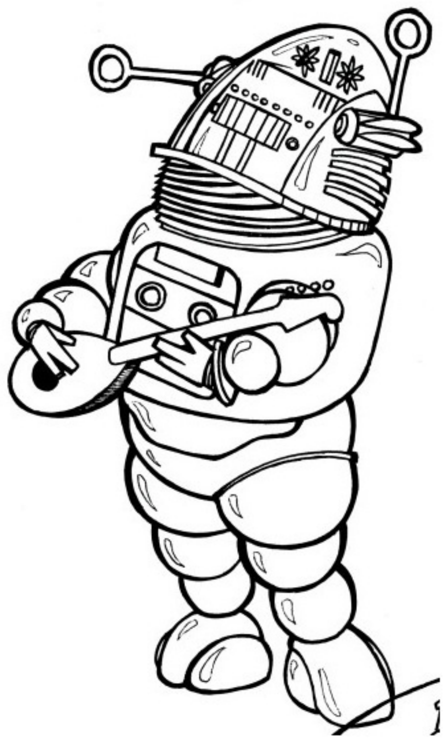
BARD: Botnet Atom Realtime Detector

December 9, 2010



Abstract:

A botnet consists of a collection of infected computers remotely controlled by a Botmaster. Communications between individual bots and the Botmaster are established by command and control (C&C) channels. The goal of BARD is to build an active realtime distributed firewall whose primary purpose is to detect botnets under C&C infrastructure from infiltrating our network. Our intrusion detection system analyzes network flow information for patterns common to botnet behaviors. The detection of compromised machines within a network will serve to protect the network before a botnet attack is executed by preemptively securing predicted attack vectors. By embedding this botnet defense within the network, we hope to be able to supply a substantial increase in protection against known and unknown botnet attacks.

Section 1: Introduction

Malicious software or code finds its way onto host computers by methods, such as email, peer-to-peer (P2P) file sharing, drive-by downloads, etc. These malicious software are usually designed to self-propagate, thus once a host is infected, the virus or worm will begin to search for new potential hosts to infect. The infected hosts, or *Bots*, comprise an ever network of bots referred to as a *Botnet*. As an increasing number of bots join the botnet, this process can easily grow in an exponential fashion.

Once the bot has secured its foothold on a host computer, it begins to listen for instructions from a command source known as the Command and Control (C&C). The command and control center is an external source that communicates with individual bots and gives them specific commands and updates. Currently, two main kinds of command and control systems exist: centralized and peer-to-peer. The centralized C&C is the more classic technique of the two and is the infrastructure we focused on for this project. It is characterized by a central server node that all bots are programmed to connect to in order to receive commands.

Known Bots and Botnet Families

The complexities of botnets arise from the wide breadth of techniques and methods that bots can implement. As a result, bots can be categorized into different botnet families based on several factors such as architecture, botnet control mechanisms, host control mechanisms, propagation mechanisms, target exploits and attack mechanisms, malware delivery mechanisms, obfuscation methods, and deception strategies. The vast majority of botnets use Internet Relay Chat (IRC) for C&C, although a few use other methods like hypertext transfer protocol (HTTP). Furthermore, these differences in architectural design can be vast. AgoBot features a modular design and extensive documentation, suggesting that the build team adopted many concepts from software engineering to build the final product. On the other hand, SDBot is a simple, primitive, and monolithic design by comparison. However, it was designed to be extended and customized, and thus hundreds of versions and mutations of these bots exist. Another differing behavior is of the botnet host. Some scan IP addresses to locate new hosts while others do not. Some push commands to hosts while others rely on the host to pull commands from a server.

Botnets can also be classified into families according to their traffic signatures for command and control. Well-known signatures are a probable indication that a botnet is active. As such, many protection schemes exploit this feature by scanning network traffic to detect botnets via signature pattern matching. Successful detection can lead to protection against entire families. Another approach is to target IRC servers that cater to botnets. If the main channel for botnet communication is shut down, the hosts will idle.

Observable anomalistic behaviors of networks and botnets can also be used to detect malicious activity. Many bots use IRC and HTTP protocols in order to communicate amongst each other in the network as well as with the C&C server. This traffic can be used to note abnormal traffic patterns such as IRC and HTTP traffic in parts of the network that do not allow it. Another sign of suspicious activity are communications that are in an an unreadable language or syntax. Secondly, because botnets often use Domain Name System (DNS) queries to find the location of the C&C server, frequent queries can be flagged as suspicious activity. Currently, botnet operators are improving the undetectability of their botnets by using Dynamic DNS names as well as frequently changing the location of their C&C servers to avoid detection. However, these methods still exhibit high levels of DNS queries and can still be detected. Other botnet-like characteristics include unusual domain queries like whiz.dns4bot.org, IP addresses to a given domain name that frequently updates, and abnormally fast response times.

Attacks, such as distributed denial-of-service (DDos) attacks, can be identified by characteristics such as several invalid transmission control protocol (TCP) SYN packets or invalid source IP addresses. Additionally, hosts tend to exhibit virus-like behaviors on the machine. These hosts carry out series of suspicious routines such as changing system files, disabling antivirus programs, modifying registries, and creating unknown network connections.

