On Advanced Monitoring in Resilient and Unstructured P2P Botnets

Shankar Karuppayah*†, Mathias Fischer*, Christian Rossow‡, Max Mühlhäuser*

*Telecooperation Group, Technische Universität Darmstadt / CASED firstname.lastname@cased.de

†National Advanced IPv6 Center, Universiti Sains Malaysia (USM), Penang, Malaysia

‡Horst Görtz Institute for IT-Security (HGI) Ruhr University Bochum, Germany christian.rossow@rub.de

***Abstract*—Botnets are a serious threat to Internet-based services and end users. The recent paradigm shift from centralized to more sophisticated Peer-to-Peer (P2P)-based botnets introduces new challenges for security researchers. Centralized botnets can be easily monitored, and once their command and control server is identified, easily be taken down. However, P2P-based botnets are much more resilient against such attempts. To make it worse, botnets like *P2P Zeus* include additional countermeasures to make monitoring and crawling more difficult for the defenders. In this paper, we discuss in detail the problems of P2P botnet monitoring. As our main contribution, we introduce the Less Invasive Crawling Algorithm (LICA) for efficiently crawling unstructured P2P botnets and utilize only local information. We compare the performance of LICA with other known crawling methods such as Depth-first and Breadth-first search. This is achieved by simulating these methods on not only a real-world botnet dataset, but also on an unstructured P2P file sharing network dataset. Our analysis results indicate that LICA significantly outperforms the other known crawling methods.**

I. Introduction

Many cyber-crimes such as banking fraud, spam, and denial-of-service are executed via botnets that can encompass more than hundreds of thousands infected machines all over the globe. Early botnets utilized the client-server principle, in which all bots are controlled by a centralized Command-and-Control Server (C2), e.g., by Internet Relay Chat (IRC) or Hypertext Transfer Protocol (HTTP) servers. Centralized botnets are easy to deploy and have a low communication overhead and latency. However, centralized servers also serve as a single point of failure that can be easily taken down and thus endanger the whole botnet.

The paradigm shift of P2P-based botnets, e.g., *Storm* [1] and *P2P Zeus* [2], [3], [4], is a step forward that overcomes the drawback of centralized botnets. These P2P-based botnets establish an overlay for communication among the participating bots instead of relying on a centralized server. Although a P2P botnet completely relies on potentially unreliable end systems, i.e., infected machines, the botnet is much more resilient against attacks on its operation from the outside than a centralized C2 botnet.

To understand if an attack on a P2P botnet is possible, it is necessary for the researchers to monitor the botnet. Information such as the population size and the infected machines of a botnet can be estimated and identified. Moreover, the information about the connectivity within the botnet is required when attempting to disinfect them as well as to plan a coordinated shutdown or sinkholing of the network [5], [6].

The most common methods to monitor P2P botnets is to utilize crawlers or to deploy sensor nodes in the botnet [3]. Crawlers are used [3] to obtain *snapshots* of botnets in graph representation for further analysis, for disclosing the identity of participating bots, and to get an estimation of the botnet size. Alternatively, multiple sensors are deployed that try to spread the knowledge about each other in the botnet to become popular and to get contacted by other bots. Although, this method allows a better approximation of the size of botnets than with crawlers, the deployment of sensor nodes alter the connectivity properties within the botnet. Therefore, we focus on crawlers throughout the remainder of this paper.

Crawling a botnet presumes a reverse-engineered botnet malware and a *seedlist* of some bots actively participating in the botnet. After that, a crawler that follows the botnet protocol needs to be developed that bootstraps to the botnet and conducts the actual crawl by iteratively contacting the available bots.

P2P botnets can be either structured or unstructured. In structured P2P networks, the overlay is organized into a specific topology on the basis of a structured ID space that guarantees efficient lookup functionality. Early P2P botnets such as the early variants of the *Storm* botnet [7] utilized this architecture. However, due to the imposed structure, such botnets can be crawled efficiently, e.g., by crawlers similar to the Kademlia crawler of Salah and Strufe [8].

In contrast, crawling unstructured P2P networks is more challenging as there is no structure that can be exploited for crawling. This overlay structure is established and maintained by an internal membership management mechanism, e.g., a gossiping approach [9], [10]. For that, each node maintains a neighbor list that contains a small subset of other known nodes. These lists are shared between nodes upon request to replace inactive neighbors and thus to ensure overlay connectivity. A crawler for unstructured P2P botnets can leverage this exchange of neighbor lists to iteratively request neighbor lists of participating bots until all nodes are known to the crawler. However, due to the lack of structure, estimating the network

size or getting to know all peers is much more difficult to obtain than for structured botnets

In the remainder of this paper, we focus solely on crawling unstructured P2P botnets and refer to them as P2P botnets and interchangeably use the terms *peer* and *node* with *bot*.

