Revealing Cooperating Hosts by Connection Graph Analysis

Jan Jusko[1,2] and Martin Rehak[1,2]

[1] Faculty of Electrical Engineering, Czech Technical University in Prague
{jan.jusko,martin.rehak}@fel.cvut.cz
[2] Cognitive-Security s.r.o., Prague, Czech Republic

Abstract. In this paper we present an algorithm that is able to progressively discover nodes cooperating in a P2P network. Starting from a single known node, we can easily identify other nodes in the peer-to-peer network, through the analysis of widely available and standardized IPFIX (NetFlow) data. Instead of relying on the analysis of content characteristics or packet properties, we monitor connections of known nodes in the network and then progressively discover other nodes through the analysis of their mutual contacts. We show that our method is able to discover all cooperating nodes in many P2P networks. The use of standardized input data allows for easy deployment onto real networks. Moreover, because this approach requires only short processing times, it scales very well in larger and higher speed networks.

1 Introduction

Peer-to-peer networks generate a significant amount of traffic in today's Internet. Peer-to-peer protocols are popular with file sharing applications, are implemented for a VoIP application (Skype) and have also been adopted by malware as a Command & Control (C&C) channel. The ability to observe peer-to-peer networks is useful — it can be used to manage networks more effectively thus providing better quality of service, to detect and mitigate botnets employing P2P for their C&C architecture, etc. Furthermore, peer-to-peer traffic can degrade the performance of anomaly detection techniques. The detection rate can decrease by up to 30% and false positive rate can increase by up to 45% [9].

In this paper we propose a method that tries to exploit the inherent properties of the peer-to-peer networks to find *cooperating* hosts in the network. We consider two hosts to be cooperating if they are part of the same *overlay* network. We find cooperating hosts by observing their mutual peers. It shows that if two hosts are in the same overlay network their sets of peers overlap. Some theoretical ground for this observation in connection with random graphs can be found in [4].

While graphs and graph algorithms are used to detect peer-to-peer networks in [4,10] our approach differs in both the graph representation and the employed graph algorithm. In [4] the graph is created based on all network traffic and only afterwards the likely members of a peer-to-peer botnet are identified by a graph algorithm. In [10] the graph is created based on flows grouped together by

A.D. Keromytis and R. Di Pietro (Eds.): SecureComm 2012, LNICST 106, pp. 241–255, 2013.

clustering and afterwards its properties are evaluated. Based on the properties the algorithm decides whether the flows were induced by peer-to-peer network or not. Unlike the two, we employ a graph *construction* algorithm and the graph constructed by this algorithm represents a single peer-to-peer network.

In our evaluation we show that the algorithm is able to link hosts cooperating in the usual peer-to-peer networks, such as KAD, Gnutella, BitTorrent and Skype P2P network; and also to link hosts infected by the same malware using peer-to-peer as its C&C channel. Knowing which hosts are engaged in the same overlay with the infected host might help to mitigate the botnet in the network. We believe that this method can be used as a pre-processing layer for packet inspection based detection, where we would first find clusters of hosts in the network and then perform the detection only for few of them and extend the results on the remaining hosts in the cluster.

2 Related Work

There is a plethora of research in the field of peer-to-peer networks. One can find studies of BitTorrent in [17,12,18], BitTorrent's DHT [5], KAD (which is based on Kademlia) in [19,14] and Gnutella in [13,1,15]. There are also many works proposing various improvements to peer-to-peer protocols, but those are not of primary interest here.

Peer-to-peer architecture is now often used by botnets for their C&C. An overview of peer-to-peer botnets and an analysis of one of them can be found in [7]. A peer-to-peer based C&C is, on an example of Kademlia, analyzed theoretically in [8], where the authors show that P2P based C&C is harder to monitor compared to the centralized C&C architecture. Besides that, they also propose several mitigation techniques.

Detection of peer-to-peer networks is another topic often dealt with. There are three main groups of detection methods — packet payload based, flow based methods and graph methods. Within all three groups the detection can be based on the observation of either the specific peer-to-peer network behavior or inherent peer-to-peer networks properties. We do not dive into packet payload based methods in this overview and also skip the methods based on specific peer-to-peer protocol features.

A flow-based method to detect peers using inherent properties of peer-to-peer networks is introduced in [2]. The method itself does not use any protocol-specific features and thus, in theory, might be used for any peer-to-peer network. The authors validate the method on BitTorrent and Gnutella networks.

As an example of graph methods, we can mention one introduced in [3]. The method is agnostic of any specific peer-to-peer protocol features. It creates a connection graph of the peers communicating on a given port and based on the network diameter and number of hosts that function as both client and server determines whether they constitute a peer-to-peer network.

