Detecting Botnets in the Wild

Chavin (Kris) Udomwongsa

Department of Computer Science and Department of Mathematics, Dartmoth College

Email: chavin.udomwongsa.24@dartmouth.edu

Botnets are currently a major threat to online services, with the potential to deal extensive damage e.g. through a distributed denial-of-service attack (DDoS). We take a dataset of synthesized botnet data, real botnet communication data and with background internet traffic, all in the form of a communication graph, and apply graphical machine learning to try to predict whether or not a device is part of a botnet based on topographical features of the graph. There remains a lot of potential in graphical machine learning to detect botnets, its current limitations seem to lie in feature selection and the scalability of computing such features if implemented on a large scale.

Background

A botnet is a collection of compromised computers being controlled by an adversary, which can be used to send massive amounts of spam to take down a victim, known as a Distributed Denial-of-Service Attack (DDoS). As the number of electronic devices grows exponentially, the potential for botnets also grows, making them capable of dealing more damage to targets. Botnets are currently a major source of network attacks, and as we become more reliant on

technology and services, the taking down of certain service could lead to potential harm. The detection of botnet is therefore a crucial line of defense in preventing these attacks.

The phases of a botnet's life could be divided into the following four sections, formation, command and control, attack, and post attack (*1*). Formation refers to the creation of the botnet, the spreading infection of other machines on the Internet. Command and control refers to the infected computers being provided instructions on a relatively regular basis. The attack phase refers to all of the infected machines doing some malicious task, for example sending spam to a target. Finally, the post attack phase is where bots could be detected and removed, and could lead into the formation or command and control phase once again.

Working towards the detection of botnets is not novel, however, the use of different techniques and other kinds data can be explored. One well-explored approach is by looking at network traffic. According to Saad et al. (*2*) there are three types of network traffic identification for botnets:

1. **Port-based Identification**: This is referring to the port of the device that the network packets are sent to and from. The fatal flaw to identifying botnets based on the port alone is the sheer number of false positives, normal network traffic will likely use the same ports as botnets, and if that port is flagged as used by botnets, the adversary could simply use another unregistered TCP/UDP port. In conclusion, this form of identification is easy to avoid and leads to a high number of false positives.

2. **Packet Payload-based identification**: This is referring to the contents of specific network packets, which raises concerns with privacy, as it would allow the machine learning algorithm to view private communication between people if implemented. Furthermore, this is computationally expensive, and if implemented, would drastically slow down network performance. Despite this, it has the potential to be very effective at detecting

botnets. Some work done on this have looked at network layer data, and Internet Relay Chat nicknames (*3–5*).

3. **Behavior-based**: This is referring to a combination of information that can be extracted without looking at the contents of packets, examples would be the size of the packet, the duration of the sessions, the rate of packets sent etc. Communication patterns have the potential to be a great form of identification for botnets. As it doesn't infringe on privacy, and is relatively inexpensive, many researchers have looked at the behavior of devices to identify botnets (*6–9*).

Regarding the three types of network traffic identification methods, Gu et al. (*10*) claims that all of the three types can be adapted by the adversary, meaning that for a machine learning model to effectively detect novel botnets, its training data must be updated, because the adversary can modify the any of the features described.

Whether or not botnets are centralized should also be considered, as their behavior drastically differs depending on this fact. A lot of botnet research that focused on mostly centralized botnets find their models very ineffective in detecting peer-to-peer (p2p) botnets. Especially at the time of the emergence of p2p networking, a lot of research focused on p2p botnets (*2, 7–9*).

Other researchers have explored beyond network traffic, Zhuang et al. researched detecting botnets through email spam records (*11*). The focus of this paper lines up with the novel work of Zhou et al. (*12*), where our focus is on the topological features of botnets. Using topological features, we attempt binary classification on the devices in a network, to predict whether or not they're part of a botnet.

Data

We look at a communication graph, which is a graph where its nodes represent devices, and an edge between two nodes indicate communication between the two devices. We use a dataset provided by J. Zhou et al. (*12*). This data provided includes: synthesized p2p botnet traffic, real decentralized p2p botnet traffic, the centralized botnet C2 that was captured in 2011, two botnets from new malware, and background traffic provided by CAIDA. The background traffic is vital to our botnet detection, as we must be to distinguish infected devices from regular ones.

