

SHANKAR KARUPPAYAH

ADVANCED MONITORING IN P2P BOTNETS



# ADVANCED MONITORING IN P2P BOTNETS

SHANKAR KARUPPAYAH

Vom Fachbereich Informatik  
der Technischen Universität Darmstadt genehmigte

*Dissertation*

zur Erlangung des akademischen Grades

*Doctor rerum naturalium (Dr. rer. nat.)*

Eingereicht von:  
M.Sc. Shankar Karuppayah  
geboren in Georgetown, Penang

Erstreferent: Prof. Dr. Max Mühlhäuser (Technische Universität Darmstadt)

Koreferent: Prof. Dr. Vern Paxson (University of California, Berkeley)

Tag der Einreichung: 18. Mai 2016

Tag der Prüfung: 01. Juni 2016



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Fachgebiet Telekooperation  
Fachbereich Informatik  
Technische Universität Darmstadt  
Hochschulkennziffer D-17

Darmstadt 2016



## ACKNOWLEDGMENTS

---

This thesis would not have come into existence without the help and encouragement of colleagues, friends and family. My heartfelt gratitude and thanks go out to all of them.

First, I would like to express my sincere gratitude to my advisor Max Mühlhäuser for the continuous support of my PhD study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my PhD study. Second, I am grateful to Vern Paxson for acting as a second referee. Third, I am indebted to the supervision of Mathias Fischer since the beginning of my PhD study. The various discussions, experiences and guidance I received from him are definitely unforgettable.

I would like to thank all my colleagues and students at the Telecooperation Group of TU Darmstadt, who have all shared wonderful memories with me, especially all of those at the areas of Smart Security Infrastructures and Smart Security and Trust: Leon Böck, Jörg Daubert, Carlos Garcia, Tim Grube, Sheikh Mahbub Habib, Steffen Haas, Sascha Hauke, Stefan Schiffner, Emmanouil Vasilomanolakis, Florian Volk. They, and everybody else at TK, made this phase of my life a pleasure. Not forgetting, a great thank you to the great staff at the group: Elke Halla and Fabian Herrlich among others, for their patience and cooperation in handling me throughout this three and a half year.

Special thanks to my MIG family that has always been proud of my achievements. Particularly, Mr. Sivasuriyamoorthy Sundara Raja and Mrs. Dhamayanthi Singaram (Selvi Akka) for being both a strong emotional support and a mentor for me. Also not forgetting, Aroon Kumar Mathivanan whom has always been there whenever I am in need of a break from my stressful days.

My biggest thanks definitely goes to my future wife, Prevathe Poniah. You have tolerated my physical and psychological absence throughout these years. Thank you for being my support when I needed it the most. I could not have done it without you.

Last but not the least, I would like to thank my family: my parents Mr. C. Karuppayah and Mrs. K. Anjama, my brothers and sisters-in-law for supporting me emotionally and spiritually throughout writing this thesis and my life in general. Without your supports, I would not have been successful.

This thesis would also not have been possible without the funding and support of Malaysian Ministry of Higher Education and Universiti Sains Malaysia.



## ABSTRACT

---

Botnets are increasingly being held responsible for most of the cyber-crimes that occur nowadays. They are used to carry out malicious activities like banking credential theft and Distributed Denial of Service (DDoS) attacks to generate profit for their owner, the botmaster. Traditional botnets utilized centralized and decentralized Command-and-Control Servers (C2s). However, recent botnets have been observed to prefer P2P-based architectures to overcome some of the drawbacks of the earlier architectures.

A P2P architecture allows botnets to become more resilient and robust against random node failures and targeted attacks. However, the distributed nature of such botnets requires the defenders, i.e., researchers and law enforcement agencies, to use specialized tools such as crawlers and sensor nodes to monitor them. In return to such monitoring, botmasters have introduced various countermeasures to impede botnet monitoring, e.g., automated blacklisting mechanisms.