Graphs were used to even greater extent in [10], where they are used to determine whether certain group of flows was generated by the peers in a peer-to-peer

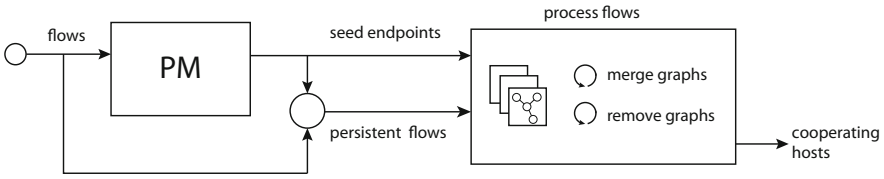

Fig. 1. Schema of the detector. As an input it takes flows from the network which are processed by the Persistence Module (denoted by the PM). The set of seed endpoints is then transferred to the Graph module which processes the flows induced by persistent endpoints and merges and deletes graphs as needed. The output of the detector are sets of endpoints that appear to be cooperating in peer-to-peer networks.

network. The groups of flows are identified based on packet payload inspection, which might limit the potential use of this method. Traffic Dispersion Graphs are also used in [11] to analyze the network traffic and identify unwanted applications.

Peer-to-peer architecture of the botnet C&C is used against the botnet itself to detect its members knowing one starting bot [4]. Their proposed method is similar to ours; it also starts with one known node of a given P2P network and is based on monitoring of mutual contacts. However, they use a rather different graph representation and determine the detected node's confidence after the graph is constructed.

In this work we also use ideas from paper aimed at detecting botnet C&C [6]. The authors focus on observing long term connections that are possible used for botnet C&C. They use whitelists and any long-lasting connection not whitelisted is considered a C&C channel.

3 Detection Method

We propose a detector that takes network traffic as input and finds hosts cooperating in peer-to-peer networks. The detector is composed of two separate modules. At the core of the detector there is the *Graph Module* which constructs graphs around starter nodes, further called *seed nodes*. Nodes are a representation of hosts participating in peer-to-peer networks and each graph represents a single peer-to-peer network. Seed nodes are selected by the *Persistence Module*. The schema of the two is depicted in Fig. 1 and both are further explained in detail in Sections 3.1 and 3.2.

The network traffic processed by the detector is represented by set of *flows*, where flow is a tuple

$$(\text{src ip, src port, dst ip, dst port, protocol}).$$

Flows can be constructed either from NetFlow data or packet capture.

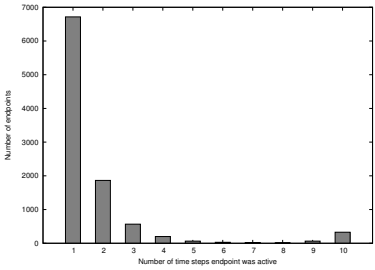

Fig. 2. The histogram shows that majority of endpoints are active only one or two time intervals. Then we can see only a marginal number of endpoints being active between three and nine time steps. All services that run steadily and are regularly used are active all 10 time windows.

The goal of the detection method is to find cooperating hosts in the network. We consider two hosts in the network to be cooperating if they are participating in the same peer-to-peer network. Any host that is participating in an arbitrary peer-to-peer network needs to listen for incoming connections from other peers within the same network — thus it has to keep an open port. Therefore, each peer in a peer-to-peer network can be represented as a tuple *(ip address, port)* which we call *endpoint*.

In reality, the host can participate in more peer-to-peer networks. For each peer-to-peer network it uses, the host needs to keep a listening port open. For each such host, different peer-to-peer networks are represented by independent endpoints, enabling us to separate peer-to-peer networks effectively.

Note that the aforementioned nodes used in the Graph Module and selected by the Persistence Module are in fact endpoints.

Our choice of node representation is different to the one used in [4] because we argue that one host may be taking part in several peer-to-peer networks, e.g. downloading music on BitTorrent, using Skype and at the same time be infected by a P2P botnet. If the authors in [4] chose such a host as a starter node in their graph algorithm, we believe they would suffer a hight false positive rate. We show in the evaluation that using endpoint as the node representation can overcome this issue. In our approach, such a host would simply appear as three distinct endpoints that belong to different graphs.

We would like to note that, while the algorithm could process flows continuously, we process flows in 5-minute batches, i.e. we collect flows for five minutes, which are then processed at once. It follows that observation window size, tryout and ignore periods and memory limit can only be a multiple of 5 minutes.

