BoobyTrap: On Autonomously Detecting and Characterizing Crawlers in P2P Botnets

*§Shankar Karuppayah, *Emmanouil Vasilomanolakis, *Steffen Haas, *Max Mühlhäuser, †Mathias Fischer

*Telecooperation Group
TU Darmstadt / CASED, Germany
firstname.lastname@cased.de

† Networking and Security Group
International Computer Science Institute, USA
mfischer@icsi.berkeley.edu

§ National Advanced IPv6 Center,
Universiti Sains Malaysia (USM),
Malaysia

***Abstract*—The ever-growing number of cyber attacks from botnets has made them one of the biggest threats on the Internet. Thus, it is crucial to study and analyze botnets, to take them down. For this, an extensive monitoring is a pre-requisite for preparing a botnet takedown, e.g., via a sinkholing attack. However, every new monitoring mechanism developed for botnets is usually tackled by the botmasters by introducing novel anti-monitoring countermeasures. In this paper, we anticipate these countermeasures by proposing a set of lightweight techniques for detecting the presence of crawlers in P2P botnets, called *BoobyTrap*. For that, we exploit botnet-specific protocol and design constraints. We evaluate the performance of our *BoobyTrap* mechanism on two real-world botnets: *Sality* and *ZeroAccess*. Our results indicate that we can distinguish many crawlers from benign bots. In fact, we discovered close to 10 crawler nodes within our observation period in the Sality botnet and around 120 in the ZeroAccess botnet. In addition, we also describe the observable characteristics of the detected crawlers and suggest crawler improvements for enabling monitoring in the presence of the BoobyTrap mechanism.**

I. Introduction

Nowadays, botnets are one of the biggest threats on the Internet. They can comprise hundreds of thousands of infected machines, so-called bots, and can be used for a variety of malicious activities. For example, they can carry out Distributed Denial of Service (DDoS) attacks, or they can be used for proxy-services and spams. Traditional cyber security countermeasures, e.g., Intrusion Detection Systems (IDSs) [13], cannot cope with the amount and the distributed nature of such attacks. In particular, the emergence of botnets exhibiting highly resilient Peer-to-Peer (P2P) architectures, renders monitoring and takedown attempts to be more difficult. Hence, to design and deploy countermeasures against such botnets, efficient monitoring mechanisms are required.

Botnet monitoring is a first and necessary step to gather the required knowledge for a potential takedown, which requires enumerating all participating bots. Monitoring can be conducted via the deployment of *sensor nodes* [6], by *crawling* the botnet [7], or by a combination of both techniques. Sensors are nodes that follow the botnet protocol and participate in the botnet overlay to gain knowledge on other bots. Hence, a sensor allows to enumerate the bots in the botnet but does not provide insights on the interconnectivity graph of the botnet. In contrast, crawling is more aggressive and uses graph traversal techniques to query nodes for their neighborhood connectivity, i.e., neighbor lists. Hence, this allows the reconstruction of the connectivity graph of a particular botnet.

The advent of P2P botnets, manifesting in an increasing number of attacks carried out by them, and their open nature currently attracts much attention from researchers and law enforcement agencies. As such, many researchers and organizations are independently monitoring such botnets and specifically conduct botnet crawling. However, what we currently experience in the botnet area is a classic race of arms. Every upgrade on the side of research and law enforcement is usually followed by another upgrade of the botnets by botmasters. Botnets are a valuable asset to their botmasters, which they will try to protect by all means. We saw already efforts to impede monitoring activities with the introduction of restricted information exchange in botnets like *P2P Zeus*, *Sality* or *ZeroAccess* that were answered from research side by improved monitoring mechanisms [7], [8]. Hence, the next escalation by the botmasters is only a step ahead and will probably come in the form of improved blacklisting mechanisms that might be able to detect the majority of currently used crawlers.

In this paper, we anticipate this next escalation step. The main contribution of this paper is a crawler detection mechanism that we call *BoobyTrap (BT)*. It enables bots, but also sensors, to autonomously detect the presence and activities of crawlers in P2P botnets. For that, we exploit botnet-specific protocol and design constraints that are often violated by crawlers. As a result and just by local observations, bots (or sensors) can easily blacklist crawlers to impede monitoring activities. We are aware that our BT mechanism can be exploited by botmasters. However, we argue that it be necessary to point out the vulnerabilities of current crawlers to raise the stakes so that the research community, as well as law enforcement agencies, are prepared and improve their monitoring mechanisms accordingly. For that, we also give recommendations for future crawlers at the end of this paper.

We also provide extensive evaluation results of our BT mechanism that were deployed on real-world sensor nodes participating in the Sality and ZeroAccess botnets. Our results indicate that we can successfully distinguish between crawlers and bots. Within our observation period, we also successfully detected a high number of crawlers. In addition, we also present a brief analysis of the crawlers' characteristics.

