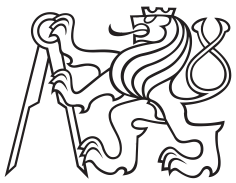
Bachelor's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Graph-Based Analysis of Malware Network Behaviors

Daniel Šmolík
Open Informatics, Computer and Information Science
smolidan@fel.cvut.cz

May 2017
Supervisor: Ing. Sebastián García, Ph.D.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Daniel Š m o l í k

Study programme: Open Informatics

Specialisation: Computer and Information Science

Title of Bachelor Project: Graph-Based Analysis of Malware Network Behaviors

Guidelines:

1. Review the state-of-the art methods for malware network behaviors analyses with special attention to graph-based techniques.
2. Propose and implement a graph-based method to detect malware infections and separate them from normal traffic. The problem to solve is how to identify infected computers in the network based on the behavior of all their connections.
3. Experimentally evaluate the proposed solution on the malware datasets of the Stratosphere project (https://stratosphereips.org). Especially in the CTU-13 dataset.
4. Critically analyze the results and propose further extensions of the solution with respect to its applicability to the real time detection of infected computers.

Bibliography/Sources:

[1] Sommer Robin, Paxson Vern - Outside the Closed World: On Using Machine Learning for Network Intrusion Detection - IEEE Symposium on Security and Privacy 2010
[2] Noble Caleb C, Cook Diane J - Graph-Based Anomaly Detection - University of Texas at Arlington 2003
[3] Jha Somesh, Fredrikson Matthew, Christodoresu Mihai, Sailer Reiner, Yan Xifeng – Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors – University of Wisconsin-Madison 2013
[4] Jusko Jan, Rehak Martin - Identifying Peer-to-Peer Communities in the Network by Connection Graph Analysis - International Journal Of Network Management 2014

Bachelor Project Supervisor: Ing. Sebastián García, Ph.D.

Valid until: the end of the summer semester of academic year 2017/2018

L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 9, 2017

Acknowledgement

I wish to express my sincere thanks to Ing. Sebastián García, Ph.D. for sharing his expertise and his valuable guidance. I would also like to thank my family for support during my studies.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 25. 5. 2017

.....

Abstrakt / Abstract

Existuje mnoho různých rodin malware a každá se vyznačuje jinými vlastostmi. Cílem této práce je zaměřit se na detekování škodlivého chování pomocí odchozí síťové komunikace. Naše hypotéza je, že tato škodlivá komunikace má sekvenční behaviorální vzory. Představujeme novou grafovou reprezentaci odchozí komunikace, kde jako vrcholy grafu používáme trojice (IP adresa, port, protokol). Myslíme si, že tato reprezentace může být užitečná při detekování vzorů programem i lidským okem. Pro předpověď byl použit algoritmus Random Forest. Testování proběhlo na datech normálních uživatelů, nakaženého počítačů a normálních uživatelů, jejichž počítače byly později nakaženy. Byli jsme schopni detekovat škodlivou komunikaci až s 97% úspěšností.

Klíčová slova: analýza grafu, detekce malware, strojové učení, analýza sítě

Překlad titulu: Analýza chování Malware v síti pomocí grafu

There are many malware families and every each of them has some unique features. The aim of this work is to focus on detecting malicious behavior using leaving network communication. Our hypothesis is that this malicious communication has sequential behavioral patterns. We present a new graph representation of leaving network communication using (IP address, port, protocol)-triplets as vertices. There is an edge between two vertices if they come one after the other in the record of the leaving communication of the inspected host. We think this representation might prove useful in detecting the patterns by a program and even by a naked eye. Random Forest algorithm was used for predicting. Testing was done against datasets of normal users, infected hosts and normal users that are later infected. We were able to detect malicious communication with up to 97% accuracy.

Keywords: graph analysis, malware detection, machine learning, network analysis

Contents /

Tables / Figures

Chapter 1
Introduction

The rising number of devices connected to the Internet brings also the rising number of successful tries of exploiting people's trust and inexperience. A great amount of these tries end with having a running software on a user's computer without his knowledge. There are many families [1] of such malicious software (or malware), depending on its features and purpose. In this work, we want to focus on detecting malware which sends information somewhere out of a local computer. Communicating with outside world might have multiple reasons — it can simply send data about the infected device (passwords, information about credit cards, etc.) or the malicious program might try to communicate with its command and control server (C&C)[1]. In that case, the computer of an unsuspecting user would be part of a botnet[2] and might try to communicate even with other bots in the system or to be a part of an attack on another computer. Also to hide before detecting tools C&C operators use P2P networks to remove the dependency on fixed servers. All cases have in common the effort to communicate with their master, but they differ in their scale and frequency and other features of the network communication. This difference makes it harder to detect them by a single tool. Another problem, when attempting to detect a malware, is that some botnets use HTTP protocol to hide themselves in normal traffic [4].

Our goal is to find such features that would help us predict whether a computer is infected or not.

We hypothesize that, as a malware tries to communicate with its master, other bots or it attacks someone else in the outside world, we could find patterns in a graph representing this communication and recognize it as malicious.

In this work, the incoming communication is not used at all, we only care about network communication that is leaving a host inspected by us and from this communication, we build a directed cycle graph. Transported data itself are not used either. This is good since we don't want too much to interfere with a user's privacy and also because some malware use an encryption algorithm to hide their communication [5]. From this graph, various features are being extracted, such as the number of nodes, number of edges, number of self-looping nodes, etc. Malware programs tend to connect to their master server(s) or other bots in sequential steps [6] following an attacker's algorithm and on a regular basis [2, 6], therefore we expect graphs of malicious and normal traffic to have different structures and properties.

We the found features in Random Forest algorithm to make a classifier. The built graph itself is also used in our displaying tool so we can see the graph with our own eyes.

[1] Command and control infrastructure (servers and other devices) is used to control malware on remote devices [2].

[2] A botnet is a collection of devices infected by a malware and controlled by the author of the malware. [3]

Chapter 2
Related Work

Using graphs to represent behiavior is nothing new, there is a number of works related to the topic of using graphs to analyze network traffic.

One of the most related works to ours is [6]. Garcia in this paper describes a method for using directed graphs to a represent behavior of a *group* of connections. This seems be useful because there are normal connections which look like malware ones — but using groups of connections might help to differ their behavior by searching for patterns and sequences in them.

Jha et al. [7] also use behavior-based detection of malware, but they build graphs differently — their nodes represent system calls and edges are dependencies between operations. They label particularly significant files, directories, registry key, and devices. This allows them to recognize even otherwise benign behaviors. From this representation they extract k significant subgraphs using a behavior mining technique. These mined behaviors are then synthesized in a discriminative specification.

In [8] there are studied graph-based data, too. More specifically, there is studied the possibility to find anomalies (unusual occurences) in a graph. The authors present two different methods, both of them using the Subdue system (at its core designed to find repetitive patterns in graphs).

Sommer and Paxson [9] argue that applying machine learning on intrusion detection is significantly harder than in other areas, such as optical recognition systems, product recommendations systems or spam detectors. According to the authors of the paper it is mainly because machine learning algorithms excel at finding similarities than in finding not belonging activities.

Jusko and Rehak [10] build graphs where nodes represent tuples (IP, port). Their goal is to find the whole P2P[1] network from one known host. The node (IP, port) representation helps them in this as it does not matter if a host is in multiple P2P networks because only IPs are the same but different ports are used in different networks.

In [11] authors analyse the behavior of a Zbot family botnet. In a capture, that took 57 days to make, they studied individual botnet channels (UDP, TCP, HTTP). Among others, they studied statistical features of traffic which was aggregated by the source IP, destination IP and destination port, ignoring source ports. They were able to identify which actions were related to individual channels. From the experiments we can also see that most malware generate more connections that just one.

