRC1	4	57	IRC7	8	53	IRC13	2	8
IRC2	9	50	IRC8	3	72	IRC14	5	38
IRC3	2	11	IRC9	3	17	IRC15	3	24
IRC4	4	94	IRC10	2	7	IRC16	1	1
IRC5	1	8	IRC11	11	35	HTTP	2	5
IRC6	1	20	IRC12	7	21	STORM	2	110
						TOTAL	70	631

`ShadowServer.org`). The collection period was more than 8 months. All bot samples were executed in our traffic capturing environment, each producing a traffic trace with a length of five days.

In the next step, the bot traffic traces were divided into families of bots. This was a manual process, based on the content of the traces. However, this step could be automated in the future [3,6]. The classification process yielded a total of 16 different IRC bot families (with 356 traffic traces) and one HTTP bot family consisting of samples of Kraken (also known as Bobax, with 60 traffic traces). In addition, we obtained 30 network captures for the Storm Worm (also known as Peacomm and Zhelatin), which is the most well-known example of a botnet that uses a peer-to-peer protocol for its C&C communication [13]. The Storm Worm captures were separately generated at the University of Mannheim. Thus, in total, there were 446 network traces available as input for our detection model generation process.

Using these 446 network traces, our system produced a total of 70 detection models. A more precise breakdown of this number for the different bot families is shown in Table 2. The table also shows the numbers of token sequences produced. Recall from Section 4.1 that there can be multiple token sequences associated with a single detection model, but it is sufficient that a single one triggers to switch the model into the second state (where it checks for suspicious response activity). As can be seen, our system succeeded in producing at least one detection model for each bot family. This is particularly interesting when considering that Storm uses encrypted commands. When examining the Storm signatures, we observed that our system correctly identified that the byte string "`.mpg;size=`" is characteristic for this bot type. That is, even though we cannot precisely identify a command in the network trace, our algorithm is able to extract specific artifacts of the bot communication. Also, it should be noted that this automatically-generated token sequence is very close to the human-specified signature in Snort [29], a popular network intrusion detection system.

To understand the quality of our automatically-generated detection models, we compared them to the human-developed bot and C&C signatures used by Snort. This serves as an initial, qualitative assessment to determine whether the signatures are "reasonable" and match traffic that a human analyst would associate with bot activity. In many cases, we found that the signatures were very similar to the human-created references, which confirms that our approach is capable of delivering intuitively correct results. This was true for signatures for all three bot classes (IRC, HTTP, and P2P) that we examined. In other cases, we found that our signatures were overly specific, and contained

```
signature irc1-000-2 {
  dst-ip == local_nets
  payload /.* PRIVMSG #.* :\.asc .*5 0 .*/
}

#DIFFERENT IPS > 20
```

Fig. 1. Automatically-generated Bro signature and corresponding behavior profile for an IRC bot

artifacts of a particular bot that was analyzed (e.g., IRC channel names, IP addresses, time stamps). However, it is typically not problematic to include such specific signatures. While they likely do not detect any bots, they typically do not contribute to false alarms either.

An example of an automatically-generated detection model for a family of IRC bots is shown in Figure 1. The token sequence consists of three tokens that need to be identified in an inbound IP packet. The first token (PRIVMSG #) contains a part of the IRC protocol header for transmitting a message. This token restricts the signature to match only on IRC traffic. The second token (:.asc) contains the command that instructs the receiving bot to begin scanning. The third token (5 0) contains parameters for the scan command. At first, it might seem that this token makes the signature overly restrictive. However, very often, the same set of parameters is used for a command. Thus, this is not a significant restriction. In comparison, a human-created Snort signature matches on "PRIVMSG .*:.*asc". The network behavior that needs to be matched in the second detection phase (once the token sequence has been identified in the traffic) requires that a host contacts more than 20 distinct IPs within a time period of 50 seconds. This reflects the scan that a bot initiates when receiving the .asc command. Only if this second condition is fulfilled as well, the host is reported as bot-infected.

For additional examples of HTTP and P2P detection models, as well as encrypted C&C channels, the reader is referred to the technical report [33].