The presence of anti-monitoring mechanisms not only render any gathered monitoring data to be inaccurate or incomplete, it may also adversely affect the success rate of botnet takedown attempts that rely upon such data. Most of the existing monitoring mechanisms identified from the related works only attempt to *tolerate* anti-monitoring mechanisms as much as possible, e.g., crawling bots with lower frequency. However, this might also introduce noise into the gathered data, e.g., due to the longer delay for crawling the botnet. This in turn may also reduce the quality of the data.

This dissertation addresses most of the major issues associated with monitoring in P2P botnets as described above. Specifically, it analyzes the anti-monitoring mechanisms of three existing P2P botnets:

1) *GameOver Zeus*, 2) *Sality*, and 3) *ZeroAccess*, and proposes countermeasures to circumvent some of them. In addition, this dissertation also proposes several advanced anti-monitoring mechanisms from the perspective of a botmaster to anticipate future advancement of the botnets. This includes a set of lightweight crawler detection mechanisms as well as several novel mechanisms to detect sensor nodes deployed in P2P botnets. To ensure that the defenders do not lose this arms race, this dissertation also includes countermeasures to circumvent the proposed anti-monitoring mechanisms. Finally, this dissertation also investigates if the presence of third party monitoring mechanisms, e.g., sensors, in botnets influences the overall churn measurements. In addition, churn models for *Sality* and *ZeroAccess* are also derived using fine-granularity churn measurements.

The works proposed in this dissertation have been evaluated using either real-world botnet datasets, i.e., that were gathered using crawlers and sensor nodes, or simulated datasets. Evaluation results indicate that most of the anti-monitoring mechanisms implemented by existing botnets can either be circumvented or tolerated to ob-

tain monitoring data with a better quality. However, many crawlers and sensor nodes in existing botnets are found vulnerable to the anti-monitoring mechanisms that are proposed from the perspective of a botmaster in this dissertation. Analysis of the fine-grained churn measurements for Sality and ZeroAccess indicate that churn in these botnets are similar to that of regular P2P file-sharing networks like Gnutella and BitTorrent. In addition, the presence of highly responsive sensor nodes in the botnets are found not influencing the overall churn measurements. This is mainly due to low number of sensor nodes currently deployed in the botnets. Existing and future botnet monitoring mechanisms should apply the findings of this dissertation to ensure high quality monitoring data, and to remain undetected from the bots or the botmasters.

## ZUSAMMENFASSUNG

---

Heute werden Botnetze zunehmen für die Mehrzahl der verübten Cyber-Straftaten verantwortlich gemacht. Die Besitzer der Botnetze, sogenannte Botmaster, nutzen die Netze um bösartige Aktivitäten wie beispielsweise den Diebstahl von Bankzugangsdaten und Distributed Denial of Service (DDOS) Angriffe durchzuführen. Ein Botmaster kontrolliert das eigene Botnetz mit einem Command-and-Control Server (C2) und verteilt über diesen Befehle und Aktualisierungen an die Bots. Traditionelle Botnetze verwendeten zentrale oder verteilte C2s. Allerdings zeigen Beobachtungen, dass aktuelle Botnetze mehr und mehr auf P2P-basierte Architekturen setzen und damit die Nachteile einer zentralen Architektur umgehen. P2P-basierte Botnetze sind widerstandsfähiger und robuster gegenüber zufälligen Ausfällen und gezielten Angriffen.