The data provided is in HDF5 format, and is extracted into a python object with a library function provided by Zhou et al. (*12*). It is divided up into 7 datasets, named after the botnet or the topology of the botnet:

- **chord**: a synthetic botnet with 10,000 botnet nodes
- **debru**: a synthetic botnet with 10,000 botnet nodes
- **kadem**: a synthetic botnet with 10,000 botnet nodes
- **leet**: a synthetic botnet with 10,000 botnet nodes
- **c2**: a real centralized botnet from 2011, made up of about approximately 3,000 botnet nodes
- **p2p**: a real decentralized p2p botnet, made up for approximately 3,000 botnet nodes

Method

The data handling, feature extraction, and model construction and model evaluation can be seen on the Jupyter Notebook `.ipynb` file likely attached to this paper, but all of the aforementioned items will also be described in this section.

Data Handling

As mentioned in the data section, the data provided was in a HDF5 format, but we were provided a library function that would extract the graph data into a python object. The python object consisted of two arrays of node labels, the nodes on the same index have an edge between them. Furthermore, there was also an array of 0's and 1's (0 meaning not part of a botnet, 1 meaning part of a botnet), the index corresponds to the node label. For example, if the value at index 2 is 1, that means the node with label 2 is part of a botnet.

With that, we constructed a graph by looping through the arrays within the python object, the graph was constructed using the `NetworkX` python framework, which allows us to easily manipulate and access data regarding our graph. Below is the python code where given the data extracted from the helper library function, and any graph name, a networkx graph object is created.

```
import networkx as nx

def create_graph(data,graph_name):
    G = nx.Graph(name=graph_name)

    node_labels = data["y"]

    # go through each node and add a node to the output graph G
    for index in range(0,len(node_labels)):
        G.add_node(index,evil = node_labels[index])

    edge_node1 = data["edge_index"][0]
    edge_node2 = data["edge_index"][1]

    edge_labels = data["edge_y"]

    for index in range(0,len(edge_node1)):
        G.add_edge(edge_node1[index],edge_node2[index],evil = \
        edge_labels[index])

    return G
```

Next, to make feature extraction easier, we loop through each node, and for each node n, we find the node's neighborhood. By this, we mean a dictionary, where its keys are the node labels that the selected node n is adjacent to, and the value for that key are node labels adjacent to n that are also adjacent to the node label in the key. For example, given four nodes 1,2,3 and 4, let's say that there exists edges between 1 and 2, 1 and 3, 2 and 3, and 3 and 4. If we're trying to find the neighborhood for node 1, there would be two keys, 2, and 3. For key 2, the value would be an array of [1,2,3]. For key 3, the value would be the array [1,2,3] also. Node 4 is not in the values for key 3 because the node 4 isn't adjacent to node 1. This is to find relationships between nodes adjacent to each selected node, and will be used to extract features later. Below is the code that adds a neighborhood as an attribute to the networkx graph object

```
def add_neighborhood(graph):

    adjacency = graph.adjacency()

    # for each node
    for node_label in list(graph.nodes):

        # add number of neighbors attribute
        graph.nodes[node_label]["neighbor_count"] = \
        len(graph.adj[node_label]) - 1

        out_dict = {}

        neighbors_neighbors = []

        # for each neighbor
        for neighbor in graph.adj[node_label]:

            for other_neighbor in graph.adj[node_label]:
                if neighbor != other_neighbor and \
                neighbor in graph.adj[other_neighbor] and \
                not other_neighbor in neighbors_neighbors:
                    neighbors_neighbors.append(other_neighbor)

            out_dict[neighbor] = neighbors_neighbors
```

```
graph.nodes[node_label]["neighborhood"] = out_dict
```

Features

Keep in mind that we're doing binary classification for each node, so these features are corresponding to each individual node. To begin with, we look at shapes that each node is in, for example how many triangles a given node is in. We utilize the neighborhood that we found for each node. Recall the example above, to get the number of triangles that are in a given node, we extract the values from the neighborhood dictionary for that given node, then the number of triangles would be $\frac{\sum(\alpha-2)}{2}$ where α is the length of the array for each value in the dictionary and the summation covers the entire neighborhood dictionary for that node. Refer to the Jupyter Notebook for clarity. Triads of nodes are potentially the same topology as p2p botnets, so the number of triangles a given node is in may be very helpful to predicting whether or not the node is part of a botnet. Below is the function for getting the number of triangles for each node in a given graph.

```
def get_num_of_triangles(graph):

    # for each node
    for node in list(graph.nodes):

        # if it has no neighbors, it has no triangles
        if graph.nodes[node]["neighbor_count"] == 0:
            graph.nodes[node]["num_of_triangles"] = 0
            continue

        # loop through each neighbor in neighborhood
        num_triangs = 0
        for key,value in graph.nodes[node]["neighborhood"].items():
            # don't count if the neighbor is the target node itself
            if key == node:
                continue
```

```
        # add up length of value - 2
        if len(value) != 0:
            num_triangs += len(value) - 2

    # error check
    if num_triangs % 2 != 0:
        print("Something went wrong with node " + str(node))

    # divide sum by 2
    num_triangs /= 2

    graph.nodes[node]["num_of_triangles"] = num_triangs
```