The remaining of this paper is structured as follows. In

Section II, we describe the general operations of P2P botnets. Section III provides a comprehensive description of our BT approaches. In Section IV, we evaluate our approach in a real-world scenario for two well-known botnets and present our analysis of the detected crawlers. In Section V we discuss the related work with a focus on crawlers as well as their detection in botnet environments. Finally, Section VI concludes this paper and proposes future work.

II. Botnet Membership Management Mechanism

Like traditional P2P networks, P2P-based botnets experience node churn, i.e., nodes joining and leaving the network at high frequency. Botnets usually deploy a *Membership Management (MM)* mechanism which ensures that participating bots remain connected to the botnet overlay to withstand churn. This mechanism is responsible for ensuring that unresponsive bots, e.g., offline bots, are removed from a particular bot's neighborlist NL and replaced by more responsive ones.

The maintenance of the NL usually follows a periodic *MM maintenance cycle* that is botnet-specific and can be in between 1 second to 40 minutes in some botnets. During such a cycle, a bot would iterate through the entries in its NL and probe each neighbor with a special message for its responsiveness. Bots are removed if they are unresponsive for several successive probing messages or *MM*-cycles. Bots are also able to request the NL of their neighbors to increase the size of their own NLs. Most botnets utilize a dedicated port to listen for incoming probe messages from other bots, i.e., the server socket, and either a dedicated or a combination of several dynamic ports to probe peers in the NL, i.e., the client socket(s).

Newly joined bots announce their presence in the botnet by an announcement message to notify existing bots to propagate information of the new node, i.e., when the NL is requested. In the following, we describe the specific *MM* of Sality and ZeroAccess based on our work on reverse engineering the most recent binaries of the respective botnet.

A. Sality

Each bot in Sality maintains an NL with at maximum 1000 entries and utilizes a membership maintenance interval of 40 minutes. During this maintenance cycle, a bot probes all neighbors in its NL using a *Hello* message. Then, the entries in the NL are marked either as *online* or *offline* based on their responsiveness. Bots also leverage this process to retrieve newer configuration updates issued by the botmasters (if available) by indicating the sequence number of the most recent update known to them [4]. If a bot receives a *Hello* message with an older sequence number, the bot replies the message by appending the latest update.

Furthermore, if the number of entries in the NL is below 980 at the beginning of a maintenance cycle, a *Neighbor List Request* message (NL_{Req}) is sent to all responsive neighbors. Bots receiving a NL_{Req} will respond with a *Neighbor List Reply* (NL_{Rep}) message containing information on one randomly picked bot from its NL, i.e., IP address and Port number, which was marked as *online* in the last *MM*-cycle. The returned entry is then processed according to Algorithm 1.

First, the bot checks if the current neighbor list is already full (Line 1) and returns if this is the case. Otherwise the algorithm proceeds and checks the IP address in the returned reply (Line 4). If the address is unknown, the returned entry is added to the NL (Line 9). In case the IP address is already known, it is checked whether the corresponding port in the reply matches with the existing entry in the NL (Line 5). If the ports do not match, the bot additionally checks (Line 6) if the entry in the NL was marked *offline* during the last membership maintenance cycle. Sality allows an existing neighbor, i.e., based on the IP address, to rejoin the botnet with a different port. Nevertheless, an existing entry will only be updated (Line 7) if the existing entry was already marked as *offline* in the NL.

Also, take note that a bot in Sality uses a new socket/port for every request message that needs to be sent. We believe this design is adopted to prevent researchers from conducting replay attacks or spoofing replies to manipulate entries in the *NL*.

```
1  if NL.getSize() >= 1000 then
2  |  return // Neighborlist is full
3  end
4  if entry.IP ∈ NL.getAllIPs() then
5  |  if NL.getPort(entry.IP) ≠ entry.Port then
6  |  |  if NL.getStatus(entry.IP) == Offline then
7  |  |  |  NL.updatePort(entry.IP, entry.Port)
8  else
   |  // NL not full and IP is unknown.
   |     Add new entry.
9  |  NL.add(entry)
10 end
```

Algorithm 1: $ProcessNL_{Rep}(entry)$

B. ZeroAccess

Bots in ZeroAccess maintain three types of NLs. The *primary* list consists of a maximum of 256 entries and the other two consist of more than 16 million entries each. Figure 1 depicts the sequence of messages exchanged between a Bot_X and Bot_Y in a *MM*-cycle. ZeroAccess's membership maintenance is carried out every 256 second in which it sequentially probes one entry from the primary list every 1 second with a *getL* message. Bots that successfully respond to the *getL* message are placed at the top of the primary NL. The position of a bot in the list indicates how recently it has responded to a probe message compared to other bots in the list. Bots need to respond to each of the probe messages with a *retL* message that includes the most recent 16 entries, i.e, first 16, from the primary NL along with information of *plugins* [4] that are available to be downloaded from it. These plugins contain additional features for the bots, e.g., Bitcoin mining or ad-clicking.