Tsatsaronis et al. [12] applied Power Graph Analysis methodology for the first time to the field of community mining to the problem of community detection in complex networks. The Power Graph Analysis method was presented in bioinformatics research before. They proved to be succesful and also the Power Graph Analysis compressed their graphs (number of edges) by up to 60%.

[1] Peer-to-Peer

Chapter 3
Graph Analysis

As presented in [6] it is possible to improve detection of an infected computer by studying groups of connections[1] rather than just one connection at a time. We liked the idea of using a directed graph to model a network communication. The natural (and maybe usual) way in the graph representation of the network is to create a vertice for every IP address or for every pair $(IPaddress, port)$ and having an edge between a pair of such nodes if one if these two nodes communicated with the other one. However, we define our graph in a slightly different way than it can be usually seen, partly because of a different vertice definition and partly because we do not inspect the whole network communication, but only a part of it.

3.1 Our Graph

We wanted to inspect only a communication of one host, namely its leaving communication. Our graph represents only the behavior of the host — the sequence of its connections.

We define our graph as an ordered triplet $G=(V, E, v_0)$ where V is a set of vertices (or nodes), E is a set of edges and v_0 is the first vertice in the recorded communication. It is a directed cycle graph.

Our nodes are defined as triplets *(IP address — port — protocol)*. Both IP address and port are the essential features of a node. We feel that used internet protocol is also a very important feature as it may be one of the cognitive marks of a service[2]. In vertices there are saved information about their size (number of flows[3] with IP address of the inspected host as the source IP address and the destination IP address, destination port, and the protocol of this node) and a set of edges that come from this node.

There is an edge from a node i to a node j if the connection to node j was the first connection of the inspected host after the connection to i. Edges save information about the node to which they head and also the size of themselves (number of flows in the connection).

Apart from vertices and edges, there is another structure that we use and study — cycles. As we believe that the tries for communication from a bot with its C&C masters happen in sequential order but we also believe that these sequences should happen more times in a row — hence using cycles.

The real order of flows is important in our search for cycles, because in this case we do not want to find cycles in a graph — the found cycles could contain nodes that are

[1] From [6](page 1): *"A connection is defined as all the flows that share the following key: source IP address, destination IP address, destination port and protocol. All the flows matching the same key are grouped together. This way of grouping flows allows the analysis to focus on the behavior of a user against a specific service."*

[2] Protocol UDP and port 53 are used by DNS service, or TCP and port 80 are used by HTTP protocol, for example.

[3] We use bidirectional NetFlows (BiNetFlows).

not really related to the other nodes inside the found cycle and it would be also quite difficult to say how many times we observed the cycle. That is why we do not use the graph representation itself for the cycle search. Instead, we use a list of nodes which are in the order they were connected to by the inspected host. Self-loops[1] are not counted as a cycle and when part of a longer cycle, self-looping node is in cycle counted only once[2].

3.2 Visualization

To see the graph we produced and observe its properties we created a web graph visualization tool (see Section 4.3 — Visualization Tool). It helped us to see and prove to ourselves that some malware indeed makes connections sequentially (Figure 3.1). There can be sometimes seen remarkable differences between normal traffic and malicious traffic.

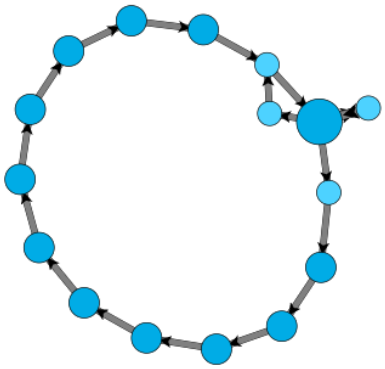


Figure 3.1. This an example of a suspicous graph structure. A computer connected to these nodes in this sequence — even though there could be some nodes between them that were not interesting for us and were removed using a filter (see Section 3.3 — Filtering of Needless Nodes).

3.3 Filtering of Needless Nodes (F1 filter)

As our plan is to detect repeating structures and there were lots of nodes that appeared only once, we decided to remove those nodes. Not only they were needless for our analysis but they also hid some interesting formations. We call this F1 filter.

After a graph is built, we go over all nodes and if their size (number of flows) is 1, they are removed from the graph. The nodes that led to the removed nodes are connected to the nodes that followed the removed ones.

This approach appears to be helpful because it really removed lots needless nodes (Figure 3.2), however, in very big graphs the changes were not so obvious in our visualization tool (Figure 3.3) and the graphs still look too overcrowded. Node compression is usually at least 10 percent, the average compression (from inspected captures) was 54%.

[1] Self-loop is an edge leading from a node i back to i

[2] Example: The list of nodes A-B-B-C-A would have one cycle — A-B-C of length 3, 2 Bs in row would not be counted. On the opposite, A-B-C-B-A would have 2 cycles — first is the whole list (with both Bs) of length 4 and the other is B-C of length 2.

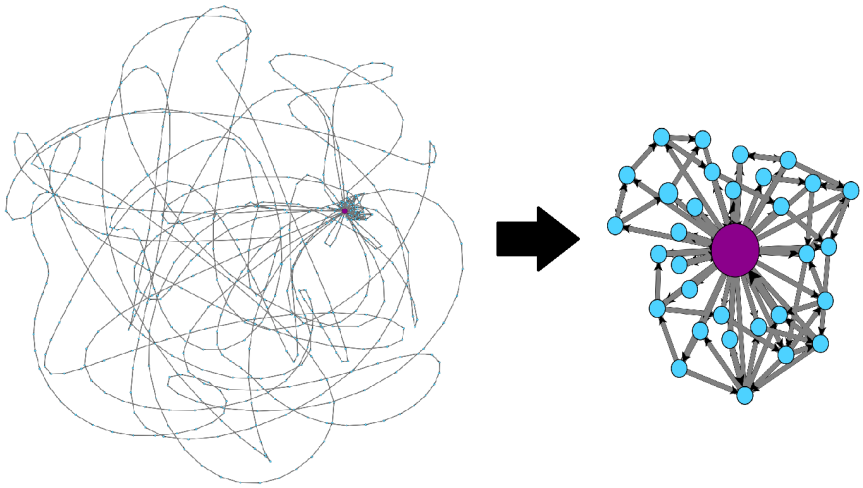


Figure 3.2. An example of how filtering nodes with size 1 can make the graph clearer and linkage between nodes more obvious. An unfiltered graph on the left (zoomed out) and filtered one on the right. 32 nodes were left out of 670.

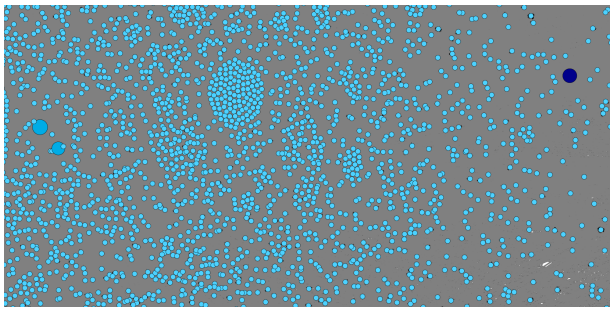


Figure 3.3. Even after removing nodes with size 1 (in this case this lead to removing 15892 out of 24633) there are too many nodes and edges which lead to an unclear graph.

Label	Total number of nodes	Number of nodes after filtering	Compression Ratio(%)
Infected	26730	16	99.94
Infected	24633	8741	64.52
Infected	4293	1871	56.42
Normal	3128	293	90.63
Normal	1072	202	81.16
Infected	620	30	95.16
Infected	500	325	35
Infected	244	1	99.59
Normal	203	150	26.11

Table 3.1. Examples of compression after filtering of needless nodes.