Bots in einem P2P Botnetz sind über ein Overlay miteinander verbunden. Dieses Overlay wird kollaborativ und verteilt von den Bots verwaltet. Diese verteilte Verwaltung erschwert die Überwachung und macht spezialisierte Überwachungslösungen wie Crawler und Sensorknoten nötig. Diese Überwachungslösungen setzen Wissen über das jeweilige Botnetzprotokoll und dessen Nachrichtenformat voraus um teilnehmende Bots und deren Kommunikationsbeziehungen zu bestimmen. Darüber hinaus setzen Botmaster diverse Gegenmaßnahmen ein, wie beispielsweise automatisches Blacklisting. Diese Gegenmaßnahmen führen nicht nur dazu, dass die gewonnenen Erkenntnisse unvollständig sind, sondern erschweren auch Angriffe auf das Botnetz selber (sogenannte Takedown-Angriffe). Der Großteil der verwandten Arbeiten im Feld der Botnetz-Überwachung versucht diese von Botnetzen initiierten Gegenmaßnahmen beispielsweise über ein langsames Crawling zu umgehen. Allerdings führen solche Ansätze aufgrund der höheren Verzögerungen im Crawling auch zu Rauschen in den gewonnenen Daten, was wiederum deren Qualität reduziert.

Diese Dissertation adressiert die meisten der oben genannten Herausforderungen in der Überwachung von Botnetzen. Im Detail werden Gegenmaßnahmen in den folgenden bekannten Botnetzen analysiert und mittels Verbesserung der Überwachungsmethoden umgangen: 1) GameOver Zeus, 2) Sality und 3) ZeroAccess. Darüber hinaus werden in dieser Dissertation mehrere fortschrittliche Gegenmaßnahmen aus der Perspektive der Botmaster vorgestellt, um zukünftigen Entwicklungen vorzugreifen. Zu diesen Maßnahmen zählt eine leichtgewichtige Crawler-Erkennung sowie neue Methoden zur Erkennung von Sensoren. Um aber solche Botnetze auch noch zukünftig beobachten zu können, beschreibt diese Dissertation auch Methoden um diese von Botnetzen intiierten Gegenmaßnahmen erneut zu umgehen. Weiterhin werden in dieser Dissertation auch die Auswirkungen von Überwachungsaktivitäten Dritter auf die über Überwa-

chungsmaßnahmen gewonnenen Daten, zum Beispiel Churn-Messungen, betrachtet.

Die in dieser Dissertation vorgeschlagenen Beiträge wurden mit echten Botnetz-Datensätzen evaluiert. Diese Datensätze wurden mit Crawlern und Sensoren in Botnetzen sowie durch Simulationen erhoben. Die Resultate deuten darauf hin, dass viele Crawler und Sensoren in Botnetzen für die vorgeschlagenen Gegenmaßnahmen auf Seiten der Botnetze anfällig sind. Daher sollten heutige und zukünftige Überwachungsmechanismen für Botnetze die Ergebnisse dieser Dissertation berücksichtigen, um weiterhin die Qualität der erhobenen Daten sicherzustellen und von Botmastern unentdeckt zu bleiben.

## PUBLICATIONS

---

Some parts of this dissertation have been published in peer-reviewed journal articles, and in the proceedings of international conferences and workshops.