As depicted in Figure 1, a responsive neighbor Bot_Y is also expected to send a *getL+* message to the server port of the probing bot, i.e., Bot_X. This exchange is used to determine if Bot_X is publicly reachable by other bots. If this exchange is successful, Bot_X is placed at the top of Bot_Y's NL. Note that ZeroAccess has only two types of messages: *getL* and *retL*, both with a parameter *flag* which has the default value of zero, i.e., $flag = 0$. Meanwhile, the *getL+* and *retL+* messages are both of the same packet structure but with a different *flag* value, i.e., $flag = 1$. The specific behavior of a bot receiving a *getL*

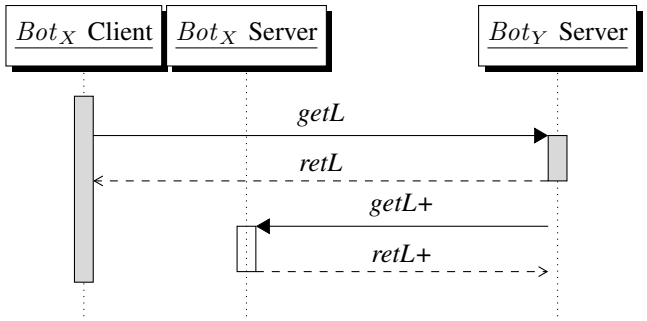

Fig. 1. Message exchange for Bot_X probing Bot_Y in ZeroAccess.

or *getL+* message is depicted in Algorithm 2. After receiving a request, a reply is always sent to the sender (Line 3) with the *flag* copied from the received request message (Line 2). As such, if a bot received a *getL* message with $flag = 0$, it replies with a *retL* message and additionally sends an additional *getL* message with the $flag = 1$, i.e., *getL+*.

```
1 // Reply to all received requests
2 rep ← createRetL(msg.getFlag())
3 send(sender, rep) // Send a retL to sender
4 if msg.getFlag() == 0 then
5 |   // Send a getL+
6 |   send(sender, createGetL(flag = 1))
7 end
```

Algorithm 2: $Process\_GetL(sender, msg)$

It is also worth noting that UDP hole punching can be performed in this botnet as reported by Rossow et al. [11]. Due to space constraints, we briefly describe them in the following. UDP hole punching can be performed in this botnet by continuously sending valid *requests*, e.g., *getL* or *getL+* messages, to the client socket of a Bot_X in addition to the *retL* and *retL+* messages. The additional requests will be processed by the bots as though they were received from the server socket. As such, once a communication channel is established from bots behind NAT-like devices, it is possible to keep communicating to them by sending valid requests periodically.

III. BoobyTrap

In this section, we first summarize our system design and our assumptions, before we introduce our generic crawler detection mechanism called *BT*. As this mechanism needs to be specifically tailored for each botnet, we provide tailored BT nodes for the Sality and ZeroAccess botnet as examples.

A. System Design & Assumptions

A BT node is essentially a sensor node that is enriched with additional detection mechanisms or '*traps*' to identify misbehaving nodes that are contacting it in an autonomous manner. All communication with the BT node is logged in a relational database for future reference along with additional metadata, i.e., timestamp, payload, source IP, and source port. Compared to conventional sensors, BT nodes can have additional functionalities such as responding to NL requests with valid replies and (re)sending valid probe messages to the sender of a request message. In addition, BT nodes can also listen for incoming connections or messages on a secondary port (if applicable), as it is needed for one of the traps (cf. Section III-C). Take note that BT nodes only return non-bot entries, i.e., other sensor nodes, to avoid participating in the botnet's maintenance activities due to legal constraints. Therefore, BT-enhanced sensor nodes do not participate in any activities that may, directly or indirectly, aid the botnet in any manner.

The BT detection mechanisms leverage upon the following assumptions of crawlers that are derived from our observations in real-world botnets and requirements from a legal perspective.

1) Crawlers greedily attempt to discover/contact all online bots as much as possible by aggressively abusing the botnet's protocol to request NL.
2) Crawlers are not able to distinguish BT nodes from normal bot, without first interacting with them.
3) Crawlers do not aid the botnet in any manner, e.g., returning valid neighbors or updates.

B. Detection Classes

The main idea of BT is to identify misbehaving nodes, i.e., crawlers, by distinguishing their behavior from benign bots by violations of the respective botnet's *MM* protocol. These behaviors can be categorized according to the following classes: *Defiance*, *Abuse*, and *Avoidance*, that are easily adaptable to any P2P botnets. These classes are described in detailed in the following.

1) Defiance: Bots that defy the botnet-specified *MM* protocols can be classified as crawlers. Such behavior includes omitting certain prerequisite actions or mandatory message exchange(s) before requesting an NL. Moreover, in some cases, it is also possible to identify a crawler based on its behavior of greedily contacting *all* discovered entries, even when the botnet protocol applies some restriction to entries that can be chosen as potential neighbors. For example, new entries that have a matching $/20$ subnet with existing entries are ignored by P2P Zeus [2]. As such, if a *BT* node returns an entry of another BT node that is from the same $/20$ subnet, and if both BT nodes were contacted by an identical bot, this bot can be classified as a crawler.