DDoS attacks generally involve flooding a server with ping requests to overload the network and thus crippling it. Commonly, the main intent is to temporarily or permanently eliminate a website or service so that it is unavailable to users. Zombie computers involved in a DDoS attack have recently been using true source IP addresses. Because firewalls have been attuned to filter out packages that have no true source IP address, using true source IP addresses has decreased the overall detectability of these zombie computers. Existing methods for DDoS attack detection are classified into categories based on abrupt traffic change, flow asymmetry, and distributed source IP addresses. Typically, a time series of data based on similar source IP addresses are required to identify a DDos attack. Preventing the attack might involve writing software at the router level to not allow the packets to flow after they have been identified as being associated with a DDoS attack.

Due to these various visible behaviors, many counter measures for detection over a network also exist. Thus, we must employ a series of mechanisms that maximizes our rate of detection on our network.

Section 2: Related Work

Botnet Detection

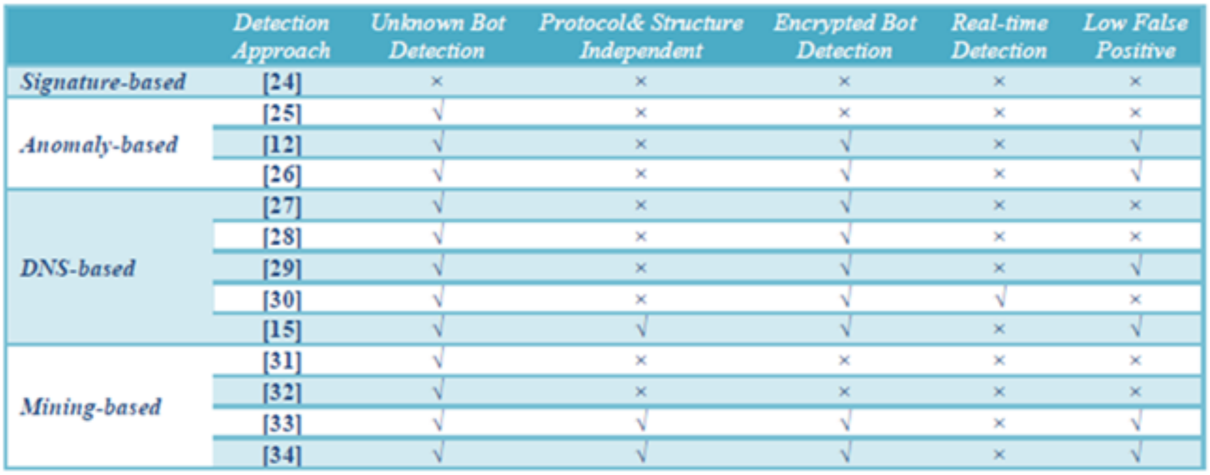
Passive network traffic monitoring to detect botnets can be divided into four main categories: signature-based, anomaly-based, DNS-based, and mining-based detection systems.

Signature-based detection relies on regular expressions to represent the signatures of currently known bots to detect suspicious activity. The shortcoming of this approach is that this technique is not functional for bots that are not yet known and additionally requires the implementation of new rules when new bots are discovered. This signature-based approach can be even more specialized to specifically IRC-based detection in which network traffic is monitored for odd or suspicious IRC nicknames, IRC servers, and uncommon server ports. However, this method has become outdated as botnet activity has shown increased use of HTTP protocols, while moving away from the common IRC method for communication. However, both HTTP and IRC use a centralized topology with a botmaster located at the focal point of the communication network. This design structure exhibits vulnerabilities if the location of the central botmaster can be detected and as a result, a rise in more decentralized networks, such as peer-to-peer networks, has gradually emerged.

In order for bots to get access to the C&C server, a bot must perform a DNS query to locate the server (this is usually hosted on a dynamic DNS provider). DNS-based detection detects bots by monitoring and detecting DNS traffic anomalies such as high frequency, regular periodicity, group size and various other abnormalities. This detection system tends to generate many false positives by falsely detecting popular domains that use DNS with short time-to-live (TTL). Additionally, this approach will not be effective if the bot is able to fake DNS queries.

Anomaly-based detection analyzes network traffic (ie: high network latency, high volumes of traffic, traffic on unusual ports, and unusual system behavior) in order to pinpoint nefarious activity. Botsniffer, a currently implemented botnet detector, uses spatial-temporal correlation based on the idea that bots within the same botnet will exhibit strong similarities in their activities and responses.

Lastly, mining-based detection utilizes several mining techniques such as machine learning to detect communication between botnets and their C&C. The most recent example of this is Botminer which clusters similar suspicious activity and compares these clusters in order to determine hosts that have similar communication patterns and malicious activity patterns. This method is protocol and structure dependent and thus makes it more robust than the aforementioned detection techniques. Table 1 compiles the efficacy of these different approaches.



	Detection Approach	Unknown Bot Detection	Protocol& Structure Independent	Encrypted Bot Detection	Real-time Detection	Low False Positive
Signature-based	[24]	×	×	×	×	×
Anomaly-based	[25]	√	×	×	×	×
	[12]	√	×	√	×	√
	[26]	√	×	√	×	√
DNS-based	[27]	√	×	√	×	×
	[28]	√	×	√	×	×
	[29]	√	×	√	×	√
	[30]	√	×	√	√	×
	[15]	√	√	√	×	√
Mining-based	[31]	√	×	×	×	×
	[32]	√	×	×	×	×
	[33]	√	√	√	×	√
	[34]	√	√	√	×	√

Table 1: Comparison of Botnet Detection Techniques [5]

Available Software

BotHunter

The idea behind BotHunter's approach is to analyze out-bound and in-bound communication flows across a network to determine any botnet activity taking place across the network. The system is designed to track bot activity in the absence of tracked inbound flows as well as a large time period between inbound and outbound flows. The BotHunter is stated to be located at the "boundary of the network" although it is unclear whether this means at the router-level.