3.4 Inspected Features

In this work, our objective was to search for such features in graphs that would help us differentiate between normal traffic and malicious traffic. Some of our results coincide with our expectectations (graphs of infected computers have higher node with bigger size)

It is worth noting that in the rest of the work (if not said differently) we use graphs after we use F1. 107 hosts were inspected (see Section 5.1 — Data)

Here we present a list of features we extract from our graph at the moment.

The first feature is the total number of vertices in a graph. It is true that such feature may seem to be quite dependent on the size of the inspected capture, but such assumption is not quite right as we can see from the Table 3.2. We observed that infected hosts had more vertices in their graph than normal ones in general (Table 3.3).

Label	Number of flows	Size of file in MB	Number of nodes in graph
Infected	186255	38	723
Infected	241242	160	24633
Infected	3050877	395	795
Normal	9797	2.25	1050
Normal	7769	0.8	1072
Normal	2398	0.2	203

Table 3.2. Examples of counts of vertices in graphs compared to original numbers of flows and sizes of files the flows were stored in.

Another feature is, already mentioned, filtering of nodes with size 1 — or more precisely said, the number of nodes that were not filtered out. Even though average ratios of compression on our data were very similar to each other for both normal and infected graphs (slightly over 50 percent), the median was quite different. For the hosts labeled as normal it was 50 percent but for infected hosts, it was 70.5 percent.

Label	Avg. NS	Med. of NS	Avg. F1 compr. (%)	Med. of F1 compr. (%)
Infected	2530	670	53.3	70.5
Normal	202	48	54.7	50
All	1007	156	54.3	62.47

Table 3.3. Average (avg.) and median (med.) of node sizes (NS) and of F1 filter node compression ratios calculated from our data. It can be seen that infected hosts have much higher both average and median of their node sizes (how many times the inspected host made a contact with them) and higher median of F1 compression ratio.

The next feature is also related to nodes. We expect infected files to have more nodes with bigger size than normal ones — we set thresholds for node size and observe how many and which nodes remained above this threshold. In Table 3.4 we can see that there is a lot more nodes with their sizes above thresholds in graphs labeled as infected but due to smaller total counts in normal graphs ratios of the nodes above thresholds are higher in them.

The last feature we currently extract from nodes is how much they self-loop. We count a node as self-looping if there are two flows with this node as the destination node[1] (if there were some flows between these two flows and the nodes created from them were removed by the F1 filter, it is also counted as a self—looping). We thought that infected hosts would tend to create more self-looping nodes than normal users but from our data it seems it's the opposite — normal data have more self-looping nodes.(Table 3.5).

[1] In the graph: A-A-B-A-B-A-A there is one self-looping node, A, and it has 2 self-loops.

Threshold	# inf.	% inf.	# norm.	% norm.	# all	% all
1	244	100	19.5	100	66	100
3	32	54	11.5	77.4	17	72.2
4	25	34.9	7	64.4	13	55.6
5	23	25.8	6.5	50.4	9	33.3
6	21	21	5.5	46.3	8	33
10	12	11.7	3	25.1	3	15.4
15	6	8.8	3	15.5	3	9.6
20	4	5	2	9.3	2	6.4
30	2	2.7	1.5	4	2	3.3

Table 3.4. Examples of medians of counts (#) and ratios (%) of nodes with size above thresholds. It can be seen that even though graphs of normal traffic have much fewer nodes, their ratios (node with size above threshold to total number of nodes) are a lot higher than ratios of infected graphs.

Threshold	# inf.	% inf.	# norm.	% norm.	# all	% all
1	13	5.7	11.5	66.7	13	50
2	4	2.5	5	50	4	37
3	3	1.7	4.5	39.8	3	27.4
4	2	0.9	3	34.5	3	12.9
5	2	0.8	2.5	27.8	2	10.1
6	2	0.8	1.5	18.4	2	7.5
10	1	0.4	1	11.3	1	4.4
20	1	0.2	1	4.4	1	2.1

Table 3.5. Examples of medians of counts (#) and ratios (%) of self-looping nodes from the total number of nodes for different self-looping thresholds calculated from our data. It can be seen that normal hosts (norm.) have higher ratio of self-looping nodes than infected ones (inf.).

We also study edges. The extracted features from edges is quite similar to the ones extracted from vertices — again it is total number of edges.

And again, we set a threshold and observe how many of edges are above it as for edges it is also expected that they would be bigger (had more flows inside them) in a graph of an infected host. In Table 3.6 it can be seen that this is true for absolute counts of edges but not for ratios of edges above threshold to total count of edges, normal traffic in the graphs from our data has a lot fewer edges than malicious traffic in total.

The last structures we currently create in our graph are cycles. We study how many cycles are present in total and how many cycles are of specific lengths. We expect normal users to have less cycles in their communication.

We thought it might be interesting to find the most used port in every graph and compare the ratio of the number of occurences of this port to the total number of flows. Ratios for normal and infected graphs slightly differ, they are higher by a bit in normal graphs (Table 3.7).

The last feature in this list is the most used protocol in a graph. The ratio of the most freqented protocol in a graph to total number of flows seems to be higher in malicious traffic (Table 3.8).

Threshold	# inf.	% inf.	# norm.	% norm.	# all	% all
1	490	100	44	100	175	100
2	89	26.4	11	46.3	24	32.4
3	49	11	5.5	21.3	6	14.7
4	29	6.28	3.5	10.5	4	9.3
5	15	3.247	3	6.5	3	5.5
10	4	0.9	0.5	0.2	1	0.4
15	0	0	0	0	0	0

Table 3.6. Examples of medians of ratios (%) and counts (#) of edges above thresholds of edge size. Setting threshold to 1 means that all edges are taken in count.

Label	Average (%)	Median (%)
Infected	51.5	61.3
Normal	66.8	67
All	61.5	65.8

Table 3.7. Averages and medians of ratios the most frequented ports. It can be seen that they are quite similar for both labels, normal graphs have them slightly higher.

Label	Average (%)	Median (%)
Infected	80	91.4
Normal	71.4	70
All	74.4	73.6

Table 3.8. Averages and medians of ratios the most frequented protocols. It can be seen that the graphs of infected hosts have the ratio higher than normal graphs.

Chapter 4
Features Extraction Tool

As we wanted to inspect the features mentiones in the previous chapter we needed a tool that would build and extract those features for us.

It was decided that we would create this tool in Python, because this programmimng language allows better speed of developmnet to a programmer while it is fast enough for our purposes.

The tool is available from the GitHub repository of our project[1].

4.1 Compatibility

This tool was written in version 2.7 of Python[2]. It is recommended to run this tool with Python2.7. Compatibility with other versions is not guaranteed. Tested on Windows (10) and Linux.

This tool does not use any third party libraries.

4.2 Workflow of our Tool

In this section we describe how our tool proceeds when a user runs it.

Firstly, the tool processes user's arguments. Users have to specify a file with data and the IP address of the host they want to inspect.

In addition to the mandatory arguments there is a number of optional ones — these arguments include options for setting F1 filter, specific protocols or ports filter, thresholds or arguments directing output of the tool.

Then the tool reads data from BiNetFlow[3] file. It only uses such lines (flows) of the file where the source IP address is the IP address specified by user's argument. If the user used protocol or port filter, it is used now.

During the reading of data, a graph is built. Its nodes and edges are counted before F1 filter is used (if user used this option). They are counted again after F1 to compare how many nodes (edges) were removed.

Then counts of nodes, edges and self-looping nodes which meet the condition of having the sizes larger than or equal to given thresholds are calculated.