Crawling a P2P botnet generates a huge amount of activity between the crawler and the botnet. This might disclose the crawler and a blacklisting by the botnet can be the result. The *P2P Zeus* botnet [4], [3] implements such a countermeasure. Bots in *P2P Zeus* ban an IP address permanently, if they receive more than twelve subsequent requests from a single host within one minute. Existing crawling approaches known in P2P botnets are Depth-First Search (DFS) and Breadth-First Search (BFS). However, these crawling approaches are easily identifiable as they crawl botnet exhaustively, i.e., all nodes. As a result, crawling a botnet becomes more difficult. Hence, to keep crawling such botnets, the crawler need to remain undetected.

Our main contribution in this paper is the Less Invasive Crawling Algorithm (LICA) that optimizes the crawling process and decreases the crawler interaction with the botnet. For that, it heuristically approximates a *vertex cover*, i.e., number of nodes to be visited to obtain a full view, by presuming only restricted knowledge about the botnet overlay. We compare the performance of our algorithm with BFS and DFS crawling approaches and found that LICA significantly outperforms them in our experimental settings on two real-world datasets. Moreover, we also show that the results of *LICA* are close to the results of an approximation algorithm for *minimum vertex cover* [11] that operates on global knowledge about the network.

The remainder of this paper is organized as follows. Section II discusses the related work in crawling of unstructured P2P botnets. Section III introduces our LICA algorithm with a formal model. Section IV evaluates our method on the *P2P Zeus* botnet and an unstructured P2P file sharing network, i.e., *Gnutella*, dataset. Finally, Section V summarizes our contribution and describes the future work.

II. Related Work

In this section, we summarize the related work in crawling techniques and countermeasures against crawling from the perspective of a botnet. First we discuss existing unstructured P2P crawling mechanisms, followed by the emergence of anti-crawling/enumeration botnets. We then describe unavoidable bias results in crawls before summarizing this section.

1) Existing Crawling Mechanisms: The following mechanisms have been utilized in crawling either real unstructured P2P botnets or file sharing systems. The botnet crawlers implemented either a DFS or BFS-based queue implementation as a node selection strategy.

- ***Storm Crawler*** [7]: A BFS-based crawler used for the infamous *Storm* botnet that iteratively queries each peer starting from the *seedlist*.
- ***Nugache Crawler*** [12]: This crawler was used in crawling the *Nugache* botnet. It conducts pre-crawls and utilizes those information as an input for their priority-queue based crawler implementation that prioritize nodes which have been observed more often available and responding in the pre-crawls. The priority mechanism is implemented as a DFS approach.
- ***P2P Zeus Crawler*** [3]: This BFS-based crawler is used in crawling the recent variants of *P2P Zeus* botnet. It starts the crawling from a seed node and appends undiscovered nodes at the end of a queue. Crawling is repeated after all nodes have been crawled.
- **Cruiser** [13]: This crawler was used to crawl the *Gnutella* file sharing network. It prioritizes *ultrapeers* from the two-tier design of the *Gnutella* network and therefore able to achieve a more accurate representation of the P2P network. The authors also report of a biased connectivity among peers which are most likely connected to peers with higher uptime.

2) Anti-Crawling Techniques: Sinkholing is the process of strategically invalidating entries in the neighbor list of nodes in a botnet to be able to contact only a small set of nodes, i.e., sinkholes. This sinkholing process leveraged the crawling mechanism to poison the neighbor lists of other nodes in the network. Recent P2P botnets were sinkholed [5], [3], [14] using this technique to take them down. Sinkholing renders the botmaster to loose control of the botnet as the bots are no longer connected to the overlay.

Along this line, we discuss anti-crawling or anti-enumeration techniques which aim to make it difficult for the defenders to enumerate and monitor the botnets. We first look at work in the area of anti-crawling in theoretical botnets and then discuss a real-world case scenario.

- ***RatBot*** [15]: Yan et al. present a theoretical anti-enumeration structured P2P botnet based on a DHT that spoofs a portion of IP addresses to hinder attempts to enumerate the botnet. This makes the crawling process difficult due to additional non-responsive nodes during crawl. It also leads to an overestimation of the botnet size, which may be a preferred feature for the botmasters, e.g., publicity in media or among potential clients. Nevertheless, crawling mechanisms introduced above may still work with the presence of the *RatBot*'s anti-crawling techniques, although with more biased results.
- ***P2P Zeus*** [2], [4]: The recent variants of the *P2P Zeus* implement an anti-crawling mechanism that blacklists a node that frequently requests a neighbor list, i.e., > 12 subsequent crawls within 60 seconds. This defense mechanism is in place locally for each bot. Furthermore, the botnet also limits the number of entries in the neighbor list that are being returned for every request to 10 entries. Therefore, although crawling is still possible, it requires more effort and longer crawling pauses, i.e., sleep time, to obtain a snapshot of the botnet.