The system is designed with three separate intrusion detection system (IDS) modules at the top level that listen to in- and out-bound traffic flows across the network. It communicates with a "dialog correlation engine" that determines and consolidates reports for suspected bots within the network. This system classifies dialog transaction into 5 distinct categories:

1. E1: External to Internal Inbound Scan
2. E2: External to Internal Inbound Exploit
3. E3: Internal to External Binary Acquisition
4. E4: Internal to External C&C Communication
5. E5: Internal to External Outbound Infection Scanning

The first module is called Statistical Can Anomaly Detection Engine (SCADE). Bots have approximately 15 different methods or "vectors" for attempting exploitation of a box, resulting in a large number of failed connections before successful infection. This module uses this bot characteristic and scans E1 and E5 type transactions for a weighted scoring system. Each score is based on the number of fails, the severity of ports used to determine the score for inbound scans and a slightly more complicated combination of scores to determine the grade outbound scans.

The second module is called Statistical payLoad Anomaly Detection Engine (SLADE). This uses a complex algorithm to determine whether the "lossy n-gram frequency" of a packet deviates from the standard normal profile.

The final module is called the Signature Engine and relies heavily on over 1000 Snort rules to determine if packets match patterns of known bot types. This module analyzes E2, E3 and E4 type transactions across the network.

These three modules deposit their findings into a network dialog correlation matrix data structure that tracks the evidence pertaining to certain host machines to see if they satisfy a minimum threshold for bot-like activity within a designated time interval [9].

BotMiner

Botminer is designed to locate already infected machines on a network without relying on tracking a certain type of C&C interaction. First, BotMiner separates traffic into Communication (C) and Activity (A) traffic planes. Then the system separately tracks A and C traffic and logs the packet information into files. The next step clusters these activities into groups with identical sources and destinations and then determines cross-cluster correlations.

Traffic monitoring is programmed in C for efficiency. C-Plane traffic capture uses an internally designed tool called capture which uses Judy library but only captures traffic across TCP and UDP flows. It compresses the data it logs into files of about 1 GB daily. The A-plane traffic capture adapts existing intrusion detection techniques implemented as Snort plugins. Specifically it borrows from BotHunter's SCADE module and egg download detection as well as using Snort for spam detection based on high volume of requests to many STMP servers from the same source.

The clustering of flows is the next step. C-plane clustering first filters out irrelevant data using basic-filtering to filter out flows not going from internal to external hosts and white-filtering to filter out requests made to well-known benign servers. Next, flows that share the same protocol (TCP or UDP), source IP, destination IP and port, are aggregated into the same "C-flow." These clusters are then converted to pattern vectors based on 4 characteristics (number of flows per hour, number of packets per flow, average number of bytes per packet, and average number of bytes per second). These measurements are divided into 13 quintiles to give the vector 52 different measurements. These C-flows are then clustered together using a 2-step X-means algorithm.

Finally there is a cross-plane correlation step to combine the clustering in both planes to determine the probability that host machine is a bot. a weighted score is determined for each host and it much be above a certain threshold to be considered a bot. This is taken a step further when actual hosts are clustered together if they appear in a the same Activity cluster and at least one C-cluster. A Dendogram with a Davies-Bouldin index is then used to determine how many distinct botnets are present in the network [10].

Section 3: Project Proposal

3.1 Overall Approach:

The project can be subdivided into four general sections. First, Snort Inline is utilized as a real-time network analysis tool to drop network packets from known bots. Second, packets that are not dropped are logged by normal Snort into a tcpdump formatted file. Third, a persistence engine is deployed to implement packet analysis functions on these Snort logs in order to pinpoint packets exhibiting bot-like behaviors. Lastly, this system is incorporated into a network relay module in order to communicate data between different sections of the network to a centralized server.