In the end there is a cycle search. User specifies range of cycle lenghts he wishes to search for. There is also a threshold for cycles to be set — minimal size of cycles (number of repetitions in the flow list).

[1] `https://github.com/dansmoliik/Malware-graph-detection`

[2] `https://www.python.org/`

[3] NetFlows are records of network communication which aggregate data using source and destination IP addresses and ports and protocol number. Another information contained in flows are its duration, numbers of bytes and packets and its start time. NetFlows are only unidirectional — flows from a client to a server is one flow and a response from the server would be another. Since this sounds quite inefficient, BiNetFlow were introduces. They are very similar, except that a single flow represents communication in both ways.

All that remains after this is to print to the consolesave to file features found by the tool. This output can be either just an overall output which consists of counts of nodes above threshold, count of cycles, etc. or it can be a longer version which prints also all edges — with their size, source and destination, for example.

It is also possible to save the built graph in a JSON file which is used by our simple web visualization tool.

4.3 Tool Input

In this section we list all possible arguments to our tool and explain them individually.

4.3.1 Data

Bidirectional flows are expected by the tool.

Only one flow is on a line and data is flows are separated by commas.

4.3.2 Mandatory Arguments

There are 2 mandatory arguments:

`-p` (or `--path`) specifies path to a BiNetFlow file (a file with extension .binetflow).

`-ip` sets the IP address of the host we want to inspect. Only flows with this IP address as source IP address are used.

4.3.3 Optional Arguments — Graph

These arguments are optional — the tool works without them but they are used to select some attributes of features a user wants. If value by a user of a threshold is lower than the default for that specific threshold, that feature is not calculated.

`-h` (or `--help`) displays help for the tool and exit the program. Usage of all option is show there.

`-f1` is used to set whether to use F1 filter (see Section 3.3 — Filtering of Needless Nodes). This is also used to calculate F1 compression ratio

$CountOfRemovedNodes/TotalCountOfNodes$.

`-f` can be used to filter specific ports, protocols or port-protocol pairs.

`-nt` sets the threshold for node size. This option is used to calculate how many nodes have their size higher than or equal to the given threshold. This also produces the ratio

$CountOfNodesAboveThreshold/TotalCountOfNodes$. Default threshold is 0 (counts every node).

`-slt` is used to set the threshold for self-looping of nodes. Our tool calculates the count of nodes that have their self-loop count higher than or equal to the given threshold. Count of self-loops is, in other words, the count of edges that lead from node i to node i. With threshold 0 (default) the tool would count all nodes, even the ones without any self-loop.

`-et` is an option that sets the threshold for edges, counts edges which have their size bigger than or equal to the given threshold and calculates ratio

$CountOfEdgesAboveThreshold/TotalCountOfEdges$. Default is 0.

`-ct` sets the minimal number of repetitions of a cycle. Cycles below this threshold are not counted. Default is 0.

`-clen` is used to specify lengths of cycles we want to search for. It can be either one number or a range of numbers given by 2 numbers. Cycles are then being searched for in this range, one length at a time. Default value is 2.

4.3.4 Optional Arguments — Output

These optional arguments are used to control output of the tool.

`--less` reduces the amount of information printed to the console. With this option only overall information (total counts, ratios, etc.) are written to the standard output. Otherwise, more information (nodes above threshold, found cycles, etc.) appears in the console. By default, everything is written to the console.

`--noout` means a user does not want to print anything to the console — just to save data in files. By default, everything is written to the console.

`--nooutf` turns off saving found features in files. By default, everything is saved to files.

`-csv` turns off normal output to the console and CSV format is used instead. There are two lines printed: in the first line there are names of features and in the second line there are values of the features or nothing if the values were not calculated.

`--label` is used only in CSV format. It specifies label (usually Normal or Infected) for the current graph. Default label is "Unknown".

`--gjson` tells the tool that it the graph should be saved in a JSON file in the end.

4.3.5 Arguments Concerning Graph Visualization

There are some arguments which can be used to remove nodes and edges which are not above threshold from JSON graph representation.

`--ntg` tells the tool not to save nodes which have size less than NT

`--sltg` tells the tool not to save nodes which have self-looping less than SLT

`--etg` tells the tool not to save edges which have size less than ET

4.3.6 Usage

The simplest example of using our tool would be

```
python main.py -p <file_name>.binetflow -ip <ip_address>
```

This would result in building a graph of leaving network communication of given IP address from given BiNetFlow file. Every threshold would be default so every node, edge and cycle (of length 2) would be counted regardless their sizes or anything else. Default output would be displayed in the console and everything would be also saved in files. JSON file would not be created.

`python main.py -h` can be used to display help and description of this program.

4.4 Tool Output

Our tool has multiple forms of output (as mentioned in the last section).

4.4.1 Detailed Information About Graph

By default, our tool displays all extracted features in the console and also saves everything in files.

If standard output is not forbidden and a user did not use option for less output, there are displayed overall information about the graph together with lists of nodes, edges, and cycles which meet the conditions. If the user used option `--less`, only overall information are displayed.

If not forbidden by a user, in the base directory of the project there are created a folder `output/data/<BiNetFlow_file_name>/<Source_ip_address>` and 5 files (or

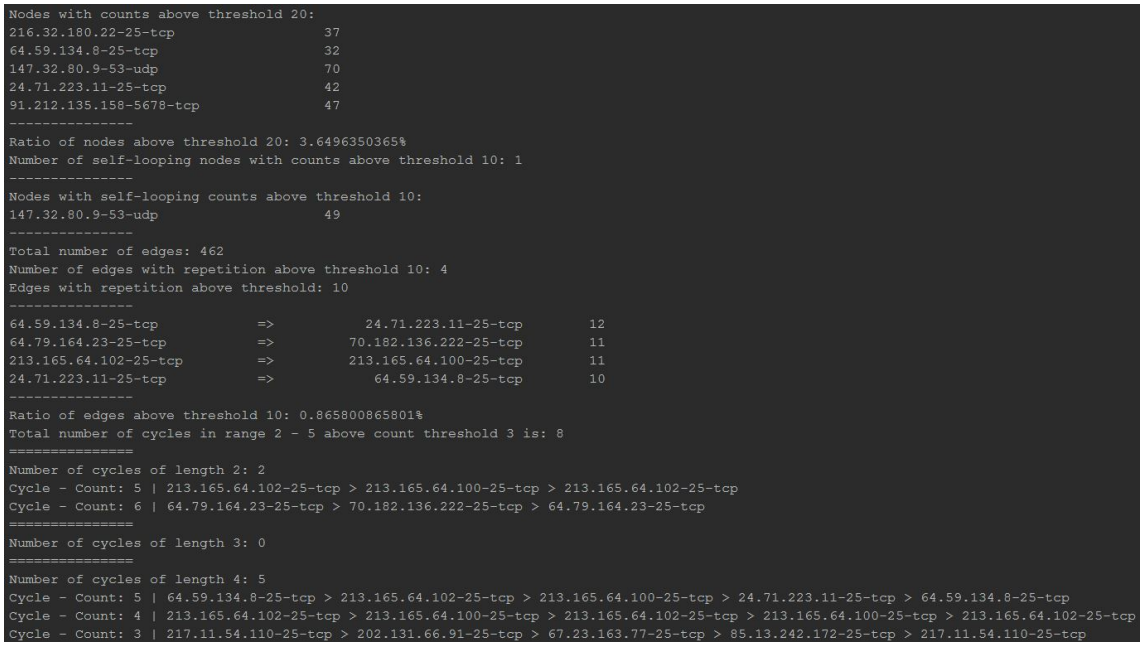


Figure 4.1. A part of an output in a console. We cam see nodes above threshold with their names and sizes, edges with their source and destination nodes and sizes and among some other things there are cycles of different lengths.

```
Total number of nodes without "f1" filter: 723
Total number of edges without "f1" filter: 4254
Number of nodes after "f1" filter: 586
Node compression: 18.948824343%
Most frequent port: 443
Ratio of most frequent port: 86.8497436063%
Most frequent protocol: tcp
Ratio of most frequent protocol: 93.4208404554%
Number of nodes with counts above threshold 10: 359
Ratio of nodes above threshold 20: 61.2627986348%
Number of self-looping nodes with counts above threshold 10: 4
Total number of edges: 4006
Number of edges with repetition above threshold 10: 846
Ratio of edges above threshold 10: 21.1183225162%
Total number of cycles in range 2 - 5 above count threshold 3 is: 13
```

Figure 4.2. An example of an overall output.

overwritten if they already existed) inside this folder. One file contains overall information. The other four files contain lists of nodes that have their sizes and auto-looping above the threshold, edges with sizes above the threshold and cycles of different lengths that have counts of repetition above the threshold.