3) Networks Noise: Some researchers that have conducted crawls on botnets have also shared the lessons learned and problems that arise during these crawls [12], [16].

Kanich et al. first addressed the problems [16] in crawling the *Storm* botnet. Their major finding is that other crawling and sinkholing attempts bias the crawling results. Hence, it is necessary to sanitize the crawled data from such attempts. Furthermore, Rajab et al. presented the challenges [17] in measuring the botnet size due to factors such as high churn rates. This is supported by Stutzbach et al. [13] that reported crawler efficiency as the most crucial requirement to get an accurate picture of the botnet in the presence of node churn.

We believe that from the details of the work presented, existing crawlers are not efficient enough by just utilizing BFS or DFS methods to crawl the botnets. Recent botnets like *P2P Zeus* implement anti-crawling mechanism to prevent monitoring attempts by introducing blacklisting and restricted neighbor list size.

For this reason, we need a less invasive monitoring mechanism for P2P-based botnets. This new mechanism needs to crawl fast, while introducing only minimal network activity to stay undetected.

III. Less Invasive P2P Botnet Monitoring

Monitoring a P2P botnet can be done by crawling or by deploying sensor nodes. Although sensor nodes do not require any active communication once they participate within the botnet, they introduce a bias to the crawled graphs. As we intend to monitor the overlay structure of the botnet, we focus only on the crawling technique in this paper.

Crawling introduces a significant amount of network activity that is easily observable and may disclose a crawler to the botnet. For example, current BFS and DFS-based crawling algorithms need to crawl all possible nodes to provide a snapshot at any particular time. However, it is possible to minimize the necessary amount of interaction between crawler and botnet. The idea behind is to only crawl a subset of all nodes to obtain a *minimum vertex cover*, which is a problem known from graph theory. A *vertex cover* is a set of vertices of a graph that has all edges in the graph incident to at least one vertex of the set. A minimum vertex cover is then the set of minimum number of vertices that are needed to cover the entire network. However, this problem is proven to be NP-hard and all known approximation algorithms require a global view on the graph.

For this reason, we intend to approximate the *minimum vertex cover* during the crawl of a botnet. For that, we try to identify the *stable core* of a botnet and to crawl this first, which is inspired by the work of Stutzbach et al. [13] on the *Gnutella* P2P network. Unstructured P2P networks, like the botnets we observed, maintain their overlay connectivity via membership management protocols. These protocols ensure the robustness of the overlay by exchanging fresh information about active peers in the network. The entry of a *stable* or an *important* peer is frequently shared and will stay longer in the neighbor list of many other peers.

To exploit this finding, we propose an iterative crawling method that employs a heuristic to plan the next crawling steps iteratively and to establish a vertex cover in the botnet. For that, we first introduce a formal model and subsequently describe our novel crawling algorithm.

A. Formal Botnet Model

A P2P botnet is a directed graph $G := (V, E)$, where V is a set of peers in the botnet. All bots follow a *membership management* protocol that establishes and maintains neighborhood relationships in the botnet to ensure a connected overlay. Hence, peers have connections to a subset of other peers in the overlay. These connections or edges $E \subseteq V \times V$, are represented as a set of directed edges (u, v) with $u, v \in V$. The neighborhood relationship, i.e., *neighbor list*, of a peer $v \in V$ can be defined as $\mathrm{NL}_v = \{u | \forall u \in V : (v, u) \in E, v \neq u\}$ and contains the set of all nodes that v has an outgoing connection to.

The *minimum vertex cover* in a botnet is then defined as a set of minimum nodes, V_{min}, that has all other nodes in the network reachable from one or more nodes in the set according to our formal botnet model as follows:

$$V_{min} = \arg\min\{|V'| \mid V' \subseteq V : \left(\bigcup_{v \in V'} \mathrm{NL}_v\right) = V\}$$

B. Crawling Algorithm

Although the simplest solution in crawling a botnet is to iteratively request neighbor lists from all known nodes, this is neither stealthy nor efficient. Unnecessary activity of frequently requesting neighbor lists may not only raise suspicions to the botmasters, but also introduce bias to the final view if the crawl is not carried out fast enough [13].

A *minimum vertex cover* would be the optimal solution to this problem as it would allow to crawl the least amount of nodes to get a full view on the network. However, the problem itself is NP-complete and only approximation algorithms that presume the full view on the graph are known, e.g., the algorithm of Bar-Yehuda [11]. As crawling is performed starting from one peer and with an iteratively increasing view on the network by acquiring neighbor lists, we need a heuristic for a vertex cover that operates with such sparse graph information.