3.1 Persistence Module

The graph algorithm used in the Graph Module needs a *seed* node around which it constructs the connection graph. The sole purpose of this module is to find such nodes. We already established that nodes representing peers have the form

of *endpoints*. There are two criteria for choosing the seed endpoints — the persistence criterion and peers count criterion.

The persistence criterion means that we choose endpoints that are *persistent*, i.e. are sending or receiving data for longer periods of time. During normal network operation, a single host uses many ports to communicate with other hosts. Most of these ports are used only for a short period of time. However, there are some ports that are kept open — these are usually used for listening for incoming connections. We performed a small experiment on the University network, in which we monitored network traffic in ten 5-minute intervals. In the first time interval we recorded all observed endpoints in our network. In the following 9 time intervals we recorded whether the given endpoints were reused. This way, we were able to create a histogram showing the number of endpoints used in either one, two or up to ten time intervals. The histogram can be found in Fig. 2. We can see that most endpoints were used only in one time interval during the experiment. Then the trend is declining with exception of endpoints that were used during all time intervals. We believe that these are the endpoints that represent services (such as web servers or IMAP servers) or active peers of peer-to-peer networks.

To define persistence of endpoints formally, we use simplified method of measuring persistence introduced in [6]. The original method was focused on revealing hidden C&C channels. We, on the other hand, are interested only in persistence of endpoints, no matter where they connect to. We are not trying to detect exact periodicity of connections but an ongoing character of a connection. For this purpose, the regularity of an endpoint activity is observed by a sliding window W, which is split into n bins. This window is called *observation window* and bins are called *measurement windows*. We can write $W = [b_1, b_2, b_3, ..., b_n]$. We then formally define persistence of an endpoint as:

$$p(e, W) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{e,b_i}$$

where e is the endpoint for which the persistence is calculated, W is the observation window and function $\mathbf{1}_{e,b_i}$ is equal to 1 if at least one connection to or from the endpoint e occurred during the measurement window b_i, otherwise it is equal to 0.

The persistence calculation itself is based on three parameters — *measurement window size*, which states how long the connections are recorded into one bin before proceeding to another, *observation window size*, which determines how many bins there are in the observation window and the *threshold persistence* p^*. This parameter determines how many seed endpoints are passed to the Graph Module.

When moving to the next observation window, we calculate persistence for all endpoints. We select those with the persistence exceeding the threshold p^* and apply the second criterion, which is the number of contacted peers during the last observation window. We assume that any peer communicates with more than one other peer in the peer-to-peer network. Therefore, from the persistent endpoints

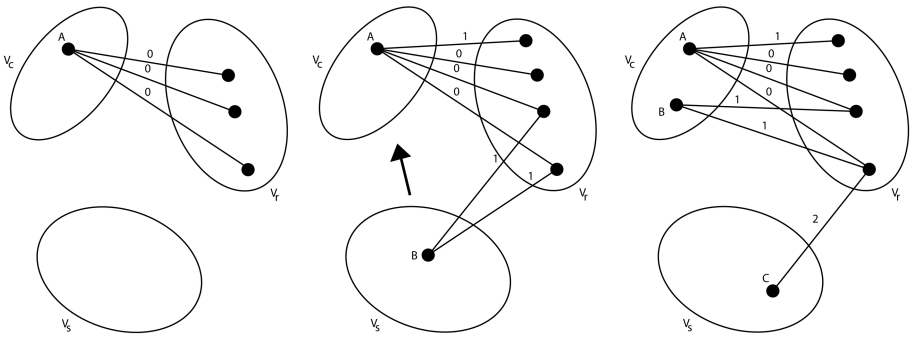

Fig. 3. Algorithm illustration. First we have a seed node A with 3 recorded contacts. In the second time interval, another node, B, is observed, sharing two mutual contacts with A. If we consider $K = 2$, then in the third step, node B is already moved to the V_c. Moreover, the algorithm detected yet another node, which has only one mutual contact with a node from V_c. Note that the weights of the edges in the graph are determined by the time step in which they occurred most recently.

we select those that had at least two peers in the last observation window. This effectively removes long lasting connection between only two peers. These could be clients downloading large files from the Internet or users connecting to other computers via Remote Desktop or SSH.

In the end, only endpoints that exceed the persistence threshold and have at least two peers in the last observation window are passed to the Graph Module as the seed endpoints.

3.2 The Graph Module

The graph module is responsible for

- constructing graphs around the seed endpoints received from the persistence module,
- merging similar graphs,
- removing graphs that failed to find any cooperating host for the given seed endpoint.