4.4.2 CSV

If a user picked the `-csv` option, default output is not displayed. Instead, two lines are printed. The first line is a header line — it contains names of features printed on the second line. Both lines have the same number of columns. These columns are separated by commas (Comma Separated Values).

This output can be useful when processing more files and hosts at one time to create a bigger table with data from multiple sources.

The CSV output is friendly to various analyst tools (Microsoft Excel, OpenOffice Calc, RapidMiner, etc.).

4.4.3 JSON File

If the option `--gjson` is used, the built graph is saved in a JSON file. This file can be used in the web visualization tool. It contains information about all nodes and edges.

For nodes there are the name of the node, its color and its size.

Edges have information about the source node, the target node, size and color.

The meanings of colors and size are explained in Subsection 4.6.2.

4.5 Processing Multiple Hosts

Because it might be needed to run the tool on multiple hosts for more data and better analysis, we created a simple Python script that can be used to run our tool on a list of hosts and a list of arguments. It takes 3 arguments.

The first one is a file containing a list of BiNetFlow files and for every file there are three lists of IP addresses which we want to examine in this file — normal host, infected hosts and not labeled hosts. This file has to have extension `.filelist`. Each line has to follow this format:

`<path_to_binetflow_file>|<normal>|<infected>|<not_labeled>`

The second argument is a file containing a list of arguments which are written in the same format as if the classic tool is run. On each line is one set of arguments — only optional arguments should be used because mandatory ones are taken from the File List. CSV argument is added automatically to each set of arguments.

The last argument is the name of the file a user wants to save data in. It should be without any extension because the script will save it in a file ending with `.csv` anyway.

The script can be run as:

```
python group_features_extraction.py
<path_to_filelist_file> <path_to_arglist_file> <out_path>
```

The resulting CSV file contains the usual CSV output of the tool — the first line is a header line — there are names of the features. The rest of lines are values of of the features of individual hosts.

4.6 Visualization Tool

As mentioned already (see Section 3.3 — Filtering of Needless Nodes), we needed a simple tool that would allows us to visualize graphs from our tool.

As we wanted to have the whole tool eventually online, it was decided to use HTML, CSS and Java Script as technologies.

4.6.1 About

We used *Forced-Directed Graph*[1] by Mike Bostock. It is a tool for vizualization of directed graphs which uses D3.js library[2]. Some changes were made to fit the tool to our needs — e.g. colors and sizes of nodes and edges and their meanings.

[1] `https://bl.ocks.org/mbostock/4062045`
[2] `https://d3js.org/`

4.6.2 Reading Graph

The size of a node in the visualization corresponds to the actual node size from the feature extraction tool, its color corresponds to its self-looping — the bigger node in a web browser, the greater count and the darker node, the higher self-looping count.

The same goes for edges — bigger edges mean the bigger count of flows inside the edge.

Also, if a user searches for cycles, then edges which are a part of any cycles get red color.

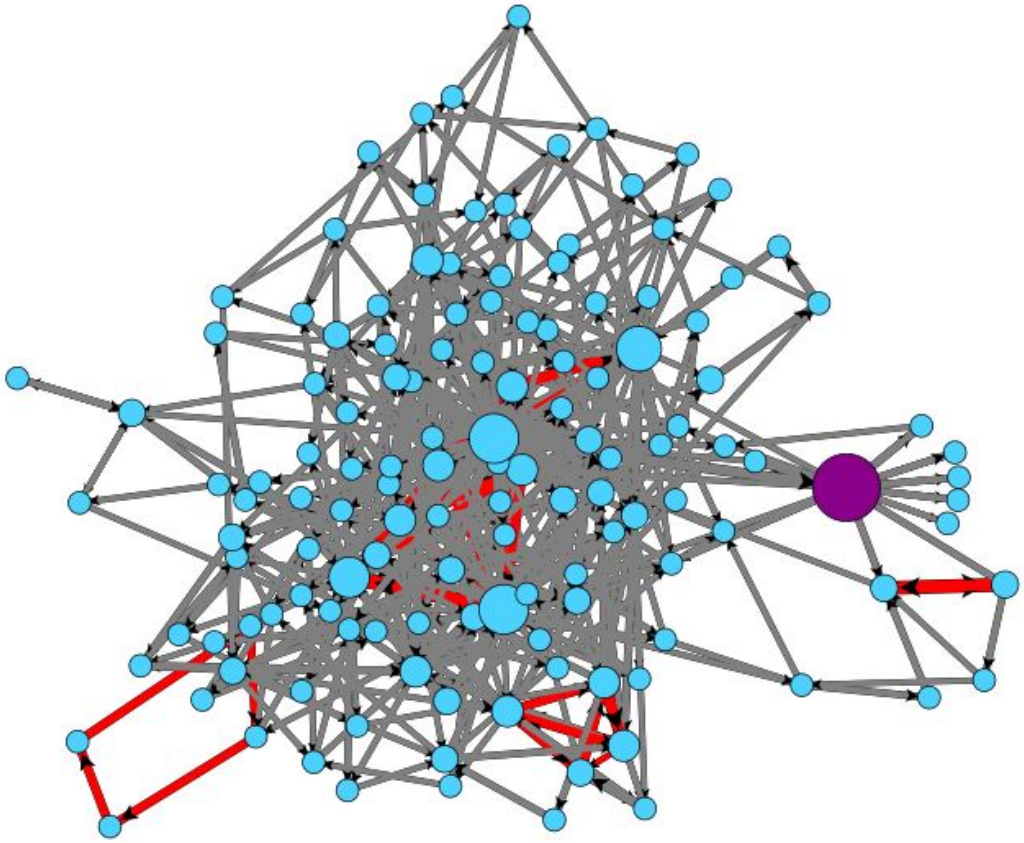


Figure 4.3. This is a graph of an infected host. We can see that theere are many edges and some of them are parts of a cycle (red ones).

We were not sure which fixed values should be upper limits for sizes and colors for the graph to look good, so we went with finding the maximum values from the built graph. We look for the largest sizes of nodes and edges and for the maximum self-looping count. The nodes (edges) with these values are the biggest (darkest) and the other nodes have their sizes smaller relatively to the maximums.

4.6.3 Usage

JSON file with the graph information needs to be in the same folder as index.html.

The tool uses AJAX calls and because of that it needs running server under it. How to start a simple server in Python is explained in readme.txt.

4.7 Possible Future Work

The whole graph analysis aside, there is a lot to improve just on the tools we created.

The most important thing would be finding new features that can be extracted and implement the search for them. Specifically it would be worth to study the nodes, edges and cycles which pass the threshold — not just their counts but other features like numbers of incoming and leaving edges of each node.