- [Alo+12] Esraa Alomari, Selvakumar Manickam, B. B. Gupta, Shankar Karuppayah, and Rafeef Alfaris. “Botnet-based Distributed Denial of Service (DDoS) Attacks on Web Servers: Classification and Art.” In: *International Journal of Computer Applications* 49.7 (2012), pp. 24–32.
- [Böc+15] Leon Böck, Shankar Karuppayah, Tim Grube, Max Mühlhäuser, and Mathias Fischer. “Hide And Seek: Detecting Sensors In P2P Botnets.” In: *IEEE Conference on Communications and Network Security*. 2015, pp. 731–732.
- [Haa+16] Steffen Haas, Shankar Karuppayah, Selvakumar Manickam, Max Mühlhäuser, and Mathias Fischer. “On the Resilience of P2P-based Botnet Graphs.” In: *IEEE Conference on Communications and Network Security (CNS)*. 2016, (In Submission).
- [Kar+14] Shankar Karuppayah, Mathias Fischer, Christian Rossow, and Max Mühlhäuser. “On Advanced Monitoring in Resilient and Unstructured P2P Botnets.” In: *IEEE International Conference on Communications (ICC)*. 2014.
- [Kar+15] Shankar Karuppayah, Stefanie Roos, Christian Rossow, Max Mühlhäuser, and Mathias Fischer. “ZeusMilk: Circumventing the P2P Zeus Neighbor List Restriction Mechanism.” In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2015.
- [Kar+16a] Shankar Karuppayah, Emmanouil Vasilomanolakis, Steffen Haas, Max Mühlhäuser, and Mathias Fischer. “Booby-Trap: On Autonomously Detecting and Characterizing Crawlers in P2P Botnets.” In: *IEEE International Conference on Communications (ICC)*, (In Print). 2016.
- [Kar+16b] Shankar Karuppayah, Leon Böck, Tim Grube, Selvakumar Manickam, Max Mühlhäuser, and Mathias Fischer. “SensorBuster: On Identifying Sensor Nodes in P2P Botnets.” In: *ACM Conference on Computer and Communications Security (CCS)*. 2016, (In Submission).
- [Vas+13] Emmanouil Vasilomanolakis, Shankar Karuppayah, Mathias Fischer, Max Mühlhäuser, Mihai Plasoianu, Lars Pandikow, and Wulf Pfeiffer. “This Network is Infected : HosTaGe - a Low-Interaction Honeypot for Mobile Devices.” In: *Security and Privacy in Smartphones & Mobile Devices*. 2013, pp. 43–48.

- [Vas+14] Emmanouil Vasilomanolakis, Shankar Karuppayah, Max Mühlhäuser, and Mathias Fischer. “HosTaGe: a Mobile Honeypot for Collaborative Defense.” In: *International Conference on Security of Information and Networks*. 2014.
- [Vas+15a] Emmanouil Vasilomanolakis, Shankar Karuppayah, Panayotis Kikiras, and Max Mühlhäuser. “A honeypot-driven cyber incident monitor: lessons learned and steps ahead.” In: *International Conference on Security of Information and Networks*. 2015.
- [Vas+15b] Emmanouil Vasilomanolakis, Shankar Karuppayah, Max Mühlhäuser, and Mathias Fischer. “Taxonomy and Survey of Collaborative Intrusion Detection.” In: *ACM Computing Surveys* 47.4 (2015).

## CONTENTS

---

1	INTRODUCTION	1
1.1	Goal and Objective of Research	1
1.2	Contributions	2
1.3	Outline	3
2	BACKGROUND	5
2.1	Botnet Architectures	5
2.1.1	Centralized Botnets	6
2.1.2	Decentralized Botnets	6
2.1.3	P2P Botnets	7
2.2	P2P Botnet Monitoring	7
2.3	Characterizing Churn in P2P Networks	8
2.4	Chapter Summary	10
3	REQUIREMENTS AND STATE OF THE ART	11
3.1	Requirements of a Botnet Monitoring Mechanism	11
3.1.1	Functional Requirements	11
3.1.2	Non-Functional Requirements	13
3.2	Formal Model for P2P Botnets	14
3.3	Related Work on Botnet Monitoring	16
3.3.1	Honeypots	16
3.3.2	Crawlers	17
3.3.3	Sensor Nodes	20
3.4	Challenges in Botnet Monitoring	21
3.4.1	The Dynamic Nature of P2P Botnets	21
3.4.2	Noise from Unknown Third Party Monitoring Activities	23
3.4.3	Anti-Monitoring Mechanisms	24
3.5	Chapter Summary	29
4	THE ANATOMY OF P2P BOTNETS	31
4.1	Dissecting GameOver Zeus	31
4.1.1	Bootstrapping Process	32
4.1.2	Membership Maintenance Mechanism	32
4.1.3	Blacklisting Mechanism	35
4.2	Dissecting Sality	35
4.2.1	Bootstrapping Process	36
4.2.2	Membership Maintenance Mechanism	37
4.3	Dissecting ZeroAccess	39
4.3.1	Bootstrapping Process	41
4.3.2	Membership Maintenance Mechanism	42
4.4	Chapter Summary	44
5	CRAWLING BOTNETS	47
5.1	Circumventing Anti-Crawling Mechanisms	47
5.1.1	Restricted Neighborlist (NL) Reply Mechanism of GameOver Zeus	48
5.1.2	Less Invasive Crawling Algorithm (LICA)	56
5.2	Advanced Anti-Crawling Countermeasures	58