5.1 Detection Capability

To obtain a quantitative measure for the capability of our detection models to identify bot-related traffic, we decided to split our set of 446 network traces into training sets and test sets. Each training set contained 25% of one bot family's traces, while the corresponding test set contained the remaining ones. We used the training sets to generate a new set of detection models. Then, this new set of models was loaded into Bro, and we analyzed the traffic traces in the test sets. In total, this procedure was performed four times per family (four-fold cross validation).

Our system reported a bot infection for 88% of the analyzed traces. The remaining 12% of traces did not trigger even a token sequence match. For all traces that did lead to at least one token sequence match, the behavior profile matching phase triggered as well, thus, correctly confirming the bot infection.

To further put the detection results into context, we decided to perform a comparison between our system and BotHunter [15]. BotHunter is the current state-of-the-art tool for detecting individual bot infections. The system uses a number of phases that model different aspects of the bot life cycle (such as spreading, C&C, and malicious activity).

To detect bot commands, BotHunter relies on manually-developed signatures (mainly the database of Snort and some custom signatures). To determine the performance of BotHunter, we ran its latest version (v1.0.2, with default settings) on all 446 bot traffic traces. BotHunter identified signs of bot infections for 69% of the traces. The automatically generated signatures produced by our system thus outperform BotHunter by nearly 20%.

5.2 Real-World Deployment

To analyze the amount of false positives that our detection models generate, we extensively evaluated our system in two real-world network environments. More precisely, we deployed one Bro sensor with our detection models in front of the residential homes of RWTH Aachen University and one sensor at a Greek university network. In Aachen, our system monitored a densely-populated /21 network (2K IPs) for a duration of 55 days. In Greece, we monitored a medium-populated /20 network (4K IPs) for 102 days. On average, we observed about 40 million packets per hour in Aachen, while the number in Greece was about 17 million packets. Thus, our experimental evaluation comprises the analysis of traffic in the order of 94 billion network packets over a period of over three months at two different sites in Europe.

The results of our evaluation are summarized in Table 3. Our deployment in Aachen yielded no alerts at all over a duration of two months. There were 130 token sequence matches, which were all correctly invalidated by the behavior profile matching phase. This demonstrates the importance of the second phase of our detection models: Random token sequence matches do not lead to an alert, because without the expected bot response, the behavior profile will not be matched.

In the Greek network, our system raised only few alerts, and over a period of over three months, reported a total of 11 hosts (IPs) as bot-infected. These 11 hosts were responsible for 60 alerts. To verify whether these alerts are false positives or indications of true bot infections, we performed manual analysis of the traffic that caused the alarms. In most cases, this led us to the conclusion that an alarm was a false positive. This is also supported by the fact that both networks are well-maintained and bot infections are very rare. However, a definite decision is difficult to make, since we did not have access to the actual hosts.

Typically, all machines that are reported as bot infected must be manually inspected. Thus, it is important that the system does not overload the administrator with incorrect warnings. Considering the average number of alerts per day that our system reports as well as the number of reported IP addresses (shown in Table 3), we believe that this goal is clearly met.

Table 3. Results from real-world deployments

	IP space	Packets/hour	Days	IPs flagged	Total alerts	Alerts/day
Aachen	2,048	40M	55	0	0	0
Greece	4,096	17M	102	11	60	0.59
BotHunter	4,096	17M	6	60	5,849	974.34
BotHunter w/o Blacklist	4,096	17M	6	5	60	10.00

Table 4. Comparison of the detection performance of our detection models vs. BotHunter

	Our detection models	BotHunter
Detection (true positive) rate on bot traces	88%	69%
Incorrectly detected IPs in real-world traffic (false positives)	11	60

Again, in order to compare our results with the current state-of-the-art BotHunter, we deployed a BotHunter sensor in the Greek network (we did not obtain permission to install such a sensor in Aachen). Unfortunately, due to performance limitations, we could run either BotHunter or our system on the machine that was provided to us, but not both systems at the same time. Thus, we deployed BotHunter for a period of only six days. Nevertheless, we feel that this period is sufficiently long to draw meaningful conclusions.