Before describing the Graph Module in detail, where we work with the term *graph* extensively, we first introduce its formal definition. Graphs can be used to represent a P2P network, where vertices represent nodes participating in the P2P network and edges represent connections between two nodes participating in the P2P network. To detect the nodes of a P2P overlay network within our network we use a 3-partite weighted graph

$$G = (V, E, w)$$

where

$$V = V_c \cup V_s \cup V_r.$$

V_c is a set of nodes from our network we *believe* are participating in the P2P network, V_s is a set of nodes from our network that we *suspect* are participating in the P2P network and V_r is a set of nodes from outside of our network communicating with nodes from $V_c \cup V_s$. E is a set of edges. Function w assigns each edge a weight — a value equal to the time when the edge was added to the graph. We ignore all intra-network communication and cannot see communication between the nodes that are outside of our network. Therefore the graph we define is indeed a 3-partite weighted graph. This also implies that $G = ((V_c \cup V_s) \cup V_r, E)$ can be considered a bipartite graph.

The algorithm for constructing graph around a seed node is explained later in this section. Graph Module, just like the Persistence Module processes flows collected in the network. Here, however, we process only flows that originate from or are directed towards a persistent endpoint. We do that because we assume, just like in the Persistence Module, that all endpoints representing peers in an arbitrary peer-to-peer network are persistent. Then removing flows assigned to non-persistent endpoints does not compromise the ability of the module to find cooperating peers.

However, before the module can construct any graph, it first needs to receive seed endpoints from the Persistence module. The persistence module feeds seed endpoints to the graph module periodically. When the module receives the first set of seed endpoints it creates a graph for each of them. For every subsequent set of received seed endpoints it checks whether given seed endpoints are already recorded in any of the graphs. For those that are not, it creates new graphs. This way we prevent the creation of duplicate unnecessary graphs.

Since we expect this method to find cooperating endpoints (which are believed to be persistent) we should, after some time, construct graphs that are very similar and describe the same peer-to-peer network despite starting from different seed endpoint. There is no point in keeping such graphs separate so the module joins them together. It rises a question though, how to define "similarity" of two graphs. Two graphs that represent the same P2P network should have similar sets V_c by some measure. However, since both graphs were iteratively constructed from different seed nodes, they do not necessarily contain similar sets of edges or set V_r. Therefore we define *similarity* of two graphs G_1 and G_2 as

$$s(G_1, G_2) = \frac{\mid V_c^{G_1} \cap V_c^{G_2} \mid}{min(\mid V_c^{G_1} \mid, \mid V_c^{G_2} \mid)}$$

where $V_c^{G_1}$ resp. $V_c^{G_2}$ represents V_c of graph G_1 resp. G_2. This definition ensures that similarity of two graphs G_1, G_2 is high (in fact equal to 1) even in the case when $V_c^{G_1} \subset V_c^{G_2}$ and $\mid V_c^{G_1} \mid \ll \mid V_c^{G_2} \mid$. This is a case of two graphs that represent the same P2P network but one of them is much smaller (either because it was created later or because the seed was not as "active" as the seed of the other graph). We merge two graphs if their similarity is greater than the *merge overlap threshold*, which is another algorithm parameter.

There is, of course, a possibility that the graph algorithm will not be able to find any cooperating hosts for certain seed. This might happen when the seed

is the only peer of the respective peer-to-peer overlay in the network, or when the seed node around which we tried to construct a graph was a service, e.g. an email server. If any graph fails to find at least one cooperating endpoint in the network for certain period of time called the *tryout period* it is removed from the module. Even thought we remove the graph, it might be recreated next time the seed nodes are received from the persistence module, because the endpoint might be active despite the fact it has no cooperating nodes. Therefore we define another time parameter, the *ignore period*, that determines how long after removing a graph with a specific seed node, this seed node may not be used to construct another graph. We do not want to ignore the given seed endpoint forever, because service using the port may change or a cooperating peer might appear later.

The Graph Algorithm. As we already mentioned, P2P networks can be represented by a graph. We try to exploit this graph structure to find other participating P2P nodes using one starter node. To achieve this, we traverse the edges of the graph which is constructed on the basis of observed network communication. Since P2P overlay networks are dynamically changing, so should the graph that represents a P2P overlay network.