2) Abuse: In P2P botnets, bots may request the NLs of their neighbors to add new neighbors to their own NL. This ability is necessary to prevent getting isolated from the botnet overlay. However, crawlers can make use of this NL exchange mechanism to reconstruct the network topology of the botnet. Therefore, bots in most recent P2P botnets return only a small subset of their NL to prevent a crawler from retrieving the entire NL easily [2]. Moreover, the presence of churn also encourages crawlers to obtain snapshots of the botnet as fast as possible to avoid introducing bias in the monitoring results [7]. Therefore, crawlers typically crawl bots with higher frequencies [8]. In contrast, bots usually probe their neighbors only once per *MM*-cycle, i.e., between 1 sec to 40 minutes, depending on the botnet in scrutiny. Thus, a frequency-based detection mechanism can be utilized to detect crawlers that abuse the NL exchange mechanism. In fact, such a countermeasure was already implemented by the P2P Zeus botnet [2].

3) Avoidance: A botnet' *MM* protocol specifies the order of message exchange as well as the structure of the messages. Hence, nodes that do not respond to responsiveness probing messages or return useless responses, e.g., empty or invalid, could be crawlers. This behavior can be due to the design of most monitoring nodes which is minimalistic and does not aim at aiding the botnet in its day-to-day operations, i.e., command dissemination, attack execution, or management of the overlay. As such, crawlers tend to avoid providing information or responding to requests that may aid the botnet in a positive manner. By deliberately sending botnet-specific requests that should generate a verifiable reply, bots that are refusing to respond (or ignore the requests) can be classified as crawlers.

C. BoobyTrap for Sality

In the following, three crawler detection mechanisms or *traps* are adapted for Sality from the different misbehavior classes presented in Section III-B. Each trap's name is prefixed by an abbreviation of the botnet's name and the respective detection class. For Sality, there are two traps in the class of *Defiance*, i.e., *SD1-IgnoreTrap* and *SD2-BaitTrap*, and one trap from the class of *Abuse*, i.e., *SA1-BurstTrap*. Take note that a trap can also be setup for this botnet from the class of *Avoidance* based on the configuration update mechanism that leverages upon the *Hello* message exchange (cf. Section II-A) to identify nodes that refuse to send the latest update. However, such a trap for Sality is omitted in this work as it may cause DDoS on the BT node itself due to the heavy traffic caused by the attached updates in the responses [10].

1) SD1-IgnoreTrap: The *MM* protocol of Sality specifies that a bot utilizes a *Hello* message to probe the responsiveness of its neighbors (cf. Section II-A). If the neighbors are responsive, and the probing bot requires additional neighbors, only then, it sends an additional NL_{Req} message. As such, a NL_{Req} is *always* preceded by a *Hello* message. Crawlers that want to simplify this process to reduce the communication overhead for crawling may decide to ignore the *Hello* message and send only NL_{Req} messages to the bots. In addition, simplifying the process also reduces the amount of time needed for the crawler to produce a snapshot. For each received NL request, the BT node checks if there has been a preceding *Hello* message logged in the database. If the database contains no records for the *Hello* message, the respective node is flagged as a crawler.

2) SD2-BaitTrap: Sality ensures that an IP address can only be present once in a bot's neighbor list (Line 4, Algorithm 1). When a bot discovers a potential neighbor with the same IP but different port (Line 6, Algorithm 1), it prefers an existing and responsive entry, i.e., IP address. This trap exploits this observation by deliberately responding to all received NL_{Req} with an entry that points back to the BT node's *secondary port*, i.e., that is also monitored for incoming requests. Legitimate bots would ignore such a reply, since the initial entry, i.e., the entry with the primary port, is still responsive in the NL from the previous *MM*-cycle. Since crawlers are often greedy in obtaining information on the botnet topology, the crawlers could also probe the secondary port of BT and trigger the crawler detection mechanism. Take note that this particular *baiting* mechanism can also be executed using two (or more) colluding BT nodes.

3) SA1-BurstTrap: Bots in Sality probe the responsiveness of their neighbors once every 40 minutes (cf. Section II-A). In addition, bots can (optionally) request neighbors of their neighbor by sending a NL_{Req}. As such, this trap keeps track of a bot's NL requesting frequency, i.e., based on the IP address of the requester. If a bot sends several NL request, i.e., > 1, within a short interval, i.e., < 40 minutes, this behavior triggers the detection mechanism.

D. BoobyTrap for ZeroAccess

In the following, three crawler detection mechanisms or *traps* are adapted for ZeroAccess from the different misbehavior classes presented in Section III-B. Each trap's name is prefixed by an abbreviation of the botnet's name and the respective detection class. Due to the simplicity of ZeroAccess' design to use only two types of messages and a fixed port for inter-communication, a trap within the *Defiance* class is not feasible to be implemented for this botnet.