5.2.1	Enhancing GameOver Zeus' NL Restriction Mechanism	59
5.2.2	BoobyTrap: Detecting Persistent Crawlers	60
5.3	Evaluation	64
5.3.1	Evaluation of ZEUSMILKER	64
5.3.2	Evaluation of The Less Invasive Crawling Algorithm (LICA)	71
5.3.3	Evaluation of The BoobyTrap Mechanism	77
5.4	Chapter Summary	83
6	DEPLOYMENT OF SENSOR NODES IN BOTNETS	87
6.1	Detecting Sensor Nodes in Botnets	88
6.1.1	Introduction	88
6.1.2	Local Clustering Coefficient (LCC)	92
6.1.3	SensorRanker	94
6.1.4	SensorBuster	95
6.2	Circumventing Sensor Detection Mechanisms	95
6.2.1	Circumventing LCC	96
6.2.2	Evading SensorRanker	97
6.2.3	Evading SensorBuster	98
6.3	Evaluation	99
6.3.1	Datasets	99
6.3.2	Experimental Setup	100
6.3.3	Research Questions and Expectations	104
6.3.4	Results	105
6.4	Chapter Summary	113
7	UNDERSTANDING THE CHURN DYNAMICS IN P2P BOTNETS	115
7.1	Accurately Capturing Churn Dynamics in P2P Botnets	116
7.1.1	Strobo-Crawler	116
7.1.2	Adaptation for Sality	119
7.1.3	Adaptation for ZeroAccess	120
7.2	Evaluation	121
7.2.1	Datasets	121
7.2.2	Experimental Setup	122
7.2.3	Research Questions and Expectations	122
7.2.4	Results	123
7.3	Chapter Summary	131
8	CONCLUSION AND OUTLOOK	133
8.1	Conclusion	133
8.2	Outlook	135
A	ZEUSMILKER APPENDIX	139
A.1	Proof of Proposition 5.1.3	139
A.2	Probability of non-optimal performance	139
A.3	Proof of Proposition 5.1.4	139
	BIBLIOGRAPHY	143

## LIST OF FIGURES

---

Figure 1	Comparison of botnet architectures.	5
Figure 2	GameOver Zeus network connectivity graph between 23,196 nodes reconstructed via crawling. The blue dots indicate the nodes (systems infected with GameOver Zeus) and the green lines indicate the edges between nodes. (Source: Dell SecureWorks)	18
Figure 3	Message exchange for $Bot_X$ probing $Bot_Y$ in <i>ZeroAccess</i> .	42
Figure 4	Visual representation of the key space (Example 5.1.2). The '+' keys discover the next bigger key, whereas the '-' keys reveal the next smaller key.	50
Figure 5	Performance analysis of ZEUSMILKER, <i>Random</i> , and <i>BinaryHalving</i> on GameOver Zeus for various NL sizes $n$ and returned NL sizes $l$	68
Figure 6	Performance analysis of ZEUSMILKER, <i>Random</i> , and <i>BinaryHalving</i> for different key distributions in NLs ( $n = 50, l = 1$ )	69
Figure 7	Performance analysis of ZEUSMILKER, <i>Random</i> , and <i>BinaryHalving</i> on the presence of different advanced countermeasures ( $n = 50$ )	70
Figure 8	Performance analysis of LICA, <i>BFS</i> , and <i>DFS</i> . (a) The performance of LICA under different combination of parameters. The performance of all observed crawling algorithms on (b) GameOver Zeus and on (c) the <i>Gnutella</i> 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 GameOver Zeus dataset without any NL restrictions, plotted by the ratio of nodes discovered depending on the total number of nodes crawled.	74
Figure 9	Daily analysis of <i>SAB-BurstTrap</i>	81
Figure 10	LCC values of bots in Sality V3.	93
Figure 11	Extreme values of LCC can be used to identify sensors deployed within a botnet overlay, i.e., $lcc^+(x) = 0.0$ or $1.0$ .	93
Figure 12	Sensor nodes that do not return valid bots as neighbors would not be part of the <i>main</i> Strongly Connected Component (SCC).	96
Figure 13	A simple example of colluding sensors evading the LCC mechanism	97
Figure 14	Effectiveness of different clustering algorithms with $R = 0.0$ on accurately classifying sensors in the Sality dataset	107