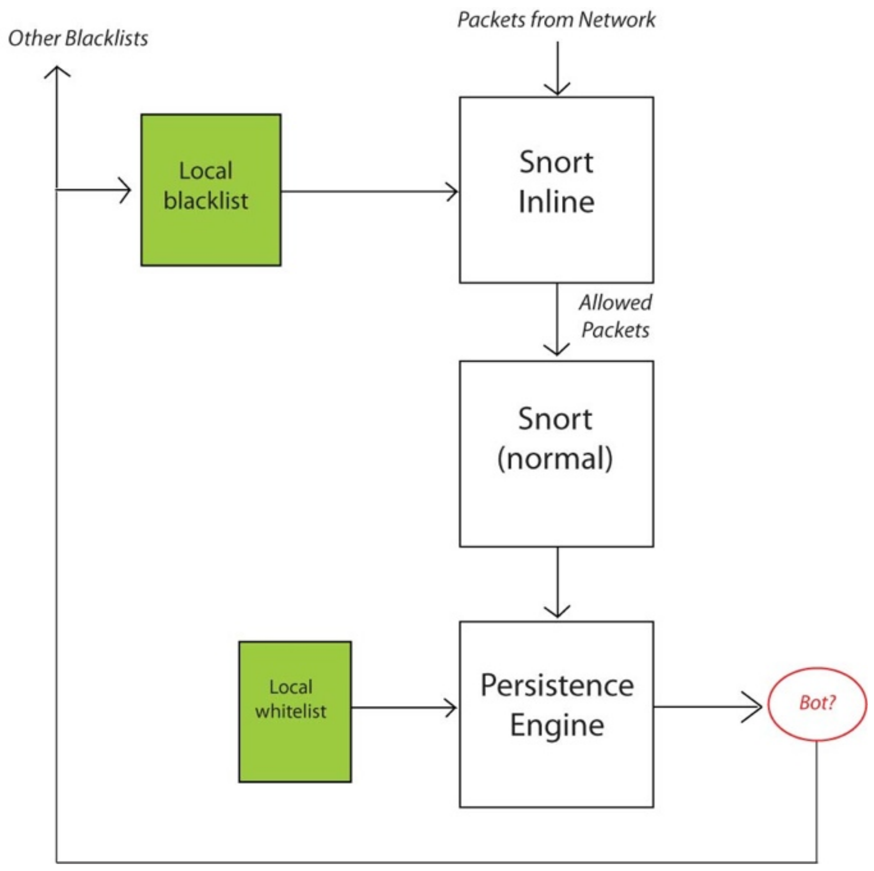


Figure 1: System Overview

Snort inline

The first step in our system is Snort Inline which utilizes the local blacklist to drop outgoing packets known to be bots. The initial blacklist uses the Emerging Threats open source project which holds a database of Snort rules for already known bots. Gradually, this local blacklist is updated with new detected bots. When a bot is flagged as a bot, a corresponding snort rule is generated and the blacklist is updated with the new information. In addition to each machine's local blacklist, the centralized server also holds a global copy of the blacklist. When a local blacklist is updated, the central server is notified and changed accordingly. These updates are proliferated throughout the network as the blacklists of individual machines within the network are also updated based on the server's new blacklist. As a result, communication is established throughout the network in order to push and pull periodic updates between the global and local blacklists.

Snort

Untouched traffic outputted from Snort Inline is then passively logged using Snort. The output file from Snort is in a tcpdump formatted file. For purposes of speed, a file is written to until a predetermined maximum file size has been reached. At this point, the file is closed and sent to a the persistence engine for further packet analysis.

Persistence Engine

The packet analyzer receives packet data from the parser in file format in order to prevent slowdown or failure due to multiple components operating at different speeds. In this sense, the parser will write output files at the rate at which it receives input from Snort. The analyzer, on the other hand, requires more time to process each packet and perform the necessary actions on it.

The persistence engine measures the regularity at which packets are being communicated to each remote destination. Because bots must communicate regularly to its C&C server to push or pull for updates, by tracking the frequency of packets being sent to a destination location over time, we can flag it to be a potential bot. Because certain websites are commonly visited (ie: google.com), this flag is alerted to the user to be verified or rejected as a bot. the user. This whitelist contains websites such as Yahoo, Youtube, and Wikipedia.com [12].