The comparison with BotHunter is instructive. We can see that an off-the-shelf BotHunter installation reports almost one thousand alerts per day. Within a period of only six days, 60 different IP addresses are reported as bot infected, each of which would require manual inspection. Given this very high number of false alerts, we investigated the reasons and even attempted to tweak BotHunter to improve its performance. On closer inspection of the alerts, we observed that a significant amount of them are due to two components (phases). These rely on blacklists of known DNS names and IP addresses that are related to malware domains and C&C servers. In an attempt to reduce the amount of BotHunter's false positives, we disabled these two components. An accordingly modified Bothunter setup produced only 10 alerts per day, reporting a total of 5 IP addresses as bot infected during the six day period. While, in contrast to the off-the-shelf setup, the amount of alerts is now manageable by a human administrator, BotHunter still does not reach the low number of false alerts our system generates.

Additionally, disabling the two components that are responsible for the vast majority of false alerts has a significant negative impact on BotHunter's detection capabilities. When rerunning the experiments on the bot traces using the modified version of BotHunter, the number of bots that BotHunter detects drops to only 39%.

Finally, a large fraction (89%) of the alerts raised by our system in the real-world deployments were triggered by only three different detection models. The situation is different for BotHunter: We observed 155 different matching BotHunter C&C signatures during the evaluation in the Greek network. This large diversity of matching signatures makes it difficult to disable a few BotHunter models that are responsible for the bulk of false positives.

We present a summary of the results of our evaluation in Table 4. Our automatically generated detection models clearly outperform the state-of-the-art solution for single bot detection, BotHunter, which relies on signatures hand-crafted by human experts.

6 Related Work

Malware, and botnets in particular, pose a significant threat to the security of the Internet. As a result, there has been a strong interest in the research community to develop adequate defense solutions. This paper touches on a number of related research areas.

Network intrusion detection. The purpose of network intrusion detection systems (IDS) is to monitor the network for the occurrence of attacks. Clearly, this is very similar to the purpose of our detection models that analyze network traffic for the presence of signs that indicate bot-infections. In fact, we directly encode our detection models in the signature language of Bro [24], a well-known, network-based IDS.

Of course, both the ideas of content-based analysis and modeling network-level properties to detect anomalies are not new. Content-based analysis has been used by signature-based IDSs (such as Snort [29] or Bro) for years. Also, network-level properties (such as the number of flows that were transferred) have been used extensively to model normal network traffic and to detect deviations that indicate attacks [21]. Our proposed work complements existing network-based IDSs by automatically generating the inputs needed by these systems to detect machines that are infected by bots.

Signature generation. As part of our detection model generation, we extract token signatures from network traffic. Research on such automated signature generation started with the work on Early Bird [30] and Autograph [19], and has later been extended with Polygraph [23] and Hamsa [20]. Of course, extracting command tokens is only a small part of the entire model generation process. In fact, we first have to record bot activity, identify likely bot responses, extract the corresponding traffic snippet, and cluster them based on behavioral similarities. Only then can we extract common tokens, using an improved version of previous algorithms.

Botnet analysis and defense. In addition to general research on malware detection, there is work that specifically focuses on the analysis [8,11,17,25] and detection [7,12,14,15,16,18,27] of botnets.

A number of botnet detection systems perform horizontal correlation. That is, these systems attempt to find similarities between the network-level behavior of hosts. The assumption is that similar traffic patterns indicate that the corresponding hosts are members of the same botnet, receiving the same commands and reacting in lockstep. While initial detection proposals [16,18] relied on some protocol-specific knowledge about the command and control channel, subsequent techniques [14,27] remove this shortcoming. The main limitation of systems that perform horizontal correlation is that they need to observe multiple bots of the same botnet to spot behavioral similarities (with small exceptions [16]). This is significant because botnets decrease in size [8], it becomes more difficult to protect small networks, and a botmaster can deliberately place infected machines within the same network range into different botnets.

A second line of research explored vertical correlation, a concept that describes techniques to detect individual bot-infected machines based on suspicious communication characteristics [7,12]. The most advanced system is BotHunter [15], which correlates the output of three IDS sensors – Snort [29], a payload anomaly detector, and a scan detection engine. A closer analysis of the results reveals that the detection capability of BotHunter strongly relies on the human-created Snort rules. Our system, on the contrary, generates detection models completely automatically. Moreover, the stages that are used by BotHunter to characterize the life cycle of a bot focus on scanning and remote exploiting. Our system, on the contrary, does not rely on a specific bot propagation strategy and does not require previous knowledge about command and control channels.