Another possible features, we could make use of durations and start times of flows. We expect that malicious communication would be more periodical.

Even though the speed of extracting features from one host in not bad, for larger graphs and for multiple files and argument processing, it is a pain (Table 4.1). For example, getting information about cycles from all our hosts took a whole night. Building the graph and filtering it with F1 takes the most of the time — it would be worth to allow users to specify more than one set of arguments and run these arguments on the already built graph.

Total node count	Creating graph	F1	Setting thresholds	Cycle search (range 2-5)
26730	13.9	15.6	15.7	16
24633	8	43	44.5	52.7
809	2.8	2.9	2.9	2.9
202	3	3	3	3.3
257	7	7.2	7.2	7.5
160	14.7	15.5	16.5	18.7

Table 4.1. Examples of the speed performance of our tool in seconds. It can be seen that there i

Also there are some unnecesary calculations or creating of unused strings (because of using CSV output, for example).

The vizualization tool is sufficient for small graphs or for graphs with a little number of edges. Large graphs can become one grey blob with blue dots, even if F1 is used (Figure 3.3, for example).

4.8 Recapitulation

We created this tool to inspect leaving communication of a host saved in flows and extract various features from this graph. We used it to study our newly defined nodes and edges. From this point of view, it does its job — it computes some basic features of graphs and displays them in readeble format.

On the other hand there is still much to do. Features we use are quite basic. The performance of the tool could be also improved.

Chapter 5
Experiments

In this work we used the leaving network communication of one host to build a graph, extract features from this graph and evaluate them — at first by looking at the features and comparing or seeing the graphs in our tool.

But our ultimate goal was to use these features to create a classificator that alone would later be deciding what is normal communication and what might be malicious. To achieve this objective we used our tool to extract features from more hosts and the data we got were used in Random Forest learning method.

5.1 Data

The important part of every experiment is to have enough of good data. In this section the data and their source are desribed.

5.1.1 Dataset

The most of the data, which we use in our research and testing, comes from the *CTU-13-Dataset* [13]. It was created in the CTU in Prague in 2011. We can find thirteen different malware captures (scenarios) inside. Each of the captures include Botnet (infected hosts), Normal (verified users) and Background traffic (this traffic is not verifed and can contain anything). There are PCAP files, BiArgus and BiNetFlow files. We are interested in the last ones.

In the Figure 5.1 we can see characteristics of each capture in the dataset.

Table 2 – Characteristics of the botnet scenarios. (CF: ClickFraud, PS: Port Scan, FF: FastFlux, US: Compiled and controlled by us.)

Id	IRC	SPAM	CF	PS	DDoS	FF	P2P	US	HTTP	Note
1	√	√	√							
2	√	√	√							
3	√			√				√		
4	√				√			√		UDP and ICMP DDoS.
5		√		√					√	Scan web proxies.
6				√						Proprietary C&C. RDP.
7									√	Chinese hosts.
8				√						Proprietary C&C. Net-BIOS, STUN.
9	√	√	√	√						
10	√				√			√		UDP DDoS.
11	√				√			√		ICMP DDoS.
12							√			Synchronization.
13		√		√					√	Captcha. Web mail.

Figure 5.1. This table show characteristics of the captures from CTU-13-Dataset. Image taken from [14].

These captures were taken for different durations — these can be seen in the Figure 5.2 together with information about the communication of which bots was being inspected and how many NetFlows were captured.

Id	Duration(hrs)	# Packets	#NetFlows	Size	Bot	#Bots
1	6.15	71,971,482	2,824,637	52GB	Neris	1
2	4.21	71,851,300	1,808,123	60GB	Neris	1
3	66.85	167,730,395	4,710,639	121GB	Rbot	1
4	4.21	62,089,135	1,121,077	53GB	Rbot	1
5	11.63	4,481,167	129,833	37.6GB	Virut	1
6	2.18	38,764,357	558,920	30GB	Menti	1
7	0.38	7,467,139	114,078	5.8GB	Sogou	1
8	19.5	155,207,799	2,954,231	123GB	Murlo	1
9	5.18	115,415,321	2,753,885	94GB	Neris	10
10	4.75	90,389,782	1,309,792	73GB	Rbot	10
11	0.26	6,337,202	107,252	5.2GB	Rbot	3
12	1.21	13,212,268	325,472	8.3GB	NSIS.ay	3
13	16.36	50,888,256	1,925,150	34GB	Virut	1

Figure 5.2. Amounts of data taken in each capture with names of bots being tracked. Image taken from [15]

Scen.	Total Flows	Botnet Flows	Normal Flows	C&C Flows	Background Flows
1	2,824,636	39,933(1.41%)	30,387(1.07%)	1,026(0.03%)	2,753,290(97.47%)
2	1,808,122	18,839(1.04%)	9,120(0.5%)	2,102(0.11%)	1,778,061(98.33%)
3	4,710,638	26,759(0.56%)	116,887(2.48%)	63(0.001%)	4,566,929(96.94%)
4	1,121,076	1,719(0.15%)	25,268(2.25%)	49(0.004%)	1,094,040(97.58%)
5	129,832	695(0.53%)	4,679(3.6%)	206(1.15%)	124,252(95.7%)
6	558,919	4,431(0.79%)	7,494(1.34%)	199(0.03%)	546,795(97.83%)
7	114,077	37(0.03%)	1,677(1.47%)	26(0.02%)	112,337(98.47%)
8	2,954,230	5,052(0.17%)	72,822(2.46%)	1,074(2.4%)	2,875,282(97.32%)
9	2,753,884	179,880(6.5%)	43,340(1.57%)	5,099(0.18%)	2,525,565(91.7%)
10	1,309,791	106,315(8.11%)	15,847(1.2%)	37(0.002%)	1,187,592(90.67%)
11	107,251	8,161(7.6%)	2,718(2.53%)	3(0.002%)	96,369(89.85%)
12	325,471	2,143(0.65%)	7,628(2.34%)	25(0.007%)	315,675(96.99%)
13	1,925,149	38,791(2.01%)	31,939(1.65%)	1,202(0.06%)	1,853,217(96.26%)

Figure 5.3. Distributions of labels in CTU-13-Dataset. Scenarios were manually analyzed and labeled. Image taken from [16]

Together with this dataset we used *CTU-Malware-Capture-Botnet-25-1*, *CTU-Malware-Capture-Botnet-31*, *CTU-Malware-Capture-Botnet-35-1* and *CTU-Normal-5*, *CTU-Normal-6*, *CTU-Normal-7*.

From these captures we built graphs of 107 hosts in total.

5.1.2 Data Format

The data we used in our feature extraction tool were BiNetFlows produced by Argus[1] and stored in `.binetflow` files.

Our tool processed these files and for every specified host generated a graph and calculated features of these graphs. These results were saved as one table in a `.csv` file which is friendly format for data mining applications.

5.1.3 CSV file

For every studied host (there are 107 of them in total), we ran our tool multiple times to obtain data for different combinations of threshold. However, we were slowed by our hardware limitations and also by software engineering misconception in our tool. This resulted in getting data just for some combination of threshold in the range from 1 to 30.

[1] `https://qosient.com/argus/`

The original CSV file contained 3317 lines of our data but, due to the nature of the loop implementation in RapidMiner, we had to add 2 lines of mock data for every combination of threshold we didn't have real data for. Without such measure, the process in RapidMiner would crash because there would not be any examples for some combination (Random forest would not get any input). Even though we added only 2 lines for every combination that did not have any data and these 2 lines produce 0% accuracy during learning/testing, the whole process take considerably more time (as there are 30x30x30 possible threshold combinations).