Finding the number of quadrilaterals that each node is in, or finding larger shapes would have also been worth exploring, but we were limited by our computing power. It would've taken too long to find the number of quadrilaterals or larger shapes that each node is in.

We look at the local clustering coefficient of each node, which is simply the number of pairs of a given node's neighbors that are also adjacent to each other, divided by the total number of pairs of a given node's neighbors. The clustering coefficient is used commonly in social network graphs, as a friend of a friend is more likely to get to know each other. Similarly, if a device were to be infected and be part of a botnet, other devices connected to that device would be more likely to be infected.

We look at the degree centrality of each node, which is a normalized version of looking at the degree of each node, relative to the rest of the graph. It is the simplest centrality measure to compute. This will emphasize the nodes that have really large degrees, and is simply a better alternative to just looking at the degree of a node. The degree centrality is a way of measuring the influence of a given node by looking at how many other nodes it's communicating with, the higher the degree, the more central the node is.

We consider the eigenvector centrality (also known as eigen-centrality or prestige score). This is more complex that degree centrality, a node with a smaller degree may have a very high

eigenvector centrality if the few connections that node has is very well-connected. This may be helpful in finding devices in the outer ring of a centralized botnet, as their degree may not be that high. The variant of the eigenvector centrality (pagerank) is famously used by Google's search engine. Although eigenvector centrality is usually used for directed networks, it still remains a helpful feature to include.

We also look at the maximum number of cliques that each node is in. A clique is defined as a fully connected sub-graph, so this feature measures for each node, how large the largest well-connected network that it's a part of is.

Model Construction

We use the following models:

- Random Forest Classifier
- Linear Support Vector Classifier
- Support Vector Classifier
- Multinomial Naive Bayes
- Stochastic Gradient Descent
- Decision Trees
- KNeighbors
- Logistic Regression

Results and Discussion

We expect the greatest limitation to the accuracy of our model is feature selection, there remains many features worth exploring, some of which currently aren't scale-able if trying to implement the model in the real world. This leads us to the other limitation of this study: another major limitation in training and testing capability was the device running the code. To train just the `chord` dataset, it would take the laptop over 3 weeks of consistent running, and that's only one of 7 datasets. For this reason, testing was only done on 6 out of 768 graphs in the `chord` dataset alone.

Using Only One Graph

First, it's worth considering the performance of the models when using only 1 graph from the dataset, below are the results from running the different models on one graph. True positive refers to when the model guessed that a given node was part of a botnet and was correct, true negative is where the model guess that a node was not part of a botnet and was correct, false positive and false negative are the same except the model is wrong.

Firstly, it's worth noting that accuracy alone doesn't give us much about the performance of the model, since all the botnet nodes made up about 7% of the nodes on the graph, predicting that none of the nodes were part of a botnet yielded an accuracy of about 93%, it is therefore better to consider the F1 score and AUC score. F1 score is better suited for our study, as it also considers imbalanced datasets, and our dataset is imbalanced.

Notice the models with an F1 score of 0, the Support Vector Classifiers and the Stochastic Gradient Descent Classifier, these three models are simply always assuming that the given node is not part of a botnet and therefore have no true positives.

An interesting observation is the fact that the model with the highest AUC score happened to be the model with the lowest accuracy: the Multinomial Naive Bayes model. This is because

Machine Learning Model Performance								
	Random Forest	Linear SVC	SVC	Multinomial Naive Bayes	SGD Classifier	Decision Trees	K Neighbors	Logistic Regression
Accuracy	0.942	0.930	0.931	0.710	0.931	0.940	0.948	0.930
True Positive	5073	0	0	9781	0	4910	4889	70
True Negative	130414	133890	133891	92347	133891	130309	131481	133741
False Positive	3477	1	0	41544	0	3582	2410	150
False Negative	4927	10000	10000	219	10000	5090	5111	9930
F1 Score	0.547	0.0	0.0	0.319	0.0	0.531	0.565	0.014
AUC Score	0.741	0.500	0.500	0.834	0.500	0.732	0.735	0.503

Table 1: Multiple Models on 1 graph from chord dataset

it has the highest number of true positives, however, it's worth noting its very low F1 score, which likely comes from the fact that it also has 41544 false positives. Since an AUC score of about 0.5 is pretty much the same as random guesses, the two Support Vector Classifiers and Logistic Regression Model seem to struggle with the given features.