1) ZD1-NonComplianceTrap: The *MM* protocol of ZeroAccess as described in Algorithm 2 allows bots to identify that a *getL* message was received by checking the flag value of the received request message (Line 4), i.e., $flag == 0$. In reply, bots always send a *getL+* message that has its flag set to 1. However, it is still protocol-compliant if a bot sends a *getL+* message with the flag set to any non-zero integers, i.e., $flag \neq 0$. This trap deliberately sends a *getL+* with a modified flag-value, e.g., $flag = 3$, for every received *getL* message. A legitimate bot will answer all received requests with a *retL* or *retL+* message that has the *flag* values *copied* from the received request messages (Line 2). In addition, due to the possibility of UDP hole punching in ZeroAccess (c.f. Section II-B), all legitimate bots (including those behind NAT-like devices) should respond to any *getL+* message received. Hence, the BT node examines whether the received replies contain inconsistent or modified flags. Therefore, any crawlers that are non-compliant to the *MM* protocol, i.e., not copying the flag value received in the reply, will be flagged accordingly.

2) ZA1-BurstTrap: Based on our own work of reverse-engineering the recent variants of ZeroAccess, we found that a bot would normally contact a particular neighbor at most three times within a duration of 256 seconds in a *MM*-cycle. The only exception is during the initial phase of bootstrapping into the overlay at which point a bot could contact a bot more frequently. The general observation is exploited in this trap which is triggered if any bot attempts to communicate with the BT node aggressively in quick successions, i.e., more than three requests within 256 seconds. Similar to *SA1-BurstTrap*, if more than three requests are received from a single bot within a short interval, i.e., < 256 seconds, our detection mechanism is triggered.

3) ZA2-IgnoreTrap: This particular trap works in combination with the *ZD1-NonComplianceTrap*. Crawlers that received our BT node's *getL+* requests with modified flag values may (intentionally or unintentionally) decide not to respond to the message. Considering the fact that UDP hole punching is exploited, all bots including those behind NAT are expected to respond to the received requests. Therefore, any node deliberately refusing to reply can be flagged as a crawler.

IV. Evaluation

In this section, we describe the details of the used datasets and explain our experimental setup. Furthermore, we present the evaluation results for our BT mechanism that were deployed in two existing real-world botnets, i.e., Sality and ZeroAccess, at the end of the section by answering the following research questions:

1) What are the appropriate threshold values to minimize false positives generated by bots behind NAT and proxies in frequency-based detection mechanisms?
2) How susceptible are current crawlers against the BT detection mechanisms?
3) What are the common characteristics exhibited by existing crawlers in the wild?

A. Data Sets

We implemented the BTs, as described in Section III, on sensor nodes and deployed them in the Sality *Version 3* [4] and ZeroAccess *Network 2* (port 16470) [9] botnets. Each of our BT nodes was popularized for a period of two weeks before we conducted measurements for one week in total in each botnet: Sality (23/09/2015 00:00:00 CET to 29/09/2015 23:59:99 CET) and ZeroAccess (02/10/2015 15:57:55 CET to 09/10/2015 15:57:54). The summary of the datasets is provided in Table I.

	Sality (Version 3)	***ZeroAccess (Port 16470)***
Total IPs	735,443	25,236
Average IPs/day	162,804	7,128
Min IPs/day	155,957	5,905
Max IPs/day	177,267	7,864

TABLE I. Description of the datasets

B. Experimental Setup

We conducted our analysis using sensors that were implemented in the *Python* language for both botnets, i.e., Sality and ZeroAccess. All of our detection mechanisms are triggered by the type and contents of the received (or missing) responses from a particular node. However, for the frequency-based detection mechanisms, i.e., *Abuse* class, a configurable sliding window-based detection mechanism is implemented to identify IPs aggressively contacting our BTs. This detection mechanism takes two input parameters: the length of the sliding window t (in seconds) and the minimum number of messages $n_{\min}$ to trigger the detection mechanism. A detection is triggered if a particular IP sends more than $n_{\min}$ messages within any sliding window observations.

To evaluate the performance of our proposed mechanisms, we consider the amount of IPs that triggered the BTs and then attempt to verify manually if the behavior of a node behind an IP is indeed a crawler. Since manual checking cannot always yield a binary answer, the IPs are classified on a best-effort basis using the following classifications: 1) *Highly Possible*, 2) *Possible*, 3) *Unknown*, and 4) *False Positive*. A node is classified as *Highly Possible* when there is significant evidence that resembles a crawler's behavior, e.g., avoiding to exchange information. A node is classified as *Possible* when there is evidence that (almost) equally resembles as both a possible crawler and a bot. Meanwhile, a node is classified as *Unknown* when the available evidence is not helpful to make any conclusion. Finally, a node is classified as *False Positive* when logs only indicate the behaviors of a benign bot. In such cases, an explanation of why those bots were initially flagged is also provided. In some cases, we were also able to identify the organizations behind the detected crawling activities. Nevertheless, we do not disclose any details regarding this to prevent targeted attacks from the botmasters.