The detection algorithm monitors network traffic and constructs (modifies) the graph defined in the beginning of the section based on the observed network activity in the following way:

- the graph starts with only the seed node $n \in V_c$,
- when a network connection occurs between any node $n \in V_c$ and some node m outside of our network then there are two options:
  - $m \in V_r$ already; in this case we just update $w(\{m,n\}) = current\_time()$,
  - $m \notin V_r$ yet; in this case we add m to V_r and $\{m,n\}$ to E and set $w(\{m,n\}) = current\_time()$.
- when a network connection occurs between any local node not yet in the graph and some node $m \in V_r$, we add n to V_s, add $\{m,n\}$ to E and set $w(\{m,n\}) = current\_time()$,
- any edge $e \in E$ for which $t_{now} - w(e) > t_L$ is removed from the graph,
- any node $n \in V$ is removed from the graph when it does not have any incident edge (it has a zero degree),
- if $(\exists m \in V_s)(\exists n \in V_c)(\mid Adj(m) \cap Adj(n) \mid > K)$ then we move m from V_s to V_c, where $Adj(n)$ is a set of vertices adjacent to n.

The output of the algorithm is the set V_c which at any given moment contains a list of active P2P nodes in the local network. There are two parameters used in this algorithm:

- a *memory limit*, t_L, which specifies how long a recorded connection (an edge in the graph) is kept in memory,
- a *mutual contacts overlap threshold*, K, which specifies how many mutual adjacent vertices a node from V_s needs to have with any node from V_c to be moved to V_c, i.e. to consider it a P2P node.

First three steps of such an algorithm can be found in Fig. 3.

Table 1. List of peer-to-peer networks with their respective clients installed on the client hosts in the control set. Last column specifies how many hosts is running given client application.

network	client application	hosts
Skype	official client	18
BitTorrent	μTorrent	26
KAD	eMule	15
Gnutella	Phex	18

4 Evaluation

4.1 Experiment Setup

To evaluate the detector we deployed it in the University network consisting of approximately 1000 hosts. Since we did not have access to all the computers and could not establish the ground truth concerning the network activity, i.e. what service did every endpoint in the local network belong to, we chose 155 hosts from two subnets for a small control set.

The first subnet contains 36 hosts of which 18 are running Windows XP, 15 are running Windows 7 and 3 are running Linux. We refer to these hosts as *client hosts*. The client hosts were engaged in casual Internet activity, such as browsing the web, working with email, listening to music, watching videos, sharing files, etc. On these we also installed client applications for several P2P networks, where one host can participate in several peer-to-peer networks. The list of installed client applications can be found in Table 1.

To examine whether the algorithm is capable of linking hosts participating in a botnet, we infected three computers with *Trojan.Sirefef-6* malware, which uses peer-to-peer for its C&C [16]. To ease up the determination of the ground truth for the client hosts we set all client applications belonging to the same peer-to-peer network to use the same port. This has no effect on detection capabilities of our algorithm.

The second subnet contains servers - we refer to this hosts as *server hosts*. None of the them is running any of the aforementioned applications. They run many services, such as web servers, IMAP/POP services and other.

We were collecting network traffic for 20 hours during a working day. The traffic was collected in form of NetFlow data by a network probe. Flows were always collected for five minutes and then sent in a batch to our algorithm. Number of flows within one 5 minute interval ranges from 37000 at night to 240000 during peak hours. To establish the ground truth for the client hosts in the control set, we collected `netstat` information on each client host every five minutes. This was necessary since many applications tend to open more ports than the main port. This way we were able to determine what application did every endpoint of the client hosts belong to. Ground truth for the server hosts was determined in cooperation with their administrators.

Table 2. Parameters and their values used in the experiment

parameter	values
persistence threshold	0.5, 0.8
mutual contacts overlap threshold	3, 4, 5, 6
memory limit	60, 90, 120 minutes
merge overlap threshold	0.3, 0.5, 0.7

Some of the parameters mentioned in the previous sections do not have any impact on detection performance. They are used to tune the memory and processing power requirements of the detector. These are tryout period, ignore period, and measurement and observation window sizes. In our experiments we fixed value of tryout period to 1 hour. Ignore period was set to 1 hour as well, however, every consecutive time the graph around a certain seed node is removed because it failed to find cooperating peers, the ignore period for the given seed increases by 1 hour. Observation window size is 5 minutes, which is also the smallest value we can set (because we process flows in 5-minute batches). Resorting to higher values would extend the time an endpoint needs to become persistent. Measurement window size was chosen in accordance with [4].

The remaining parameters and their values used in the experiment are summed up in Table 2.

4.2 Evaluation Methodology

Since the algorithm runs continually and modifies the graphs according to the changes in the network (hosts joining/leaving peer-to-peer networks) we need to choose a point in time when we evaluate the detection performance. In our control set we started the client application and let them run for several hours. Therefore we decided to choose the point when the numbers of detected nodes of the peer-to-peer networks in their respective main graphs stabilize, i.e. the numbers are same for at least three consecutive time intervals.