We designed a crawling algorithm for unstructured P2P botnets that can optimize the coverage of subsequent crawling steps and thus decreases the overall required number of steps for crawling a botnet. Hence, we do not intend to conduct a complete botnet enumeration, but to extend our monitoring coverage to have the best, i.e., largest, snapshot of a botnet overlay. We assume that bots in the botnet are all online at the same time, hence, we ignore diurnal patterns and churn. Furthermore, we assume that we can receive neighbor lists (or a random subset thereof) from other bots, so that we learn either all neighbors or subsets of the neighbors of one particular peer.

Our *Less Invasive Crawling Algorithm (LICA)* is described in Algorithm 1. It does not only aim on crawling efficiency but is also configurable for an adaption to a specific environment or a specific botnet via parameters $\mathrm{seedpeer}$, r, w, and t.

The $\mathrm{seedpeer}$ is the start node of the crawl. Parameter r is the maximum number of requests allowed to be sent, i.e., subsequent crawling iterations, to any node in the network

Algorithm 1: LICA(seedpeer, r, w, t)

```
   // Initialization
1  V_known ← seedpeer
2  c_crawl ← 0
3  gain ← 0
   // Maximum allowed requests (per node)
4  for i = 0, i < r, i = i + 1 do
       // Utilize previous crawl
5      V_crawl ← V_known
6      V_visited ← ∅
7      while
       V_crawl ≠ ∅ & (c_crawl mod w ≠ 0 || gain ÷ w > t) do
           // Reset gain if necessary
8          if c_crawl mod w = 0 then
9              gain ← 0
           // select the next node to crawl
10         Choose u ∈
           arg max_{∀v∈V_crawl} Σ_{∀w∈V_known} |NL_w| − |NL_w − v|
           // Crawl + get neighbor list of u
11         NL_u ← crawl(u)
           // Update list of visited nodes
12         V_visited ← V_visited ∪ {u}
           // Update list of nodes to crawl
13         V_crawl ← (V_crawl ∪ NL_u) − V_visited
14         c_crawl ← c_crawl + 1
           // Calculate gain
15         gain ← gain + |NL_u| − |V_known ∩ NL_u|
           // Update visited nodes
16         V_known ← V_known ∪ NL_u
```

within a particular full crawl. A full crawl ends when all contactable nodes in the network have been discovered.

The window parameter w determines the number of subsequent requests, for which a *gain*, e.g., ≥ 0, is calculated. The *gain* measures the number of new nodes learned during a crawling window w. Thus, the *gain* divided by w requests gives the *learning curve* during the crawl which terminates the algorithm execution when dropping below a threshold t, ≥ 0.0.

LICA utilizes the initial seedpeer for bootstrapping itself into the botnet overlay. Then, starting with the seedpeer, our algorithm obtains the neighbor list NL_u from u (line 11) and further extends its knowledge by iteratively requesting neighbor lists from the discovered peers. For each request sent by *LICA*, counter c_{crawl} is incremented by one.

Upon receiving a neighbor list from u, it is immediately added to V_{visited} (line 12) and the undiscovered peers in the received entries are added to V_{crawl} (line 16) as potential candidates for the crawl.

Line 10 in the algorithm selects the next candidate for the crawl. The algorithm goes through all received neighbor lists of peers in V_{known} and ranks all remaining peers in V_{crawl} based on their popularity, i.e., its in-degree or the number of the occurrences among the neighbor lists of previously crawled peers. The function $\arg\max$ returns the most-popular peer, i.e., highest ranked, as the next candidate to be crawled. In the event of equally ranked peers, the algorithm randomly chooses between equally ranked peers.

At every window interval, i.e., after w requests, the algorithm checks the accumulated gain (line 15) within the past window and terminates the current crawl iteration if the ratio of the observed gain drops below threshold t (line 7). Depending on the value of r, *LICA* may repeat another iteration of the crawl; however, this time *LICA* utilizes the information of previously crawled peers V_{known} instead of the seedpeer. The algorithm terminates when there are no more peers to be crawled or the number of maximum allowed iterations is exceeded.

In the next section, we evaluate *LICA* regarding its crawling performance and compare it against a BFS and DFS-based crawling.

IV. Evaluation

We describe the details of the used datasets and explain our experimental setup in this section. Furthermore, we present the evaluation results on the crawling performance of *LICA* at the end of this section by answering the following questions:

1) When should a crawling process terminate to avoid further crawling unnecessary nodes?
2) How can we improve a crawling algorithm's performance by utilizing only local information, e.g, neighbor lists?
3) How can we measure the efficiency of a crawling algorithm?