First, we conduct an analysis to identify the best threshold values for the parameters t and $n_{\min}$ in our frequency-based detection mechanisms, i.e., *SA1-BurstTrap* and *ZA1-BurstTrap*, for both botnets. These threshold values are important to minimize the false positives that may occur due to bots behind NAT and proxy-like devices. Then, we evaluate the performance of the detection mechanisms described in Section III-C and III-D. Finally, we highlight the common characteristics exhibited by the crawlers that were detected by our detection mechanisms.

C. Results

We first investigated on the best threshold values for the frequency-based detection mechanisms. For that, we conducted an analysis of the various combination of parameters of t and $n_{\min}$. Specifically, we varied the sliding window interval, i.e., $60, 120, \ldots, 2400$ seconds, and repeated the experiments with different number of minimum messages required to trigger a detection, i.e., $10, 20, \ldots, 100$ requests, for Sality. Due to space constraints, the results of the complete analysis is summarized in this paper. Our analysis indicated that Sality's BT performs best with the parameters $t = 120$ and $n_{\min} = 30$. The analysis was also repeated in a similar manner for ZeroAccess, and the results indicated that the best parameters are $t = 60$ and $n_{\min} = 40$. We speculate that the higher number of messages required for ZeroAccess in comparison to Sality can be due to the short *MM*-cycle interval of this botnet, i.e., 256 seconds.

Next, we evaluate the performance of all detection mechanisms in both botnets within the measurement period. The overall results are summarized and presented in Table II according to the respective misbehavior classes (cf. Section III-B).

	Defiance			**Abuse**		**Avoidance**
	SD1	SD2	ZD1	SA1	ZA1	ZA2
Detected IPs	4,212	3	88	11	188	108
After Sanitization	966	-	-	-	-	-
Highly Possible	4	3	7	9	116	35
Possible	-	-	81	1	72	73
Unknown	962	-	-	-	-	-
False Positives	3,246	-	-	1	-	-

TABLE II. Performance of the *BoobyTrap* mechanisms

For the class of *Defiance*, two BTs were set up for Sality (*SD1* and *SD2*) and one for ZeroAccess (*ZD1*). The *SD2-BaitTrap* for Sality was least often triggered by crawlers. However, this particular trap is also the most obvious indicator for a crawler as benign bots in Sality would simply ignore entries that are already known and responsive, i.e., a benign bot would ignore the entry of the BT's secondary port as long as the entry with primary port is still responsive. The *SD1-IgnoreTrap* was triggered by $4,212$ IPs throughout the week, which seems abnormally high compared to other traps. Detailed analysis of the results indicates that many of the flagged IPs are behind ISPs that use multiple NAT IPs or load balancing configurations. Since each request in Sality is sent from a new port (cf. Section II-A), NAT devices assume that a new flow or connection is being established and may decide to

route the packet using a different proxy or NAT IP as a load balancing technique. As such, our BT node recorded *Hello* messages from a different IP than the one received for the NL_{Req}, thus triggering the trap. We looked into these cases and sanitized them by correlating a Sality-specific identifier. As a result, we identified 77% of the detected IPs to be false positives. Out of the remaining 966 IPs, only four IPs exhibited strong indication as crawlers. The remaining 962 IPs could not be reliably classified as their identifiers were set to Sality's default identifier. Hence, they were classified as *Unknown*.

The *ZD1-NonComplianceTrap* was triggered by 88 IPs. Seven of those IPs, which were detected on the first day, consistently responded to us with a *retL* message containing a fixed flag, i.e., $flag = 0$. These IPs are particularly interesting because they responded with exactly 65 messages before they stopped to contact our BT sensor. We suspect that these are crawlers that exhibit a blacklisting mechanism to avoid crawling or contacting rogue nodes, i.e., other sensors. We verified that those IPs kept contacting another instance of our sensor node after our BT sensor was (presumably) blacklisted. The remaining IPs responded with a flag value set to either 0 or 1 up to a maximum of five replies. As we did not observe such behavior from the malware variants we reverse-engineered, we have no explanation for this other than this could be a crawler behavior.

Our evaluation on the BTs within the class of *Abuse* was conducted based on the frequency of received *NL* request messages for both botnets. The parameters of the BTs were set according to our previous parameter study: Sality ($t = 120$, $n_{\min} = 30$) and ZeroAccess ($t = 60$, $n_{\min} = 40$). Evaluation results indicated 11 flagged IPs by the *SA1-BurstTrap*. Out of the 11, nine IPs were classified as *Highly Possible*. A daily analysis of this particular BT as presented in Figure 2 indicates that an average of four crawlers is successfully identified every day. Meanwhile, false positives were caused by many bots behind a single shared IP coincidentally contacting our BT node around the same time.