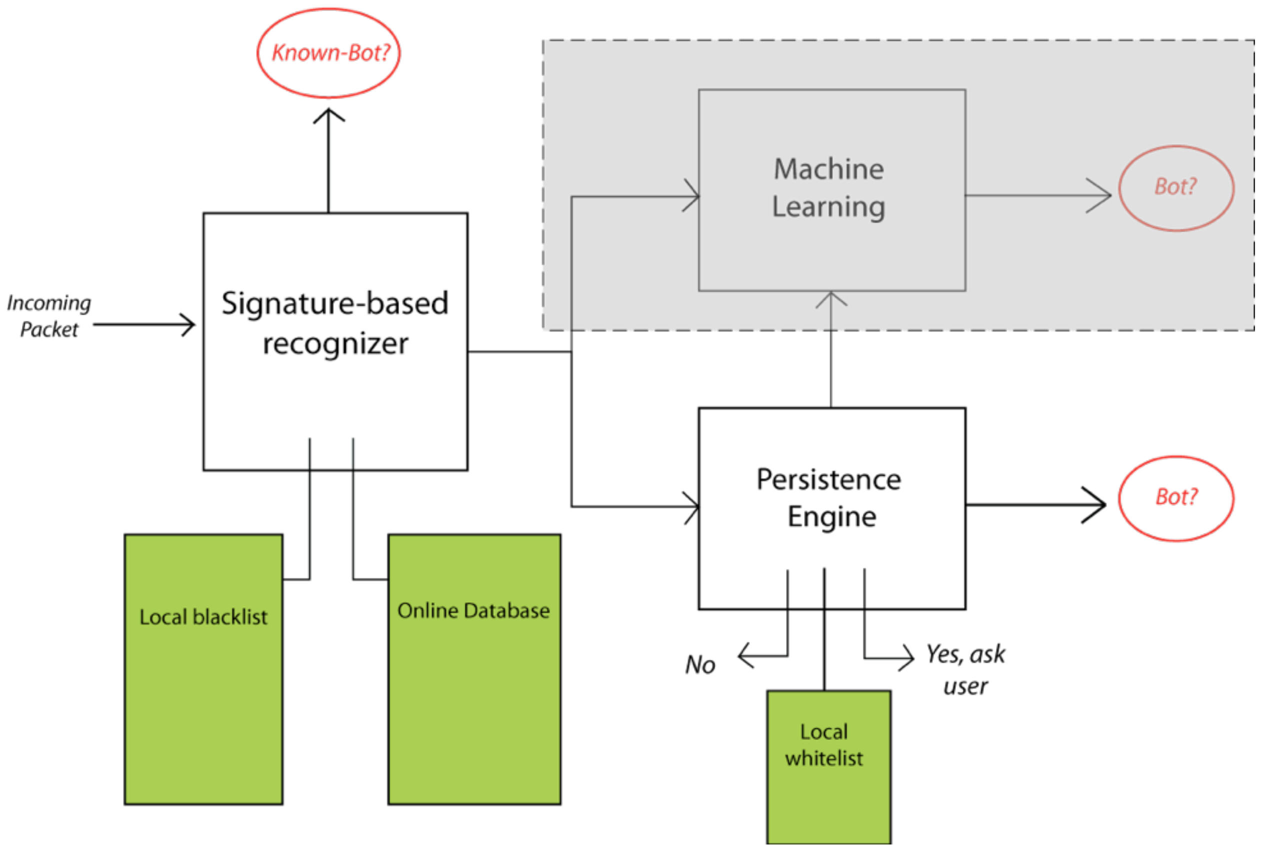


Figure 2: Packet Analyzer

The diagram above gives a more detailed overview of the packet analyzer module. Packet information first enters the signature-based recognizer, which cross-references packet metadata against two lists. The first is the program-specific blacklist which is generated based on local bot discoveries. The second is an updated list of bot signatures obtained from the Internet [11]. This initial filtration process is easy to implement, and greatly increases the reliability of detection. If a known bot signature is detected, the analyzer need not waste more effort in trying to determine whether a given packet is suspicious, and can immediately classify it as suspicious. This helps the other analyzer components increase their accuracy at detecting malicious packets by identifying already known ones which will be useful in a potential future implementation of machine learning.

Once the packet has been filtered by the recognizer, it is sent to the persistence engine (algorithmically based on Intel's Canary Detector [8]). The persistence engine analyzes outgoing traffic on a single computer by determining the 'persistence' of each host's communication with a computer in a manner independent of the volume or frequency of communication. The regularity of each unique 'atom' (a 3-tuple of [destination IP, destination port, transfer protocol]) is kept track of during a set of 'tracking windows.' For example, the tracking window could be set to 1 hour - and for each hour, the persistence engine marks whether or not an atom was seen using a bitmap. An atom is 'persistent' if the number of tracking windows it was seen in exceeds some predefined boundary percentage of the whole number of tracking windows. If an atom was seen during twelve 1 hour windows out of 24, it might be called persistent. But a different atom could be seen once every 24 hours and would still be regular, but would not be marked as persistent. Thus, in order to find every persistent atom, we analyze the same set of atoms using several different tracking window sizes (one could be 24 hours). Each time a new 'persistent' atom is discovered, the user is alerted and then prompted to either whitelist or blacklist the host. Hosts, such as google.com, which are persistent but benign, would be whitelisted and ignored in further analysis. The original whitelist consists of the most popularly visited websites in order to prevent the user having to reject commonly visited and obviously non-bot sites such as youtube.com and Yahoo. On a network level, the central server keeps track of a global list of persistent atoms in the network. For each new common atom, an alert is sent to the central location notifying it of an update. The global list is then updated and the information is relayed to the rest of the network.

Currently, the persistence engine prompts the user to determine whether a persistent atom is benign. However, for future implementations, this step could be replaced with an algorithmic check for benignancy.