5.2 Data Mining Application

We did not want to "reinvent the wheel" by implementing a learning algorithm or a statistical tool ourselves. Instead we used already implemented one — RapidMiner[1]. It is a visual design environment for rapid building of predictive analytic workflows. We used it also for statistical data showed in tables in Chapter 2.

This tool was used thanks to our positive experience and also because, according to KDnuggets[2], RapidMiner is one of the most popular tools. It is placed on one of the first places every year in an annual poll created by KDnuggets, being used by more than 30% of data scientists (from around 3000 voters)[3].

5.3 Random Forest Algorithm

We decided to to go with *Random forest* learning algorithm presented by Leo Breiman in 2001 [17]. Random forest creates a number of decision trees. Its result is the mode of predictions of the individual trees. Its pros are that this algorithm is quite flexible and should not overfit. On the other hand, the result (a set of decision trees) are not easily comprehensible.

5.4 Experiment Design

The workflow of our experiment in RapidMiner is described in this section.

5.4.1 Used Features

In the end and after some light testing, 9 graph features were chosen to be used for learning: *the total number of nodes*, *total number of edges*, *ratio of F1 compression*, *ratio of nodes with their size above threshold*, *ratio of self-looping nodes above the threshold to the count of all nodes*, *ratio of edges above the threshold*, *ratio of the most frequent port* and *ratio of the most frequent protocol*.

[1] https://rapidminer.com/
[2] http://www.kdnuggets.com/
[3] http://www.kdnuggets.com/2016/06/r-python-top-analytics-data-mining-data-science-software.html, http://www.kdnuggets.com/2015/05/poll-r-rapidminer-python-big-data-spark.html, http://www.kdnuggets.com/2014/06/kdnuggets-annual-software-poll-rapidminer-continues-lead.html

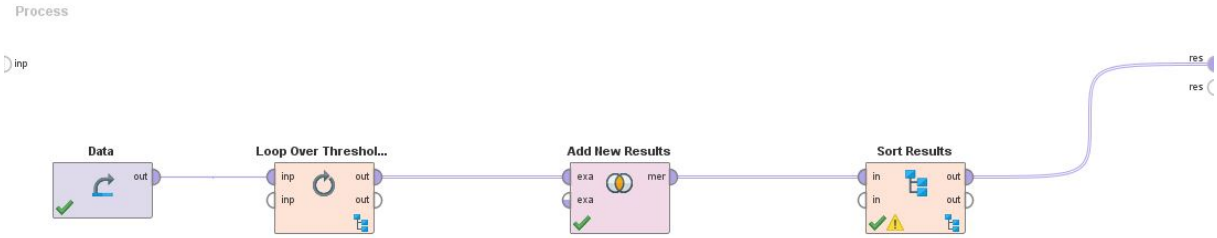


Figure 5.4. The outer part of the RapidMiner experiment. Firstly, data are loaded. Then there are loops over thresholds and the actual learning and classification (Figure 5.5). Resulting performance vectors are appended to the results table after every loop and sorted in the end.

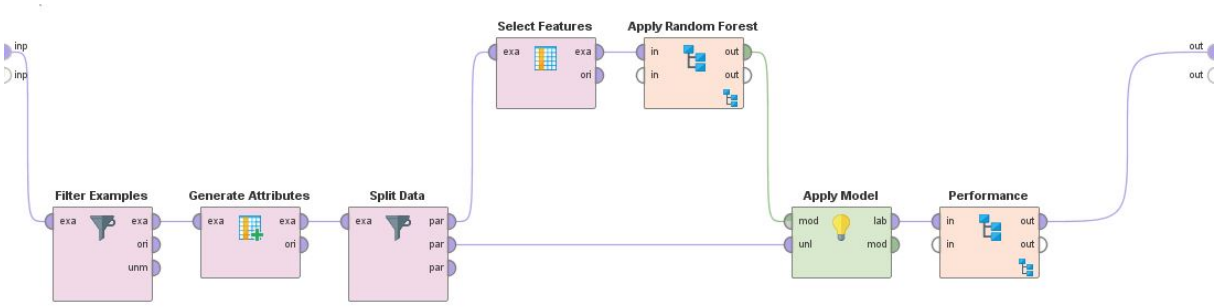


Figure 5.5. The inner part of the RapidMiner experiment. Data that do not belong in the current iteration are filtered out, new features that weren't produced in our tool can be generated (e.g. self-looping ratio). Then the data are split, 66% for learning, 34% for testing. Only some features are selected for the actual learning part using Random Forest algorithm. The created model is then used to calculate performance. The performance vector is then appended to the results table.

5.4.2 RapidMiner Experiment Design

There are 3 nested loops (one for every threshold we use). Inside the loops we split data (107 lines of comma separated values) in ratio 2:1 (66% for learning and 34% for testing) using a shuffled sampling.

Features are selected for the learning process and after that Random forest comes on the stage to create a classificator. We used Random forest with 10 decision trees.

The classificator is tested on the one third of our data previously made and accuracy, sensitivity and fallout are calculated.

Results for every iteration over thresholds are saved and the result is a table with all threshold combinations and info about their performance.

5.5 Results

As mentioned, the classifier is tested on one third of all data (36 examples) — approximately one third from this amount are graphs of the traffic (12) of infected hosts and the

rest (24) are normal graphs. These values are approximate because of the randomness of the shuffled sampling.

Using this described method we were able to get up to 97% accuracy (Table 5.1), however, the values of sensitivity (true positive rate) which indicate how many percent of infected host were recognized are not so great (Figure 5.6).