The best-performing models when provided with only one graph would be between the Random Forest Classifier and the KNeighbors Classifier, closely followed by the Decision Trees classifier. The KNeighbors classifier has the best accuracy, and the best F1 score, but the Random Forest classifier has a higher AUC score.

Using Multiple Graphs

It is insightful to compare the performance of models provided only one graph with the same models when provided multiple graphs, specifically, 6 graphs. This is still only using the `chord` dataset.

Machine Learning Model Performance								
	Random Forest	Linear SVC	SVC	Multinomial Naive Bayes	SGD Classifier	Decision Trees	K Neighbors	Logistic Regression
Accuracy	0.946	0.930	0.930	0.798	0.930	0.942	0.949	0.930
True Positive	34510	15	0	24455	0	35797	34300	440
True Negative	912896	931947	931981	774920	931981	907601	916204	931149
False Positive	19085	34	0	157061	0	24380	15777	832
False Negative	35490	69985	70000	45545	70000	34203	35700	69560
F1 Score	0.558	0.0004	0.0	0.194	0.0	0.550	0.571	0.012
AUC Score	0.736	0.500	0.500	0.590	0.500	0.743	0.737	0.503

Table 2: Multiple Models on 6 graphs from chord dataset

There remain some similarities with the previous graph, notably, the poor performance of both Support Vector Classifiers and the Stochastic Gradient Descent Classifier. These three models continue to nearly always predict that a given node is not part of a botnet. The cost function also could've been weighted for all of these, to reward correct predictions more, which could have prevented the lack of positive guesses from these three models. The Logistic Re-

gression model also saw small changes in performance.

Besides for that, there are a few encouraging differences between the outcomes of using only 1 graph compared to using 6 graphs; We see improvement for the Random Forest Classifier, and some slight improvements for the Decision Trees Classifier and the KNeighbors Classifier. The Multinomial Naive Bayes Classifier improved in overall accuracy, but had a worse F1 score and AUC score, it also no longer has the most true positives, which goes to show how the Random Forest classifier, Decision Trees Classifier, and KNeighbors Classifier improved with more data.

Conclusion

In conclusion, this study attempts a graphical machine learning approach to detecting botnets given a communication graph, our results indicate which models seem to do well with the given graphical features provided, furthermore, this study also demonstrates the difference (or lack of) in performance for models when provided more data.

This study's largest limitation is undoubtedly our lack of computing power, the heavy majority of the data available remains unused and unexplored simply because of how long it would've taken to run the code. Furthermore, harder-to-process features weren't used: finding larger shapes that the node was also part of, and betweeness centrality are among those features that weren't considered due to how long it took to find for each node. Access to more computation will undoubtedly allow far more results and progress.

References

1. J. Leonard, S. Xu, R. Sandhu, A framework for understanding botnets (2009).

2. S. Saad, *et al.*, Detecting p2p botnets through network behavior analysis and machine learning (2011).

3. J. Goebel, T. H. Rishi, Identify bot contaminated hosts by irc nicknamme evaluation. (2007).

4. A. Karasaridis, B. Rexroad, D. Hoeflin, Wide-scale botnet detection and characterization. (2007).

5. R. Perdisci, W. Lee, Method and system for detecting malicious and/or botnet-related domain names (2018).

6. E. Stinson, J. Mitchell, Characterizing bots' remote control behavior (2007).

7. F. Giroire, J. Chandrashekar, N. Taft, E. Schooler, D. Papagiannaki, Exploiting temporal persistence to detect covert botnet channels (2009).

8. D. Liu, Y. Li, Y. Hu, Z. Liang, A p2p-botnet detection model and algorithms based on network streams analysis (2010).

9. S. K. Noh, J. H. Oh, J. S. Lee, B. N. Noh, H. Jeong, Detecting p2p botnets using a multi-phased flow model (2009).

10. G. Gu, R. Perdisci, J. Zhang, W. Lee, Botminer: clustering analysis of network traffic for protocol-and structure-independent botnet detection. (2008).

11. L. Zhuang, *et al.*, Characterizing botnets from email spam records (2008).

12. J. Zhou, Z. Xu, A. M. Rush, M. Yu, Automating botnet detection with graph neural networks (2020).