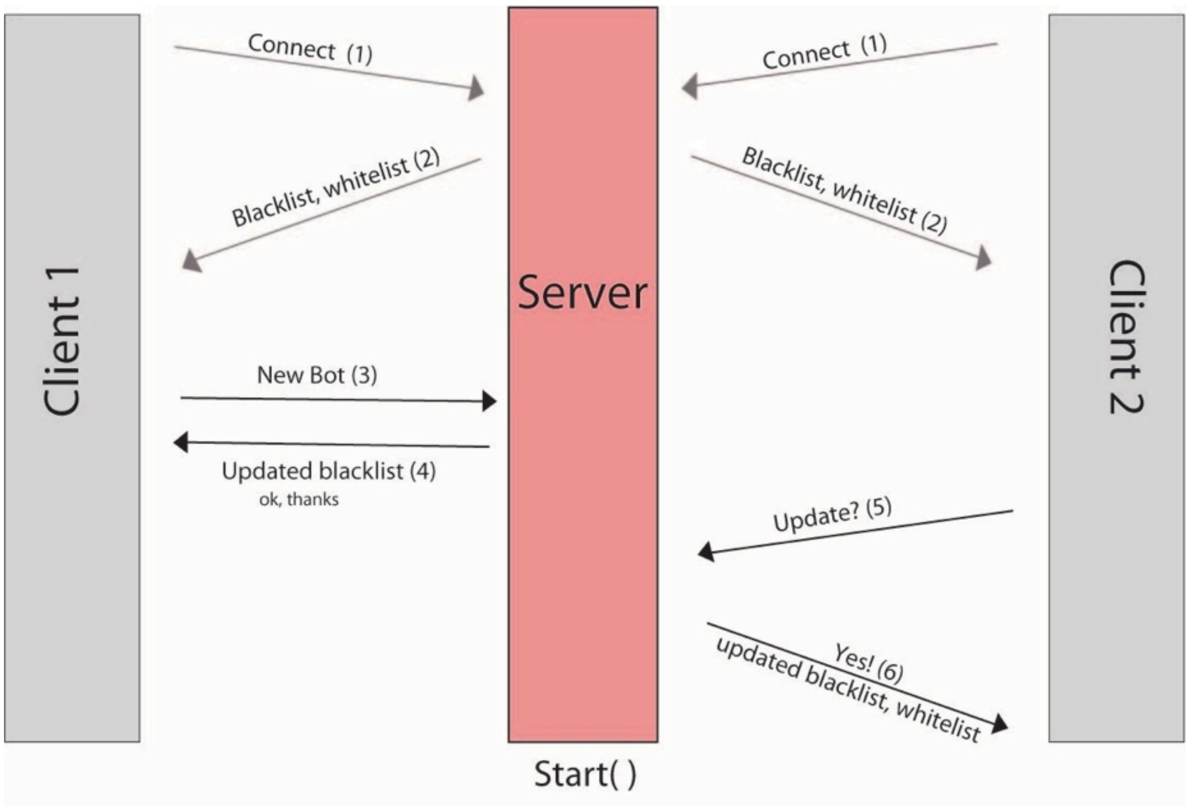


Figure 3: The Distributed System

The final component of the system is the network relay module. The relay module will enable all instances of the running software to communicate together across the large-scale network. As shown in figure 3, the software will be distributed across a large-scale network such that every instance will communicate with the central server for blacklist and whitelist updates. It is inefficient to commit one process to analyzing all traffic on such a large network; one of the determining factors of building a distributed system. As such, the process will be made much more efficient by monitoring many or all subnetworks within a network, for bot or botnet behavior.

Testing approach

Testing will first involve ensuring that the functions of our packet capture, parser, and pack analyzer are functioning correctly. To do this, we will ensure that a set of data dumped out will be parsed correctly in a straightforward manner.

Second, we will need to test whether the atoms created by our program are correctly classified as persistent. To test this, we will have to create a set of test files that log a particular packet at certain times so as to form a continuum characterized as being persistent. If our program reveals this as a persistent atom, then we have successfully characterized it as such. We will also test with generated traffic that may or may not be persistent, to test for false positives and general accuracy.

Third, testing of the network relay method will involve testing whether or not multiple instances of our program correctly receive and send messages that successfully update the current status of the whitelist and other necessary global parameters. This will be accomplished by installing the software on multiple virtual machines set up in a local network.

Lastly, we will conduct the overall system testing in order to ensure that our program successfully characterizes botnet traffic. In order to do this, we will have to find a continuum of network data that has already been characterized with a high degree of accuracy as to what is botnet traffic and what is not. We will feed it to our parser and analyzer. Over time, if our program relays an accuracy rate similar to what it is known to be then we have successfully characterized the traffic stream.

It was found that the most difficult task in testing for persistent packets was not testing the algorithm itself but obtaining testing data due to privacy restrictions and limited access to network data. Instead, we had to rely on obtaining network data from our professor's home system and as a result may not have been able to component test this portion of the system as thoroughly as would have been planned.

For future testing purposes, it would also be of interest to test our system by creating a 'honeypot' box or network.

3.2 Competing Approaches:

Passively Detect Botnet Traffic (Primarily IRC Botnet traffic)

IRC packets can be detected in the first communications between a program and an IRC server by looking for the "USER" or "JOIN" commands in the first bits of a packet body. Once IRC packets can be distinguished from other packets, several approaches can be taken to recognizing abnormal traffic. The "Response-Crowd-Density-Check" algorithm uses a threshold random walk, a random sampling approach, to recognize abnormal crowds ("a set of clients that connect to the same server" [6]) by their collective response surges [6]. Another detection algorithm calculates connections and disconnections as a percentage of total packets from which abnormal workloads can be determined [7] (perhaps this isn't applicable to real-time analysis and detection?). One novel, and possibly easier to implement, approach works based on the observation that IRC botnet packets' size as a function of time is periodic due to IRC ACK-PING-PONG cycles [no readily available source]. IRC packets that aren't PRIVMSG's could possibly be classified as benign traffic.

Typical Data Sources Used to Detect Botnets

Various data sources can be used to detect botnets. For instance, DNS data can be used to detect the spamming behavior of bots as well as communication behavior consist with bots such as DNS lookups for suspicious domains. Additionally, Netflow Data can be used to look at source and destination IP's of packets to note any suspicious traffic flow patterns. This is useful to detect botnet communications as well as certain botnet attacks. Packet content data is mostly helpful for signature-based detection algorithms. However, simple encryption such as https can easily block this information. Honeypot data is a fourth option and consists of a sacrificial host that is put on the network in the hopes that it will be harnessed as a bot. This provides insight into the activities and communication of a botnet. Lastly, host data and address allocation data are other options of data source possibilities used for detection.