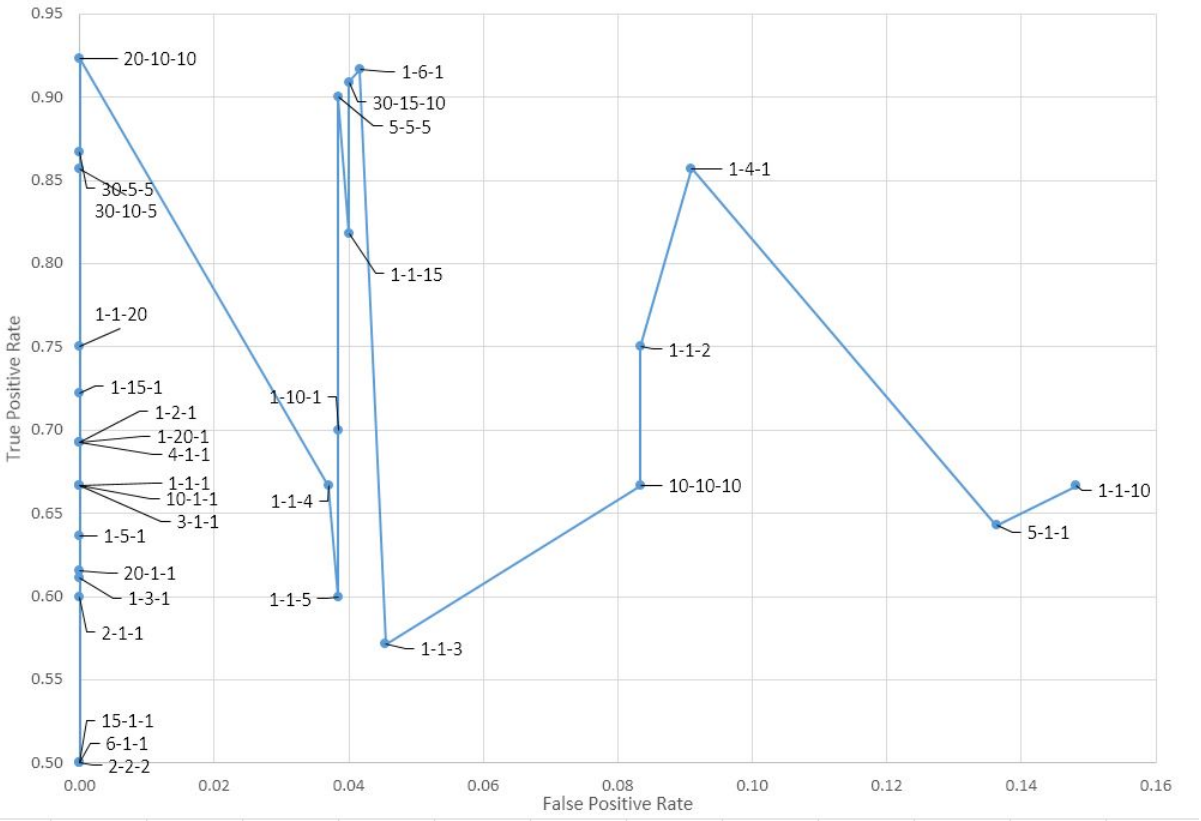


Figure 5.6. ROC curve illustrating the diagnostic ability of the Random Forest classifier for different thresholds. Labels of points are triplet *(Node size threshold - Self-looping threshold - Edge size threshold).*

5.6 Results Analysis

From the results table (Table 5.1) it can be seen that the classifier had quite high accuracy, but its sensitivity was around 90% percent or worse — we have to understand that these results were obtained from a very small testing set (35–37 examples) and the actual number of malicious data was 11–13, therefore 1 undetected infected host corresponds with around 10% missing to TPR from 100%.

Anyway, our graph representation and its analyses are an interesting path in the field of malware detection. We think that with a larger dataset the results might be even better.

NT	SLT	ET	Accuracy (%)	TPR (%)	FPR (%)
20	10	10	97.2	92.3	0
1	6	1	94.4	91.7	4.2
5	5	5	94.4	90	3.8
30	15	10	94.4	90.9	4
30	5	5	94.4	86.7	0
30	10	5	94.4	85.7	0
1	1	15	91.7	81.8	4
1	1	20	91.7	75	0
1	2	1	88.9	69.2	0
1	1	4	88.9	66.7	3.70
1	4	1	88.9	85.7	9.1
1	5	1	88.9	63.63	0
1	10	1	88.9	70	3.8
1	20	1	88.9	69.2	0
2	1	1	88.9	60	0
4	1	1	88.9	69.2	0
10	1	1	88.9	66.7	0
1	1	1	86.1	66.7	0

Table 5.1. Accuracy, true positive rate (TPR) and false positive rate (FPR) for some of thresholds.

Chapter 6
Conclusion

The aim of our project was to use network communication to build a graph and analyse features extracted from this graph. To achieve this goal we created a tool that does that. We presented an unusual way of nodes and edges definitions and discussed some features our graph.

Some statistics of the found features proved our expectations whereas a lot of them were against our beliefs. It means that we should study our data again to understand them better.

Using the Random forest algorithm in RapidMiner tool to create a classifier we were able to achieve 97% accuracy and 92.3% sensitivity. These results seem optimistic and prove that this direction of malware analysis could lead to interesting results. Even more so because we used quite basic graph features as input to Random Forest.

However, an actual deployment of our analysis tool in its current state to the real world problems would be problematic because it requires large amounts of BiNetFlows which are created from even larger amounts of packets. The most of our data were captured for more than two hours and before took tens GB — on the other hand, in these captures were saved packets of multiple hosts, not just one.

There is much that can be done to improve this project. The feature extraction tool would deserve a rework to increase its speed. The graph we build might be compressed before features are searched for to speed up the computation (using Power Graph Analysis [12], for example) There is a space for possible improvements of the classifier — different thresholds, new features, higher number of decision trees in Random forest and learning/testing on more data.

References

[1] Bisson, David. 10 High-Profile Malware Families of 2017. *The State of Security*. May 2017, pp. -.
https://www.tripwire.com/state-of-security/security-data-protection/cyber-security/10-high-profile-malware-families-2017/.

[2] Command and Control in the Fifth Domain. *Command Five*. February 2012, pp. -.
http://www.commandfive.com/papers/C5_APT_C2InTheFifthDomain.pdf.

[3] Sanders, Tom. Botnet operation controlled 1.5m PCs. *V3*. California, October 2005, pp. -.
http://www.v3.co.uk/v3-uk/news/1944019/botnet-operation-controlled-15m-pcs.

[4] Zeidanloo, Hossein Rouhani, and Azizah Bt Abdul Manaf. Botnet Detection by Monitoring Similar Communication Patterns. *International Journal of Computer Science and Information Security*. Kuala Lumpur, 2010, pp. -.

[5] Anderson, Blake. Hiding in Plain Sight: Malware's Use of TLS and Encryption. *Cisco Blogs*. January 2016.
https://blogs.cisco.com/security/malwares-use-of-tls-and-encryption.

[6] Garcia, Sebastian. Detecting the Behavioral Relationships of Malware Connections. *ACM*. 2016, pp. -.

[7] Jha, Somesh, Matthew Fredrikson, Mihai Christodoresu, Reiner Sailer, and Xifeng Yan. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. University of Wisconsin–Madison, 2013.

[8] Noble, Caleb C., and Diane J. Cook. Graph-Based Anomaly Detection. University of Texas at Arlington, 2003.

[9] Sommer, Robin, and Vern Paxson. Outside the ClosedWorld: On UsingMachine Learning For Network Intrusion Detection Robin. Lawrence Berkeley National Laboratory, May 2010.

[10] Jusko, Jan, and Martin Rehak. Identifying Peer-to-Peer Communities in the Network by Connection Graph Analysis. *International Journal Of Network Management*. San Jose, 2014.

[11] García, Sebastián, Vojtěch Uhlíř, and Martin Rehak. Identifying and Modeling Botnet C&C Behaviors. In: *Proceedings of the 1st International Workshop on Agents and CyberSecurity*. New York, NY, USA: ACM, 2014. pp. 1:1–1:8. ACySE '14. ISBN 978-1-4503-2728-2. Available from DOI 10.1145/2602945.2602949.
http://doi.acm.org/10.1145/2602945.2602949.

[12] Tsatsaronis, George etal. Efficient Community Detection Using Power Graph Analysis. Glasgow, October 2011.

[13] Garcia, Sebastian, Martin Grill, Honza Stiborek, and Alejandro Zunino. An empirical comparison of botnet detection methods. *Computers and Security Journal: Elsevier*. 2014, pp. 100-123.
http://www.sciencedirect.com/science/article/pii/S0167404814000923.

[14] Garcia, Sebastian. *Characteristics of botnet scenarios.* http://mcfp.weebly.com/uploads/1/1/2/3/11233160/145022_orig.jpg.

[15] Garcia, Sebastian. *Amount of data on each botnet scenario.* http://mcfp.weebly.com/uploads/1/1/2/3/11233160/6977136.jpg?626.

[16] Garcia, Sebastian. *Distribution of labels in the NetFlows for each scenario in the dataset.* http://mcfp.weebly.com/uploads/1/1/2/3/11233160/7883961.jpg?728.

[17] "Breiman. *"Machine Learning"*. "2001", Vol. "45", No. "1", pp. "5–32". Available from DOI "10.1023/A:1010933404324". "http://dx.doi.org/10.1023/A:1010933404324" .

Appendix A
Content of the CD

- `/code/` - the source code of our tool
- `/thesis_tex_source/` - the source code to the electronic version of this work
- `/dataset.csv` - the CSV file containing features extracted from CTU-13-Dataset using our tool
- `/Graph-Based-Analysis-of-Malware-Network-Behaviors.pdf` - the electronic version of this work