A. Data Sets

We utilized two different real-world unstructured P2P network datasets in the form of directed graphs, i.e., *P2P Zeus* and *Gnutella*, to evaluate the performance of crawling algorithms.

The *P2P Zeus* dataset used in this evaluation consists of crawling information collected in approximately five hours from the *P2P Zeus* botnet on 25th April 2013. It has been obtained from previous work in analyzing *P2P Zeus* [3]. From the initial $1,061,402$ edge entries in the database, we removed $667,704$ edge entries that consist of biases that are already known to us: sinkholed nodes (identified by an out-degree < 10), sensor nodes (identified by an in-degree > 500), and duplicated edges. We also note that the sanitizing of the dataset may not be perfect due to other monitoring activities of researchers that we were not aware of.

The second dataset is crawl data of the unstructured P2P file sharing network *Gnutella*, on August 2002 that we obtained from the SNAP repository[1].

The summary of the datasets is provided in Table I.

[1]SNAP: http://snap.stanford.edu/data/

Dataset Name	P2P Zeus	Gnutella
Nodes	82,471	62,586
Nodes (out-degree > 0)	10,794	16,387
Avg. Neighbor List Size	4.6	2.4
Highest Neighbor List Size	97	78
Edges	379,088	147,892
Avg. Clustering Coefficient	0.01934	0.00047
Diameter	11	31
Avg. Path Length	5.2	9.2

TABLE I
GRAPH PROPERTIES OF THE DATASETS.

B. Experimental Setup

We conducted our analysis using *Python* and the *NetworkX* module [18], and implemented all the crawling algorithms, i.e., *LICA*, *BFS*, and *DFS*, with Python scripts. 50 independent experiments were executed on each of our analysis and the presented results were averaged over them. For every iteration of the experiment, our simulation uniformly chooses a common seed peer for all the crawling algorithms.

We limit the size of the neighbor lists in our simulation. The reason behind this is that the datasets used in our experiments are observations of multiple crawls over some time. As such, some nodes have accumulated neighbors more than their allowed limit for a neighbor list, i.e., different neighbors due to node churn and diurnal effects. Since the neighbor list's size of the *P2P Zeus* botnet is 50 entries, we also limit the size of the neighbor lists to 50 in our simulation to closely resemble it's implementation.

Considering that some nodes in the datasets have more than 50 neighbors, we shuffle and split the entries of these nodes into a sequence of chunks, each having a maximum of 50 entries. For every received neighbor list request from our implemented crawling algorithms, a node will return a single chunk from its sequence and repeats the sequence after returning the last chunk (if queried further). In our simulation, a full crawl ends when there are no more new peers to crawl. In addition to that, *LICA* also ends it's crawl when the maximum allowed iterations have exceeded.

First we conduct an analysis to identify the best combination of values for the parameters w, r, and t based on the *P2P Zeus* dataset. Then, we measure the performance of each crawling algorithm by plotting the ratio of discovered peers in relation to the number of neighbor list requests sent by the crawling algorithms.

We also measure the efficiency of each of the crawling algorithms by the ratio of nodes discovered in dependence on the number of required crawling steps, i.e., neighbor list requests. For that, we applied the local-ratio approximation of *minimum vertex cover* presented by Bar-Yehuda et al. [11] to obtain an approximate value of the number of minimum nodes to be crawled to obtain a snapshot of the botnet graph. The approximation ratio of this algorithm is $2 - \frac{1}{k}$, where k is the smallest integer.

The implementation of this approximation algorithm that is available in *NetworkX* operates on undirected graphs. Hence, we modified it to be applied on directed graphs. In the remainder of this evaluation we will refer to this algorithm as *Approximative Minimum Vertex Cover (AMVC)*.

Furthermore, for the clarity of the resulting plots, all algorithms terminate their crawling as soon as 95% (indicated by the dashed horizontal lines) of nodes in the datasets have been discovered unless stated otherwise. The reason behind this is that all crawling algorithms produce very minimal additional observations towards the end of the crawl, i.e., > 95%. As such, they shrink the overall resulting plot and therefore are omitted due to their minimal significance.

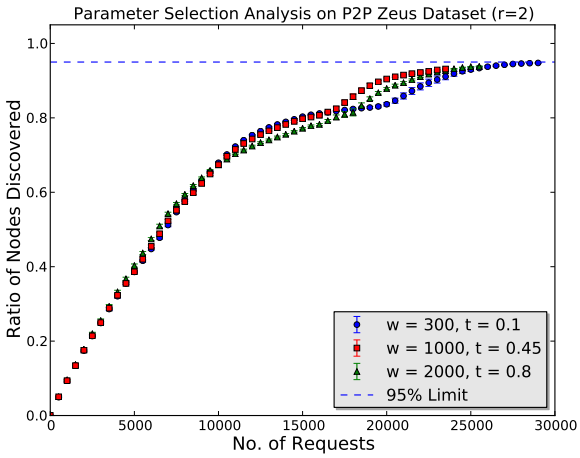
C. Evaluation Results

Existing crawling algorithms which are implemented using *BFS* or *DFS* methods to crawl botnets may not be efficient. The node selection criteria used by both these algorithms is not based upon any other information except the order the nodes are stored and processed. Therefore, we conducted a series of experiments to understand the impact of the node selection criteria on the crawling performance of *LICA*.