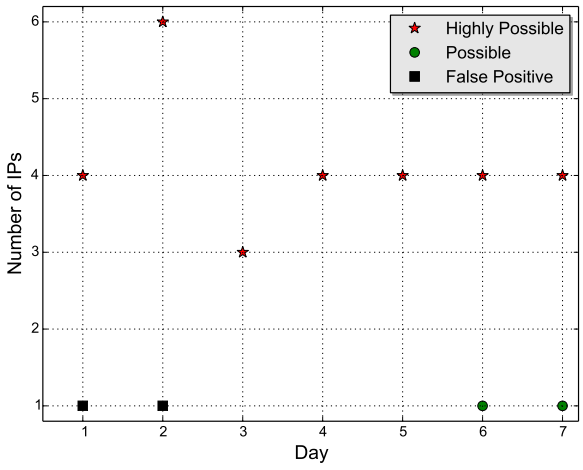

Fig. 2. Daily analysis of the *SA1-BurstTrap*

The *ZA1-BurstTrap* flagged a total of 188 IP addresses throughout the measurement period. After manual inspection, we were able to classify 116 IPs as strongly exhibiting crawler-like behavior. However, the remaining IPs were classified as *Possible*. These IPs exhibited similar behavior to benign bots that lack responsive neighbors in their *NL* during their initial bootstrapping phase. This speculation could especially be true considering that a large portion of the ZeroAccess botnet was sinkholed in 2013 [9]. As such, bots that experience a lack of neighbors could have requested *NL*s with a higher frequency.

The *ZA2-IgnoreTrap*, as an avoidance trap, within the class of *Avoidance* attempts to identify crawlers that are refusing to respond to the *crafted* requests sent in the *ZD1-NonComplianceTrap*. Our evaluation of this BT indicates 108 IPs in our ZeroAccess dataset that never responded to any of our request messages. In more details, 35 IPs were classified as *Highly Possible* crawlers because the BT node recorded abnormally high number of received *getL* requests, i.e., between 10 and $14,800$, but without any replies for any of our crafted request messages. Meanwhile, 73 IPs were classified as *Possible* crawlers. These IPs seemed to be shared among many bots, i.e., identified by distinct botnet-specific identifiers, but originate only from selected network prefixes. An alternative hypothesis for this observation can be explained if there is any packet-level filtering mechanism deployed within those networks that drops all inbound ZeroAccess' requests or replies. Such a scenario could result in the BT node observing the behavior of bots *refusing* to respond.

Finally, we analyzed the various crawlers that were detected and attempted to find the common characteristics exhibited by them. Our observations indicate that some of the crawlers are using blacklisting mechanisms to improve the quality of their crawl data, i.e., ignore rogue nodes. In addition, some crawlers also seemed to perform sanity-checking on the results obtained from bots. Meanwhile, detection mechanisms within the class of *Defiance* detected a smaller fraction of crawlers than those within the class of *Abuse*. Hence, it seems that there are crawlers that follow the implementation of the protocols very closely in the botnets. Upon analyzing crawlers detected via the frequency-based mechanism, we noticed that only a minority of crawlers were observed to crawl the botnets continuously, i.e., 24x7. Nevertheless, some of these crawlers communicated aggressively with our BT node with a high frequency, i.e., in average 15 requests per minute in Sality.

Some crawlers were also observed to utilize identical botnet-specific identifiers and port numbers across different instances in a given botnet. This observation may indicate identical crawler implementations deployed as redundant crawlers or to obtain additional vantage points for crawling. Some of the crawlers also use a dedicated port to process incoming NL replies. This can improve the crawling efficiency as it allows the processing thread to be independent of the thread that sends the requests. Moreover, we also noticed that by performing *WHOIS* queries on the detected IPs, only a few of the responses disclose information about the organization or individual that is behind the crawling activities. In fact, some crawlers have been seen sharing IP addresses with benign bots, i.e., behind NAT devices. In addition, IPs of some of these crawlers were also constantly changing due to dynamic IP address reallocation by ISPs. Such scenarios will make it more difficult to detect crawlers via any frequency-based detection mechanisms. Finally, based on the crawlers identified in this paper, we could conclude that existing crawlers do not seem to aid the botnets in any manner. Even in cases where neighbors are returned, these were either other sensors or simply invalid entries.

V. Related Work

In this section, we first summarize related work on botnet crawlers, followed by proposals on detecting botnet crawlers and existing botnet anti-monitoring countermeasures. Finally, we discuss existing techniques used to eliminate bias in monitoring data resulting from activities of other researchers.

Botnet crawlers aim at the enumeration of all bots in a botnet as well as on retrieving a graph that reflects the neighborhood relationships of bots [7]. Crawlers usually start with a seed peer and iteratively request neighbor lists from all discovered bots and terminate when there are no more new bots to be discovered. The crawlers are usually implemented based on graph traversal techniques such as Breadth-First Search (BFS) [5], [11] and Depth-First Search (DFS) [3]. Since P2P botnets exhibit high churn dynamics [12], crawlers need to be very fast and efficient to produce high-quality data. In addition, to ensure the completeness of the data, the crawlers need to conduct multiple consecutive crawls on the botnet and aggregate the respective information [14].

This observable crawler behavior, i.e., aggressively requesting neighbor lists, can be easily distinguished from regular bots. Hence, many botnets nowadays utilize techniques to avoid or slow down crawling. For instance, the *P2P Zeus* botnet implements an anti-crawling mechanism that blacklists suspected crawlers [11]. The bots blacklist any IP address that aggressively tries to contact them, i.e., more than six messages within a sliding window of one minute.