Figure 15	Analysis of the influence of artifacts to the detection mechanisms with varying values of $R$ between 0.1 to 0.9 on the Sality dataset	109
Figure 16	Performance comparison of all detection mechanism with $R = 0.6$ on the Sality dataset	110
Figure 17	Classification of PR values by range of total predecessors in Day 1 of the Sality dataset	111
Figure 18	Estimation of <i>SensorRank</i> values for $S_i$ in dependence on number of colluding sensors	112
Figure 19	Strobo-Crawler Design Overview	117
Figure 20	Sequence Diagram for the Strobo-Crawler Framework	118
Figure 21	Distribution of online bots discovered during the measurement	122
Figure 22	Churn distributions for Sality and ZeroAccess	124
Figure 23	Joint session length distributions for Sality and ZeroAccess	126
Figure 24	Session Length Distributions of ZA_16464	127
Figure 25	Distribution Fitting in ZA_16471	130
Figure 26	Quantity in Eq. 2 for $\mathbf{b=160}$ bits	140

## LIST OF TABLES

---

Table 1	UIDs of existing P2P botnets that are retrievable from crawling [Ros+13].	23
Table 2	Summary of P2P botnet monitoring mechanisms regarding their compliance with the requirements specified in Section 3.1. Checkmark symbols ✓ indicate the fulfillment of these requirements, crossing symbols ✗ indicate their non-fulfillment, average symbols $\phi$ indicate a partial fulfillment, and dash symbols - indicate not applicable.	30
Table 3	Example of a GameOver Zeus bootstrap/neighborlist	32
Table 4	Example of a Sality bot's NL	36
Table 5	Distinct botnets distinguished by ports in ZeroAccess <i>Version 2</i>	40
Table 6	Example of a ZeroAccess <i>Primary</i> NL	41
Table 7	Graph properties of the datasets.	72
Table 8	Statistics of the collected data	77
Table 9	Performance of our <i>BoobyTrap</i> mechanism	80
Table 10	Summary of the sanitized datasets	100
Table 11	Summary of the selected snapshots	101
Table 12	Maximum sensors present on a particular day dependent on $R$ in the Sality dataset	106

Table 13	Maximum sensors present on a particular day dependent on R in the ZeroAccess dataset	106
Table 14	Performance comparison of all three sensor detection mechanisms	110
Table 15	Dataset Statistics	121
Table 16	Weibull Parameters; Tuples as (shape, 1/scale)	129



## INTRODUCTION

---

Cyber-crimes like banking fraud, spam campaigns, and denial-of-service attacks are a profitable business. Most of these attacks originate from botnets, a collection of vulnerable machines infected with malware that are being controlled by a botmaster via a Command-and-Control Server (C2). Traditional botnets utilize a centralized client-server architecture for the communication between the botmaster and its bots. Thus, after such a C2 is taken down, the botmaster cannot communicate with its bots anymore. Recent Peer-to-Peer (P2P)-based botnets, e.g., *GameOver Zeus* [And+13], *Sality* [Fal11], or *ZeroAccess* [Wyk12], adopt a distributed architecture and establish a communication overlay between participating bots. In fact, all (counter-)attacks against P2P-based botnets require detailed insights into the nature of these botnets, in particular the botnet population and the connectivity graph between the bots [Ros+13]. As a consequence, monitoring such botnets is an important task for analysts.