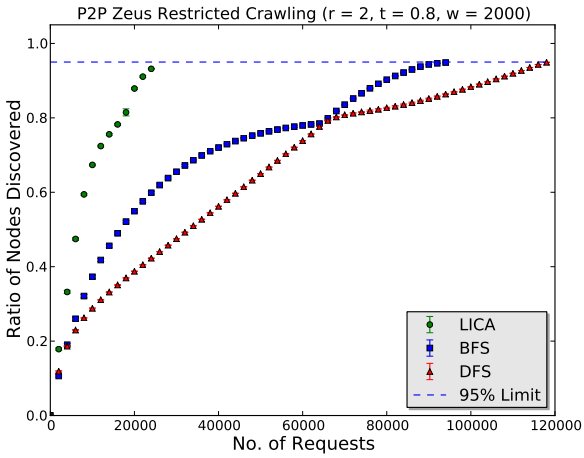
We first investigated on when to terminate an ongoing crawling process to avoid too many unnecessary crawling steps. For that, *LICA* contains a simple mechanism that checks the gain after a window w of crawling steps and that terminates when the gain drops below threshold t during the crawl. As the algorithm utilizes a *learning curve* to terminate the crawl, the algorithm is adjustable by manipulating its parameters. For example, to overcome the blacklisting mechanism of *P2P Zeus*, the r value can be set to 11, i.e., maximum number of requests that are allowed to be sent to a particular node. Alternatively, subsequent full crawls can be delayed by 60 seconds. We set the value r to 2 in all our experiments because we know from our *P2P Zeus* datasets, that all algorithms need to request the neighbor lists from any node at most twice, i.e., two chunks, to obtain the full neighbor list.

Furthermore, by deciding combinations of values for the window size w and threshold t, the resulting crawl can be shaped. For example, the user can specify a high threshold value, e.g., > 1.0 in combination with a high window size, e.g., 3000 when they intend to crawl mainly the backbone nodes. Similarly, when the intention is to crawl as many nodes as possible, the threshold value can be set to a relatively low value, e.g., < 0.05 in combination with a low window size, e.g., 300. We analyzed the effects of different combinations of promising values on the *P2P Zeus* dataset with the value r = 2 as presented in Figure 1(a).

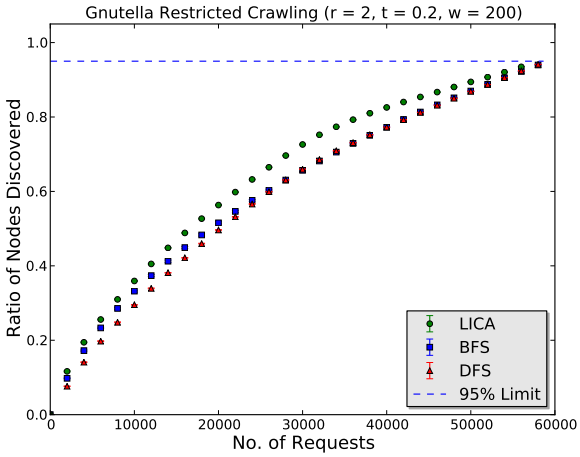
From the analysis, we identified that when the threshold value t, is low, e.g., 0.1, and with the window value w of 300, a full crawl results in about 94.7% of the entire dataset known just with about 29,000 sent requests. Meanwhile, the parameter combination of t = 0.45, w = 1000 obtained a lower coverage of 93.3% although with 17.4% lesser requests than the previous combination. However, with an increased threshold value, e.g., 0.8, and window value w = 2000, *LICA* terminates with a coverage of 93.9% although require additional 1,800 requests than the previous combination.