It is possible that endpoints participating in the same peer-to-peer network will be spread in several graphs. Therefore we need to choose the *main* graph - the graph that managed to link most of the cooperating hosts from the given peer-to-peer network. We use this graph for the performance evaluation.

Please note that the algorithm does not detect any endpoint until it receives first data from the Persistence Module.

Once we choose the point in time and graphs representing the peer-to-peer networks, we determine the *detection rate* and number of *false positives*.

Client applications used for various peer-to-peer networks differ in usage of ports. Some applications use more than 1 listening port, a typical example being Skype. Another difference is in the number of used ephemeral ports. While clients for peer-to-peer networks based on UDP use only one or small number of ports, clients for TCP based peer-to-peer networks are very eager in using ephemeral ports, e.g. BitTorrent. For each peer-to-peer network and its client application we are interested

Table 3. Detection rate for various *memory limit* and *mutual contacts threshold* values. Memory limit values are intentionally chosen very lower so that the relationship between the two is obvious. Higher mutual contacts threshold may require longer memory limit in order to attain "comparable" detection rate. Memory limit is in first row in minutes, mutual contacts threshold in the first column.

(a) Skype

	5	10	15	25	45
2	94.4	94.4	94.4	94.4	94.4
3	94.4	94.4	94.4	94.4	94.4
4	94.4	94.4	94.4	94.4	94.4
5	94.4	94.4	94.4	94.4	94.4
6	72.2	83.3	94.4	94.4	94.4
7	16.7	22.2	22.2	27.8	83.3
8	16.7	16.7	22.2	22.2	38.9

(b) BitTorrent's DHT

	5	10	15	25	45
2	96.2	100	100	100	100
3	53.8	76.9	92.3	100	100
4	34.6	57.7	69.2	88.5	100
5	34.6	42.3	61.5	73.1	92.3
6	34.6	42.3	57.7	65.4	84.6
7	34.6	42.3	42.3	65.4	76.9
8	34.6	42.3	42.3	61.5	65.4

in the main listening port. In some graphs we may observe several endpoints associated with a single host, especially if they represent a peer-to-peer network using TCP as transport protocol. In a rigorous understanding, these endpoints are *true positives* because they are used for the communication in the peer-to-peer overlay. To keep the things simple, we ignore all endpoints that are in fact true positives but are not associated with the main listening port. If we did not ignore such endpoints we would have issues with the detection rate calculation as we do not know the exact number of ephemeral ports used by a client.

Identification of false positives differs among the peer-to-peer networks. For KAD, Gnutella, BitTorrent and Trojan.Sirefef-6 we consider every detected endpoint not associated with the host from the control set and the respective listening port of the client application to be a false positive. We can do so since these peer-to-peer networks are used only rarely at the University. Using this approach we determine the upper bound of the false positives detected by our algorithm. We cannot do the same with Skype as it is very popular at the University. Therefore we evaluate false and true positives only on the control set.

4.3 Evaluation Results

We evaluated the algorithm performance for all combination of parameters, summed up in Table 2. Before we move on to the actual results of the detection we describe the effect of the particular parameters on the detection performance of the algorithm.

Increasing the persistence threshold in general lowers the number of graphs in the Graph Module. This is important for the performance consideration, especially on huge networks. Having too many graphs in the model can result in exhaustion of the system resources. To focus on detection performance, rising the persistence threshold lowers the number of endpoints induced by the client application but not associated with the main port. It does not seem to have any significant impact on false positives rate.

Table 4. Parameters used for the evaluation of the algorithm. These provide the best results, however they are not the only choice of parameters that attains the same detection performance.

parameter	value
persistence threshold	0.8
mutual contacts overlap threshold	5
memory limit	90 minutes
merge overlap threshold	0.3

Choice of memory limit has a minor impact on false positives rate — the rise of the memory limit is accompanied by the rise of the false positive rate. On the other hand, it can have severe impact on the detection rate (explanation of the connection to mutual contacts overlap threshold, will be introduced shortly). This parameter also impacts memory requirements, high memory limit result in increased memory requirements of the algorithm.

Mutual contacts overlap threshold is the parameter that we believe has the greatest impact on the false positive rate. Increase in its value is accompanied by the drop of the detection rate. There is some boundary (determined by the peer-to-peer protocol) exceeding which the detection rate would drop considerably. This can be easily seen in Table 3a. When using memory limit of 5 minutes, the change from mutual contacts overlap threshold from 6 to 7 causes a significant drop in the detection rate. However, under this limit value, we can attain the same detection rate for various values of mutual contacts overlap threshold just by adjusting the memory limit.