However, as P2P botnets represent valuable assets to their botmasters, botmasters are expected to impede the performance of monitoring mechanisms. This is already evident with the introduction of an automated blacklisting mechanism in *GameOver Zeus* and a local reputation mechanism in *Sality*. However, some of the deployed and of the proposed anti-monitoring mechanisms are still in their infancy and it will just be a matter of time before the botmasters will introduce more sophisticated mechanisms to impede or prevent monitoring.

### 1.1 GOAL AND OBJECTIVE OF RESEARCH

The goal of this dissertation is to provide improved measures for countering P2P botnets, in particular, reverse engineering, vulnerability analysis, and monitoring based on the challenges that are not sufficiently addressed as of yet, e.g., anti-monitoring mechanisms for *GameOver Zeus* or accurate churn characterization of P2P botnets. Moreover, this dissertation also attempts to "raise the bar" for botnet research by introducing advanced anti-monitoring countermeasures that can be implemented by botmasters in the near future. The main rationale behind proposing such advanced countermeasures is the prediction that botmasters will come up with (most of) these measures sooner or later. The research community that seeks to combat botnets tries to stay a step ahead, working on approaches for overcoming these countermeasures again and again.

## 1.2 CONTRIBUTIONS

The contributions of this dissertation can be categorized based on the corresponding benefactors : 1) defenders and 2) botmasters.

*Defenders*

In the context of defenders, e.g., security researchers or law enforcement agencies, this dissertation provides novel solutions and insights based on real world botnets such as GameOver Zeus, Sality, and ZeroAccess. In particular, this dissertation ...:

1. **...circumvents the neighborlist restriction mechanisms of GameOver Zeus:** The XOR-based neighbor selection mechanism that is implemented by GameOver Zeus attempts to prevent all neighbors of a bot from being completely retrieved by crawlers. In Section 5.1.1, properties of XOR operations are exploited to design and propose an algorithm to manipulate the returned neighbors for a given bot. By strategically spoofing *keys* which are included in the request messages sent to the bots, the proposed mechanism is able to systematically and provably retrieve all neighbors of a particular bot.
2. **...introduces an efficient crawling algorithm:** Existing crawlers have been reported to utilize either Breadth-First Search (BFS) or Depth-First Search (DFS)-based graph traversal strategies in crawling P2P botnets. In contrast, Section 5.1.2 introduces a crawling algorithm that leverages the observation of the existence of a botnet backbone in P2P botnets to crawl more efficiently, i.e., contacting a smaller number of peers to discover as many bots as possible. This proposed algorithm prioritizes popular bots, i.e., bots with high indegree, in its crawling process to discover new bots quickly. As a result, this algorithm is able to crawl fewer bots to discover most of the bots in the botnet overlay.
3. **...presents a more accurate churn model for existing P2P botnets:** Although much work has been done in the field of characterizing churn in regular P2P file sharing networks, very little is available for P2P botnets. Therefore, an accurate churn model is derived in Section 7.1 from empirical results of two real world botnets and they are presented with a much finer measurement granularity than most related work. The results from the two botnets were obtained using a self-developed high-speed crawler that efficiently crawls both botnets. Besides the granular measurements, this work also takes into consideration the noise potentially introduced by unknown third party monitoring activities. Sensor nodes that may be present in the measurements are detected and removed by applying novel sensor node detection mechanisms which are proposed in Section 6.1.