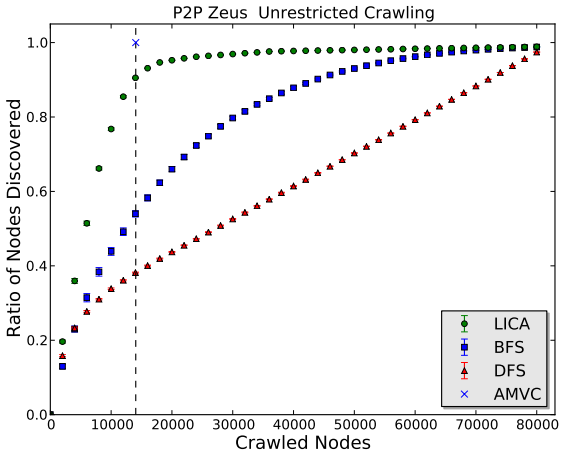
(a) Analysis of *LICA*'s Parameter Selection on P2P Zeus Dataset



(b) P2P Zeus Restricted Crawl Analysis



(c) Gnutella Restricted Crawl Analysis



(d) P2P Zeus Unrestricted Crawl Analysis

Fig. 1. Performance analysis of *LICA*, *BFS*, and *DFS*. (a) The performance of *LICA* under different combination of parameters. The performance of all observed crawling algorithms on (b) *P2P Zeus* and on (c) the *Gnutella* dataset, measured in the ratio of nodes discovered in dependence on the total number of requests sent. (d) contains the results of all crawling algorithms on the *P2P Zeus* dataset without any neighbor list restrictions, plotted by the ratio of nodes discovered depending on the total number of nodes crawled.

Therefore, we decided that the combination, $\mathrm{t} = 0.8, \mathrm{w} = 2000$ is more reasonable to efficiently crawl the *P2P Zeus* dataset. Although the criteria in selecting the best combination of parameters is not straight forward, based on the requirement of the crawl, we can fine-tune subsequent crawls with more appropriate parameters.

Next, we investigated on the performance of *LICA* during a crawl. It is expected that *LICA* performs better than *BFS* and *DFS* on both real-world datasets, because we prioritize popular nodes during the crawling. This expectation is based on the work of Stutzbach et al. [13], which reports that by crawling the *ultrapeers*, i.e., backbone nodes, in the *Gnutella* network resulted in a faster and efficient crawl.

The results of our crawl performance analysis is presented in Figure 1(b) and 1(c). *LICA* was executed on the *P2P Zeus* dataset using the previously chosen parameter combination: $r = 2$, $t = 0.8$, and $w = 2000$. The crawl performance in Figure 1(b) indicates a much better performance of *LICA* in comparison to the other methods. Note that for the plot of *LICA*, we left out the points in which we have results from less than 16 individual experiments as we cannot obtain a confidence interval from that. *LICA* requires $25,780$ requests to obtain a 93.86% coverage of the peers in the botnet. This is about 27% of the total requests made by the *BFS* algorithm to obtain a 95.0% coverage. *DFS* performed worst in this analysis by requiring additional 400% requests than needed by *LICA*.

The convergence point between the *BFS* and *DFS* algorithm indicates the point where all known nodes during the initial crawl have been crawled. The growth that is observed after that point is from new nodes discovered from re-requesting neighbor lists from previously known nodes, i.e., subsequent chunks of their neighbor list. This convergence behavior is not observed in *LICA* because it terminates the crawling when the observed gain drops below the threshold, and the gain

immediately picks up in the subsequent crawl iteration.

It is worth mentioning that by reducing the size of the neighbor lists or the returned subset of the list, the effort to crawl the entire network increases proportionally for all crawling algorithms. We verified this by running another set of experiment with a returned neighbor list of size 30 and r = 3. The observed performance between the crawling algorithms remain relatively similar to the the results in Figure 1(b). Due to space constraints, we omit the results in this paper.

We repeated the experiment on the *Gnutella* dataset using the following parameter combination: $r = 2$, $w = 400$, and $t = 0.3$. However, the performance gain of *LICA* for *Gnutella* dataset in Figure 1(c) is not as significant as in the *P2P Zeus* dataset. We further investigated this behavior and identified that the diameter of this dataset is very high, i.e., 31, with an average path length of 9.2. Moreover, nodes in the dataset have a rather low average size of the neighbor list, e.g., 2.4 entries. Hence, due to the inherent network structure in this dataset, the gain is much lower, as all crawlers need to go through almost every available nodes to obtain a full view. Nevertheless, the performance of *LICA* is better compared to the other two algorithms as presented in Figure 1(c). For example, with $31,941$ requests, *LICA* discovered 75.5% of nodes that is about 7% more than the other two algorithms.

Finally, we wanted to measure the efficiency of the crawling algorithms. For this purpose, we compared the performance of all the three crawling algorithms on the *P2P Zeus* dataset with respect to the *AMVC* value in Figure 1(d). For simplicity, we modified our simulation settings to allow all available neighbors of a node to be returned in a single request and disabled the crawl termination mechanism in *LICA*. As such, the purpose of this particular analysis is to find out how many nodes need to be crawled in order to obtain the full view of the network.