There is a connection between the memory limit and mutual contacts overlap threshold parameters. Rising the mutual contacts threshold while fixing the memory limit lowers the detection rate. On the other hand, raising the memory limit while keeping the mutual contacts threshold fixed improves the detection rate. This is best seen in Table 3.

We did not notice any impact of the merge overlap threshold value on the detection results.

We do not present results for all combinations of parameters, since there are too many of them and many bring the same results. We rather present only the results for one combination of parameters that brings the best results. For the parameters please refer to Table 4.

Detection Rate. The algorithm was able to find all cooperating hosts in Skype, BitTorrent, Kademlia and Trojan.Sirefef-6 peer-to-peer networks. On the other hand, detection rate for Gnutella was considerably lower — 44%.

While Gnutella uses TCP protocol for its communication, Skype, Kademlia and Trojan.Sirefef-6 all use UDP for their peer-to-peer overlay. Finally, newest BitTorrent protocol implementations use both UDP and TCP. In BitTorrent, TCP is used for communication in *swarms*, i.e. the communities created to share files listed in a single torrent file and UDP is used in BitTorrent's DHT implementation, which is utilized for distributed tracker functionality.

As was mentioned before, the algorithm attained 100% detection rate for BitTorrent clients. However, these were detected based on BitTorrent's DHT implementation which is used for distributed tracker functionality and not based on the BitTorrent protocol. The question is whether the algorithm would be able to detect BitTorrent clients that do not use DHT. To verify the detection performance on clients using only BitTorrent protocol without any DHT we ran the algorithm again with the same parameters, while ignoring all UDP connections from or to the μTorrent listening port (effectively removing the DHT traffic).

Here we need to realize the difference between the BitTorrent protocol and the other peer-to-peer protocols in this evaluation. While other peer-to-peer networks maintain an overlay network at all times, the BitTorrent client is not part of any overlay (if it is not using DHT) unless it wants to download a file and joins a swarm. Therefore, when we talk about detecting cooperating hosts for BitTorrent using only the BitTorrent protocol, we mean hosts that are members of the same swarm.

With such setting, we were able to detect all peers cooperating in the same BitTorrent swarm. This shows that even without DHT we were able to find cooperating hosts and that the algorithm is not restricted only to the UDP-based peer-to-peer networks and can be effective for TCP-based peer-to-peer networks as well.

Detecting Gnutella peers seems to be much harder. The algorithm found only 8 peers which constitutes around 44% of all peers. Gnutella uses TCP for communication. Unlike protocols that use UDP and the listening port is used for both incoming and outgoing connections, Gnutella uses the listening port only for incoming connections. Outgoing connections are sent through an ephemeral ports that are assigned and changed at the discretion of the operating system. That makes the detection much harder. Gnutella has two types of peers, *leaf nodes* and *ultrapeers*. Leaf nodes only connect to the ultrapeers and ultrapeers connect to both ultrapeers and leaf nodes. Ultrapeers have higher frequency of connections with other peers and are thus more likely to be linked together. Most of the cooperating hosts found for the Gnutella network were in fact ultrapeers.

The important thing to note here is that linking cooperating hosts (with the exception of BitTorrent peers detection without DHT) did not require any user activity besides connecting (and logging in) to the network.

False Positive Rate. For four of the peer-to-peer networks we experimented on we encountered no false positives. These were Skype, KAD, Gnutella and Trojan.Sirefef-6. Only one false positive was found when linking cooperating hosts in the BitTorrent's DHT network. Due to the low number of false positives we refrain from calculating the false positive rate, since it would only have a negligible value.

5 Conclusion

In this paper we presented a novel method that links cooperating hosts in the same peer-to-peer network by exploiting the inherent properties of peer-to-peer

networks. It tries to reconstruct the peer-to-peer overlay based on the observed connection in the network.

The method managed to detect all cooperating peers in most of the networks and attained almost zero false positive rate.

Since the method does not use neither packet payloads nor flow statistics, it is a viable option for deployment on the backbone network where computationally expensive models are not an option.

We believe that this method presents a viable approach to detecting peers in overlay networks, both well known file sharing networks and specialized peer-to-peer networks used by botnets as a C&C channel.

Acknowledgement. This material is based upon work supported by the ITC-A of the US Army under Contract W911NF-12-1-0028 and by ONR Global under the Department of the Navy Grant N62909-11-1-7036. Also supported by Czech Ministry of Interior grant number VG2VS/189.