DDoS detection

Detection of a Distributed Denial of Service attack presents an array of issues that places it outside the scope of the system being designed. Software for detection and by necessity prevention of DDoS attacks would need to be located at the router level. By the nature of a DDoS attack, placing it at the level of an individual machine would render DDoS detection software useless, because by the time the machine recognizes that it is being attacked it is already too late. In addition there has been little research into actual DDoS attack prevention and detection software and so even general strategies for designing software to do this is poorly defined. Because of a lack of expertise in router software design and lack of well-defined research in DDoS detection it does not seem feasible to include this feature in the system.

The Use of Honeynets for Primary Bot Detection

Previous research indicates very successful botnet detection schemes based on data collected from honeynets. Honeynets are networks of honeypots designed to communicate with the network being monitored in a safe and risk-free way. In doing so, a very large amount of data can be collected about various bot properties for every bot that infects a host on the honeynet. This is turn is used to identify traces of the bot on the protected network. This method shows great promise, especially with the development of more sophisticated honeynet systems, because it enables detection schemes to be based off of real bot data, rather than pattern recognition algorithms or other mining schemes. As appealing as this approach may sound, it would have been too difficult to maintain such a network of honeypots due to lack of resources and expertise.

Perceptron

When we have decided on the list of features that best represents packets, we can use perceptron, a linear regression algorithm, to automatically classify packets into the normal category or botnet category. Perceptrons are based on the idea that there exists a dividing hyperplane that separates points into different groups. In this space, the points the packets occupy will be defined by the features.

For example, imagine that the packets can be defined by two features: packet source and packet destination. We could map packet source to the x-axis and packet destination to the y-axis and produce a graph similar to the one above. The goal of the perceptron is to find the plane that best divides the packets into the legitimate (blue in the example) and botnet (red in the example) categories. Once this plane is found, all future incoming packets will be classified according to the side of the plane they fall on.

The algorithm requires a list of features and a labeled training set of network traffic. Feature selection is discussed later and atoms have already been explained; a training set should be provided by professor Smith. One possible challenge is that going through the training set and properly labeling each packet as either legitimate or botnet could prove time consuming or difficult. Another is the possibility that models for classifying traffic are not linear. But assuming that these obstacles are overcome, perceptron should be very implementable.

Machine Learning

Our software will use machine learning in conjunction with atom persistence to automate the whitelist/blacklist component. The relevant algorithm and packet feature selection process is outlined below.

Feature Selection

It is most often desirable to only use a certain set of features for the model. Using fewer features gives a simpler model that is less likely to over fit the data. Additionally, constraining feature sets gives a better idea of what features are actually important. This will also improve the speed of classification. The most important feature set that predicts malicious traffic must be determined. There are multiple ways to analyze these features.

Features can be tested individually and based on the prediction accuracy percentage, the top performing features can be combined. This is suboptimal because the top features could be measuring the same thing in which case you would have a set of redundant features. It is also very important to see how the features interact with each other because the model will not be based on their independent measurements but rather their combined efforts.

Another approach, referred to as “stepwise” feature selection, is to start with an empty feature set and add features noting the change in the model’s accuracy. For instance, if a model based on five features was desired, the first feature added would be the one that performed best individually. The second feature to be added is determined by trying each of the remaining features in conjunction with the first feature to see which feature further improves the accuracy of the overall model. This process of adding features based on improved accuracy would continue in this fashion until the number of features desired has been reached or features are no longer contributing improved accuracy to the system. This selection process is most likely the best and most accurate method to be integrated into our current system and thus should be considered for future modifications to BARD.