Based on this analysis, we managed to demonstrate by heuristic, that *LICA* outperforms other methods in performing closer to the calculated *AMVC* value, $14,050$ nodes. At the point of the *AMVC*, *LICA* discovered a total of 90.6% nodes in comparison with *BFS* that only discovered 54.1% or *DFS* with 38.2% of nodes. This is interesting because we managed to show that by crawling and prioritizing the '*popular*' peers, we are actually crawling the backbone of the network. This corresponds to the finding of Stutzbach et al. [13] that reports the existence of biased connectivity with peers with higher uptime, i.e., *popular* nodes in our work. This allowed *LICA* to exploit the feature and outperform existing crawling algorithms.

V. Conclusion

In this work, we discussed the needs of an efficient crawling algorithm to monitor modern P2P botnets in the rise of botnets that implement anti-crawling mechanisms. We also demonstrated that using local information that we can obtain, the botnet crawling performance can be further improved compared to current existing methods. We compared the performance of our method on two different real-world data sets. The performance of our method indicated that a botnet crawling conducted using our LICA method yields better performance and much closer to the approximation of a *minimum vertex cover*. The results of our work can be utilized as a stepping stone to monitor these resilient P2P botnets and analyze the next steps to strategically take them down. As a future work, we look into implementing LICA in a real P2P botnet crawler to measure its performance. Moreover, we would also like to extend our current method to analyze other botnet datasets.

References

[1] S. Stover, D. Dittrich, J. Hernandez, and S. Dietrich, "Analysis of the Storm and Nugache trojans: P2P is here," *USENIX; login*, vol. 32, no. 6, pp. 18–27, 2007.

[2] D. Andriesse and H. Bos, "An Analysis of the Zeus Peer-to-Peer Protocol," 2013.

[3] C. Rossow, D. Andriesse, T. Werner, B. Stone-gross, D. Plohmann, C. J. Dietrich, H. Bos, and D. Secureworks, "P2PWNED: Modeling and Evaluating the Resilience of Peer-to-Peer Botnets," in *IEEE Symposium on Security & Privacy*. IEEE, 2013.

[4] CERT Polska, "Zeus-P2P monitoring and analysis," Tech. Rep., 2013.

[5] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, "Your Botnet is My Botnet: Analysis of a Botnet Takeover," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 635–647.

[6] Threatpost.com, "Latest Kelihos Botnet Shut Down Live at RSA Conference 2013," 2013. [Online]. Available: http://threatpost.com/latest-kelihos-botnet-shut-down-live-rsa-conference-2013-022613/77567

[7] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling, "Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm." *LEET*, 2008.

[8] H. Salah and T. Strufe, "Capturing connectivity graphs of a large-scale p2p overlay network," in *Distributed Computing Systems Workshops (ICDCSW), 2013 IEEE 33rd International Conference on*, 2013.

[9] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulie, "Peer-to-peer membership management for gossip-based protocols," *IEEE Transactions on Computers*, vol. 52, no. 2, pp. 139–149, 2003. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1176982

[10] M. V. Steen, "Large-Scale Newscast Computing on the Internet," 2002.

[11] R. Bar-Yehuda and S. Even, "A local-ratio theorem for approximating the weighted vertex cover problem," *Annals of Discrete Mathematics*, 1985.

[12] D. Dittrich and S. Dietrich, "Discovery techniques for P2P botnets," *Stevens Institute of Technology CS Technical Report 2008*, vol. 4, no. September 2008, pp. 1–14, 2008.

[13] D. Stutzbach, R. Rejaie, and S. Sen, "Characterizing unstructured overlay topologies in modern P2P file-sharing systems," *Proceedings of the 5th ACM SIGCOMM conference on Internet measurement - IMC '05*, p. 1, 2005.

[14] Symantec.com, "Grappling with the ZeroAccess Botnet," 2013. [Online]. Available: http://www.symantec.com/connect/blogs/grappling-zeroaccess-botnet?SID=sh1k1i488abj\&cjid=6146953

[15] G. Yan, S. Chen, and S. Eidenbenz, "RatBot: anti-enumeration peer-to-peer botnets," *Lecture Notes in Computer Science*, vol. 7001, pp. 135–151, 2011.

[16] C. Kanich, K. Levchenko, and B. Enright, "The Heisenbot Uncertainty Problem: Challenges in Separating Bots from Chaff." *LEET*, 2008.

[17] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis, "My botnet is bigger than yours (maybe, better than yours): Why size estimates remain challenging," in *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*. USENIX Association, 2007, p. 5.

[18] A. Hagberg, P. Swart, and D. Chult, "Exploring network structure, dynamics, and function using NetworkX," no. SciPy, pp. 11–15, 2008.