Independently and concurrently to our work, a recent paper [17] has presented the idea of running bots in a controlled environment (called Botlab). The proposed system is similar to ours in that bots are executed and monitored. The difference is that Botlab is exclusively focused on spam botnets and uses the monitored activity (in addition to other inputs) to produce information about spam mails (such as malicious URLs in the mail body). However, the approach does not provide any information about bot commands or responses, and it is not designed to detect bot infected machines.

7 Limitations

Although our current system is able to effectively detect real-world botnets, we note that it has several limitations, which we discuss in this section.

To evade detection, a botmaster may instruct his bots to wait for a certain amount of time before reacting to the command (i.e., he might launch a threshold attack [31]). As a result, our analysis could miss the connection between a command and the appropriate response, both when generating detection models or once the models are deployed. Many other comparable systems rely on a time window of some sort, and thus, are *vulnerable to this same attack* [14,15,16,27]. A possible way of handling this evasion attempt is to randomize the time window, making it harder for the adversary to select an appropriate delay. Also, long time delays reduce the usefulness of botnets and increase the difficulty for the attacker [16,31].

Another limitation of our current implementation is that it uses content-based analysis to detect command tokens. Thus, the system has problems with encrypted command channels. This is a limitation that our approach shares with *all previous techniques* that aim to detect single bots [7,12,15]. To avoid this problem, the most promising approach is to use network-level properties to recognize commands. Interestingly, even in the current version, our system can sometimes identify artifacts that are present in encrypted traffic. The best example is the Storm Worm, for which our system extracts a "command" token that is characteristic for this bot. Also, our system is resistant to simple obfuscation schemes in which a human-readable command is mapped to some unintelligible string. In fact, we have generated token sequences for IRC bot families that match obfuscated commands (as demonstrated in the technical report [33]). This is different from previous approaches, such as BotHunter [15], that deploy manually-developed signatures and thus, can be thwarted by bots that use non-standard commands.

8 Conclusions

This paper presents a system that identifies bot-infected machines by monitoring network traffic. It targets the unique characteristic of bots, the fact that they receive commands from the botmaster and respond appropriately. Our system observes the behavior of bots executed in a controlled environment, and automatically derives signatures for the commands that a bot can receive, as well as network-level specifications for the responses that these commands trigger. Our approach relies neither on the propagation vector, nor on any prior knowledge about the communication channel used by the bot. As a result, we can generate models for IRC bots, HTTP bots, and even P2P bots such as Storm. We have applied our system to a number of real-world bots, demonstrating that we can automatically extract accurate detection models. Our evaluation shows that our system outperforms BotHunter, which heavily relies on hand-tuned signatures.

Acknowledgments. This work has been supported by the Austrian Science Foundation (FWF grant P18764), MECANOS, Secure Business Austria (SBA), the Pathfinder project funded by FIT-IT, and the WOMBAT and FORWARD projects funded by the European Commission.

References

1. Anderson, D., Fleizach, C., Savage, S., Voelker, G.: Spamscatter: Characterizing Internet Scam Hosting Infrastructure. In: Usenix Security Symposium (2007)
2. Baecher, P., Koetter, M., Holz, T., Dornseif, M., Freiling, F.C.: The nepenthes platform: An efficient approach to collect malware. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 165–184. Springer, Heidelberg (2006)
3. Bailey, M., Oberheide, J., Andersen, J., Mao, Z.M., Jahanian, F., Nazario, J.: Automated classification and analysis of internet malware. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 178–197. Springer, Heidelberg (2007)
4. Basseville, M., Nikiforov, I.V.: Detection of Abrupt Changes - Theory and Application. Prentice-Hall, Englewood Cliffs (1993)
5. Bayer, U.: Anubis: Analyzing Unknown Binaries, `http://analysis.iseclab.org/`
6. Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, Behavior-Based Malware Clustering. In: Network and Distributed System Security Symposium, NDSS (2009)
7. Binkley, J., Singh, S.: An Algorithm for Anomaly-based Botnet Detection. In: Usenix Steps to Reducing Unwanted Traffic on the Internet Workshop, SRUTI (2006)
8. Cooke, E., Jahanian, F., McPherson, D.: The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In: Usenix Steps to Reducing Unwanted Traffic on the Internet Workshop, SRUTI (2005)
9. Dagon, D., Gu, G., Lee, C., Lee, W.: A Taxonomy of Botnet Structures. In: Annual Computer Security Applications Conference, ACSAC (2007)
10. de Hoon, M., Imoto, S., Nolan, J., Miyano, S.: Open Source Clustering Software. Bioinformatics 20(9) (2004)
11. Freiling, F.C., Holz, T., Wicherski, G.: Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In: de Capitani di Vimercati, S., Syverson, P.F., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 319–335. Springer, Heidelberg (2005)
12. Goebel, J., Holz, T.: Rishi: Identify Bot Contaminated Hosts by IRC Nickname Evaluation. In: Usenix Workshop on Hot Topics in Understanding Botnets, HotBots (2007)
13. Grizzard, J.B., Sharma, V., Nunnery, C., Kang, B.B.H., Dagon, D.: Peer-to-Peer Botnets: Overview and Case Study. In: Usenix Workshop on Hot Topics in Understanding Botnets, HotBots (2007)
14. Gu, G., Perdisci, R., Zhang, J., Lee, W.: BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In: Usenix Security Symposium (2008)
15. Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In: Usenix Security Symposium (2007)