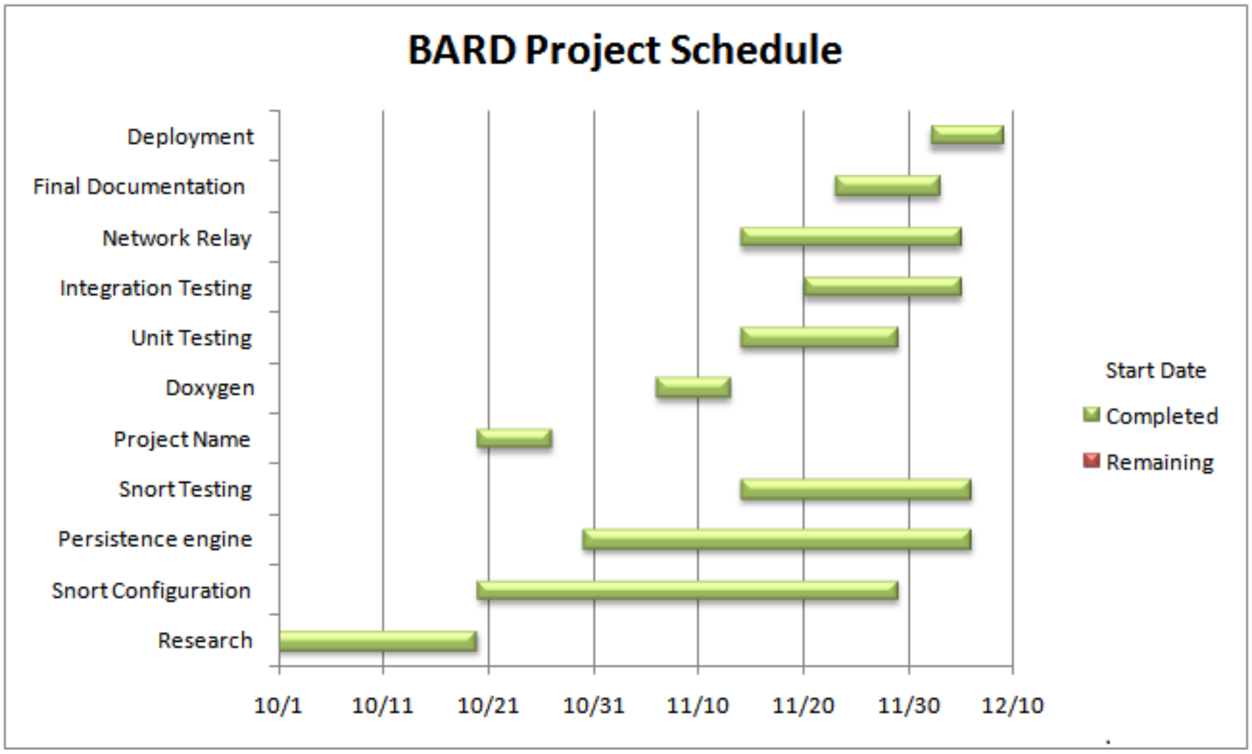
Choices of network analysis tools

There is a wide array of network security tools available for use in this project such as tcpdump, wireshark, and the pcap library. However, many of them perform similar tasks and ultimately Snort presents the most flexibility and extensibility for which there exists great support.

3.3 Resource requirements:

The operating system that will be used for testing is Ubuntu 10.04. Necessary applications include Snort, Snort Inline and VMWare. Additionally, we used Doxygen for our documentation system to generate online documentation in HTML. We used subversion for our method of version control. Our source code uses Python to meet the limitations of our group’s resources. We decided upon Python due in order to achieve our projected deadline of December 9th (approximately two months) and Python was able to aid us in increasing productivity and debugging speed because there are less likely to have bugs such as pointer errors and register assignment, and were easier to be found by the compiler. Future machine learning components will be coded in Python, as it is most suited to these algorithms.

Section 4: Schedule and Organizational Structure



Roles/Responsibilities

Our group approached teamwork using a programming group-like model with a mix of appointed roles for positions dealing primarily with overhead and coordination such as project coordinator, systems administrator, and tester. Because of the small size of our group, individuals naturally arranged themselves to roles maximally utilizing their skill sets such as those with prior knowledge of Snort and strong backgrounds in Python for coding the persistence engine. As specific tasks were completed (ie: Snort Inline configuration, Systems administration, etc), individuals moved on to aid others in tasks that were not yet completed (ie: persistence engine and network relay module). This ensured that no one was left with nothing to do and everyone was working on a task at all times.

Additionally our team decided to hold weekly 6-hour hack-meetings on Saturdays. We decided on this approach based on the idea that large blocks of time are more productive than several one-hour meetings because sustained concentration significantly reduces thinking time Additionally, by brainstorming and coding together in one room, we were able to eliminate communication overhead and maximize our efficiency. By utilizing extended periods of time, it was not necessary to spend extra time trying to figure out where we had left off during the previous meeting.. We also collaborated mid-week for one hour to regroup and discuss what we had accomplished during the previous week and prioritize the coming week's goals and tasks. This further ensured smooth communication between all individuals and allowed for everyone to come to the hack sessions prepared and ready to efficiently complete his or her task.

1) ***Project Coordinator*** – Erika

- Documentation
- Email communication
- Room/Meeting coordination
- Record Minutes

3) ***Systems Administrator*** - Gary and Anthony

- Set up software tools/IDEs
- Installing, supporting, maintaining any necessary servers

4) ***Tester*** - Stuart and Gary (assistant tester)

5) ***Team Snort*** – Alex, Jeremy, and Stuart

6) ***Main Programmers*** - Jeremy, Alex, Anthony, Mish

*Note: Everyone will contribute to programming.

Section 5: References

[1] Cheng, Jieren. “DDoS Attack Detection Algorithm Using IP Address Features” Xiangnan University, China. 2009

[2] Specht, Stephen. “Distributed Denial of Service: Taxonomies of Attacks, Tools and Countermeasures” Princeton University. 2004.

[3] Barford, Pual. “An Inside Look at Botnets” University of Wisconsin. 2005.

[4] Bailey, Michael. “A Survey of Botnet Technology and Defense” University of Michigan. 2009

[5] Feily, Maryam, Ramadass, Sureswaran, and Shahrestani, Alireza. “A Survey of Botnet and Botnet Detection” Universiti Sains Malaysia 2009.

[6] Guofei Gu, Junjie Zhang, and Wenke Lee, “BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic,” Georgia Institute of Technology

[7] James R. Binkley, Suresh Singh, “An Algorithm for Anomaly-based Botnet Detection,” Portland State University

[8] Chandrashekar, Livadas, Orrin, Schooler, “The Dark Cloud: Undestanding and defending against botnets and stealthy malware,” Intel Technology Journal, Vol. 13, Issue 2, 2009..

[9] Gu , Porras , Yegneswaran , Fong, Lee, “BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation,” Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, 2007

[10] Gu, Perdisci, Zhang, Lee, “BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection,” Proceedings of 17th USENIX Security Symposium on USENIX Security Symposium, 2008

[11] Emerging Threats, http://www.emergingthreats.net/

[12] Alexa Top 500 Sites On The Web, http://www.alexa.com/topsites