More recently, Andriesse et al. discussed the detection of crawlers in the P2P Zeus and Sality botnets by focusing on identifying anomalies in the implementation of the crawlers [1]. One of their techniques utilizes some sensor nodes to correlate their gained information on which nodes contacted them and in which frequency, which allows them to disclose crawlers. However, the authors admit that their method is not applicable to the Sality botnet because it would require hundreds of sensor nodes to have a chance to identify a crawler. In contrast, we present techniques to detect such crawlers in Sality and ZeroAccess in an autonomous manner which do not presume collaborating sensor nodes or heavy computing resources. Instead, single bots or sensors can identify crawlers independent from each other and with local knowledge only.

VI. Conclusion

In this work, we discuss the need for more advanced botnet monitoring mechanisms in anticipation of even more sophisticated botnets in the future. For that, we introduce autonomous detection mechanisms for crawlers in P2P botnets that we call *BoobyTraps*. We demonstrated that we can easily detect current crawlers by exploiting botnet-specific protocols. During our one week measurement period, we managed to identify close to 10 IPs in the Sality and around 120 IPs in the ZeroAccess botnet that are highly possible crawlers. From our analysis, we also concluded that a frequency-based detection mechanism seems to perform best in distinguishing crawlers from benign bots.

Furthermore, we also presented a brief overview of the various characteristics observed on the detected crawlers. Particularly, we found that most crawlers exhibit common characteristics especially regarding aggressively requesting *NL*s from bots. On the lessons learned from observing these characteristics, we propose that future crawlers need to follow the botnet protocols closely, even though this may limit their crawling frequency. Finally, to evade frequency-based detection mechanisms and to still obtain accurate snapshots, we suggest that crawling activities should be carried out in a distributed manner, i.e., by using multiple IPs from different ISPs. In addition, researchers should also ensure that the used IPs do not reveal the identity of the organization behind the crawling activities, e.g., via publicly available online databases like the WHOIS services. For future work, we aim on investigating the impact on the performance of crawlers in trying to evade our BT mechanisms. This investigation would allow us to evaluate how well protocol-abiding crawlers can perform in the presence of advanced anti-crawling mechanisms like BT.

VII. Acknowledgments

This work has been co-funded by the DFG as part of Project S1 within the CRC 1119 CROSSING.

References

[1] D. Andriesse, C. Rossow, and H. Bos, "Reliable Recon in Adversarial Peer-to-Peer Botnets," in *ACM SIGCOMM Internet Measurement Conference (IMC)*, 2015.

[2] H. Andriesse, D.; Rossow, C.; Stone-Gross, B.; Plohmann, D.; Bos, "Highly resilient peer-to-peer botnets are here: An analysis of Gameover Zeus," in *Malicious and Unwanted Software: "The Americas" , 8th International Conference on*, 2013.

[3] D. Dittrich and S. Dietrich, "Discovery techniques for P2P botnets," *Stevens Institute of Technology CS Technical Report*, vol. 4, 2008.

[4] N. Falliere, "Sality: Story of a Peer-to-Peer Viral Network," Tech. Rep., 2011.

[5] 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.

[6] B. Kang, E. Chan-Tin, and C. Lee, "Towards complete node enumeration in a peer-to-peer botnet," *Proceedings of International Symposium on Information, Computer, and Communications Security (ASIACCS)*, 2009.

[7] S. Karuppayah, M. Fischer, C. Rossow, and M. Muhlhauser, "On Advanced Monitoring in Resilient and Unstructured P2P Botnets," in *IEEE International Conference on Communications (ICC)*, 2014.

[8] S. Karuppayah, S. Roos, C. Rossow, M. Mühlhäuser, and M. Fischer, "ZeusMilker: Circumventing the P2P Zeus Neighbor List Restriction Mechanism," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2015.

[9] A. Neville and R. Gibb, "ZeroAccess Indepth," *Symantec Security Response*, 2013.

[10] C. Rossow, "Amplification Hell: Revisiting Network Protocols for DDoS Abuse," in *Network and Distributed System Security Symposium*, 2014.

[11] 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*, 2013.

[12] D. Stutzbach, R. Rejaie, and S. Sen, "Characterizing unstructured overlay topologies in modern P2P file-sharing systems," *ACM SIGCOMM Internet Measurement Conference (IMC)*, 2005.

[13] E. Vasilomanolakis, S. Karuppayah, M. Mühlhäuser, and M. Fischer, "Taxonomy and Survey of Collaborative Intrusion Detection," *ACM Computing Surveys*, vol. 47, no. 4, 2015.

[14] B. Wang, Z. Li, H. Tu, Z. Hu, and J. Hu, "Actively Measuring Bots in Peer-to-Peer Networks," in *International Conference on Networks Security, Wireless Communications and Trusted Computing*, vol. 1, 2009.