16. Gu, G., Zhang, J., Lee, W.: BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In: Network and Distributed System Security Symposium, NDSS (2008)
17. John, J., Moshchuk, A., Gribble, S., Krishnamurthy, A.: Studying Spamming Botnets Using Botlab. In: Usenix Symposium on Networked Systems Design and Implementation, NSDI (2009)
18. Karasaridis, A., Rexroad, B., Hoeflin, D.: Wide-scale Botnet Detection and Characterization. In: Usenix Workshop on Hot Topics in Understanding Botnets, HotBots (2007)
19. Kim, H.A., Karp, B.: Autograph: Toward Automated, Distributed Worm Signature Detection. In: Usenix Security Symposium (2004)
20. Li, Z., Sanghi, M., Chen, Y., Kao, M.Y., Chavez, B.: Hamsa: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience. In: IEEE Symposium on Security and Privacy (2006)
21. Mahoney, M., Chan, P.: Learning Nonstationary Models of Normal Network Traffic for Detecting Novel Attacks. In: Conference on Knowledge Discovery and Data Mining, KDD (2002)
22. Moore, D., Voelker, G., Savage, S.: Inferring Internet Denial of Service Activity. In: Usenix Security Symposium (2001)
23. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically Generating Signatures for Polymorphic Worms. In: IEEE Symposium on Security and Privacy (2005)
24. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks 31 (1999)
25. Rajab, M.A., Zarfoss, J., Monrose, F., Terzis, A.: A Multifaceted Approach to Understanding the Botnet Phenomenon. In: Internet Measurement Conference, IMC (2006)
26. Ramachandran, A., Feamster, N.: Understanding the Network-Level Behavior of Spammers. In: ACM SIGCOMM Conference (2006)
27. Yen, T.-F., Reiter, M.K.: Traffic aggregation for malware detection. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 207–227. Springer, Heidelberg (2008)
28. Rieck, K., Holz, T., Willems, C., Düssel, P., Laskov, P.: Learning and Classification of Malware Behavior. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 108–125. Springer, Heidelberg (2008)
29. Roesch, M.: Snort - Lightweight Intrusion Detection for Networks. In: Systems Administration Conference, LISA (1999)
30. Singh, S., Estan, C., Varghese, G., Savage, S.: Automated worm fingerprinting. In: Symposium on Operating System Design and Implementation, OSDI (2004)
31. Stinson, E., Mitchell, J.: Towards Systematic Evaluation of the Evadability of Bot/Botnet Detection Methods. In: Usenix Workshop on Offensive Technologies, WOOT (2008)
32. Wang, H., Zhang, D., Shin, K.G.: Change-Point Monitoring for Detection of DoS Attacks. IEEE Transactions on Dependable and Secure Computing 1(4) (December 2004)
33. Wurzinger, P., Bilge, L., Holz, T., Goebel, J., Kruegel, C., Kirda, E.: Automatically Generating Models for Botnet Detection (TR-iSeclab-0609-001) (2009), http://www.iseclab.org/papers/tr_botdetection.pdf
34. Yan, G., Xiao, Z., Eidenbenz, S.: Catching instant messaging worms with change-point detection techniques. In: Usenix Workshop on Large-Scale Exploits and Emergent Threats, LEET (2008)