References

1. Acosta, W., Chandra, S.: Trace Driven Analysis of the Long Term Evolution of Gnutella Peer-to-Peer Traffic. In: Uhlig, S., Papagiannaki, K., Bonaventure, O. (eds.) PAM 2007. LNCS, vol. 4427, pp. 42–51. Springer, Heidelberg (2007)
2. Bartlett, G., Heidemann, J., Papadopoulos, C.: Inherent behaviors for on-line detection of peer-to-peer file sharing. In: IEEE Global Internet Symposium, pp. 55–60 (May 2007)
3. Constantinou, F., Mavrommatis, P.: Identifying known and unknown peer-to-peer traffic. In: Fifth IEEE International Symposium on Network Computing and Applications, NCA 2006, pp. 93–102 (July 2006)
4. Coskun, B., Dietrich, S., Memon, N.: Friends of an enemy: identifying local members of peer-to-peer botnets using mutual contacts. In: Proceedings of the 26th Annual Computer Security Applications Conference on ACSAC 2010, pp. 131–140. ACM, New York (2010)
5. Falkner, J., Piatek, M., John, J.P., Krishnamurthy, A., Anderson, T.: Profiling a million user dht. In: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement, IMC 2007, pp. 129–134. ACM, New York (2007)
6. Giroire, F., Chandrashekar, J., Taft, N., Schooler, E., Papagiannaki, D.: Exploiting Temporal Persistence to Detect Covert Botnet Channels. In: Balzarotti, D. (ed.) RAID 2009. LNCS, vol. 5758, pp. 326–345. Springer, Heidelberg (2009)
7. Grizzard, J.B., Sharma, V., Nunnery, C., Kang, B.B., Dagon, D.: Peer-to-peer botnets: overview and case study. In: Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets, HotBots 2007, p. 1. USENIX Association, Berkeley (2007)
8. Ha, D.T., Yan, G., Eidenbenz, S., Ngo, H.Q.: On the effectiveness of structural detection and defense against p2p-based botnets. In: DSN, pp. 297–306. IEEE (2009)
9. Haq, I.U., Ali, S., Khan, H., Khayam, S.A.: What Is the Impact of P2P Traffic on Anomaly Detection? In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 1–17. Springer, Heidelberg (2010)

10. Iliofotou, M., Kim, H.-C., Faloutsos, M., Mitzenmacher, M., Pappu, P., Varghese, G.: Graption: A graph-based p2p traffic classification framework for the internet backbone. Comput. Netw. 55(8), 1909–1920 (2011)
11. Iliofotou, M., Pappu, P., Faloutsos, M., Mitzenmacher, M., Singh, S., Varghese, G.: Network monitoring using traffic dispersion graphs (tdgs). In: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement, IMC 2007, pp. 315–320. ACM, New York (2007)
12. Kryczka, M., Cuevas, R., Guerrero, C., Azcorra, A.: Unrevealing the structure of live bittorrent swarms: Methodology and analysis. In: 2011 IEEE International Conference on Peer-to-Peer Computing (P2P), August 31-September 2, pp. 230–239 (2011)
13. Li, C., Chen, C.: Topology analysis of gnutella by large scale mining. In: International Conference on Communication Technology, ICCT 2006, pp. 1–4 (November 2006)
14. Liu, X., Li, Y., Li, Z., Cheng, X.: Social Network Analysis on KAD and Its Application. In: Du, X., Fan, W., Wang, J., Peng, Z., Sharaf, M.A. (eds.) APWeb 2011. LNCS, vol. 6612, pp. 327–332. Springer, Heidelberg (2011)
15. Evangelos, P.: Markatos, Tracing a large-scale peer to peer system: An hour in the life of gnutella. In: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID 2002, p. 65. IEEE Computer Society, Washington, DC (2002)
16. McNamee, K.: Malware analysis report - botnet: Zeroaccess/sirefef (February 2012), http://www.kindsight.net/sites/default/files/Kindsight_Malware_Analysis-ZeroAcess-Botnet-final.pdf
17. Móczár, Z., Molnár, S.: Characterization of BitTorrent Traffic in a Broadband Access Network. In: Szabó, R., Zhu, H., Imre, S., Chaparadza, R. (eds.) AccessNets/Selfmagicnets 2010. LNICST, vol. 63, pp. 176–183. Springer, Heidelberg (2011)
18. Qi, J., Zhang, H., Ji, Z., Yun, L.: Analyzing bittorrent traffic across large network. In: 2008 International Conference on Cyberworlds, pp. 759–764 (September 2008)
19. Steiner, M., En-Najjary, T., Biersack, E.W.: A global view of kad. In: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement, IMC 2007, pp. 117–122. ACM, New York (2007)