In addition to the above-mentioned contributions, this dissertation also introduces and discusses several strategies to circumvent sensor

detection mechanisms proposed in Section 6.1 from the perspective of a defender. Most of the proposed strategies utilize multiple colluding sensors to deceive the sensor detection mechanisms in believing that the colluding sensors are in fact 'regular' bots.

### *Botmasters*

This dissertation also presents several novel solutions and insights from the perspective of botmasters. In particular, this dissertation . . . :

1. **...presents a lightweight crawler detection mechanism:** By leveraging design constraints in existing P2P botnets, a lightweight crawler detection mechanism is proposed. For instance, assume that all NL request messages (see Section 3.2) in a botnet are always preceded by another type of botnet-specific message. If a crawler deliberately omitted such a message, this behavior can be flagged as an anomaly that may be caused by a crawler. Besides evaluating the feasibility of detecting existing crawlers, this work also characterizes the design of the detected crawlers in existing botnets.
2. **...presents three novel sensor detection mechanism:** To the best of the knowledge of the author, this dissertation is the first to provide a sensor detection mechanism for P2P botnets. Using graph-theoretic metrics, three sensor detection mechanisms are proposed that can be easily adapted to existing botnets. The detection mechanisms attempt to discern sensor nodes from bots based on the observed connectivity among nodes within the botnet overlay, i.e., the neighborhood relationship.

Besides that, this dissertation also proposes several enhancements to the original NL restriction mechanism of GameOver Zeus. The enhancements emphasize on not allowing the NL requesters to have the ability to manipulate the returned entries as it was with the original mechanism of GameOver Zeus.

### 1.3 OUTLINE

The remainder of this thesis is outlined as follows: Chapter 2 introduces the background that is necessary to understand the works presented in this dissertation. Chapter 3 introduces the requirements that need to be fulfilled by any botnet monitoring mechanism. In addition, this chapter also introduces a formal model for P2P botnets, and discusses the state of the art in both botnet monitoring and anti-monitoring mechanisms. Chapter 4 presents the summarized reverse engineering analysis of three major P2P botnets that are focused in this dissertation. This chapter highlights the important details of the botnets which are useful to understand the different contributions within this dissertation.

The major contributions of this dissertation are presented in Chapter 5–7. Chapter 5 presents work on circumventing the NL restriction

mechanism of GameOver Zeus. It also introduces an efficient crawling algorithm that minimizes the number of needed interactions with bots in order to discover the most bots. In addition, a lightweight crawler detection mechanism is also proposed in this chapter to detect crawlers that can be easily implemented even in existing botnets. Chapter 6 introduces three novel sensor detection mechanisms that utilize graph theoretic properties to discern sensors from bots. Besides that, this chapter also provides a discussion on circumventing the proposed detection mechanisms. The contribution in Chapter 7 provides a churn model study for two real world botnets using fine grained measurements that additionally takes into consideration noise of third party monitoring activities. Additionally, representative churn models are also derived and presented for the botnets.

Finally, Chapter 8 presents a summary of this dissertation and details the future work in the field of advanced P2P botnet monitoring.

## BACKGROUND

---

This chapter provides essential background information for this thesis. First, Section 2.1 discusses different botnet architectures. Then, Section 2.2 presents a brief overview of different botnet monitoring mechanisms. Section 2.3 introduces the key metrics which are often used in characterizing churn in P2P networks. Finally, Section 2.4 summarizes this section.

### 2.1 BOTNET ARCHITECTURES

A botnet consists of many thousands of infected machines or bots that are controlled by a botmaster through the usage of a *Command-and-Control Server (C2)*. The C2 is used to disseminate new configurations and updates to the bots and sometimes is also used to upload data from the infected machines, e.g., stolen credit card credentials or passwords. Depending on how bots interact with the C2, a botnet can be classified into the three architectures as described in the following subsections.

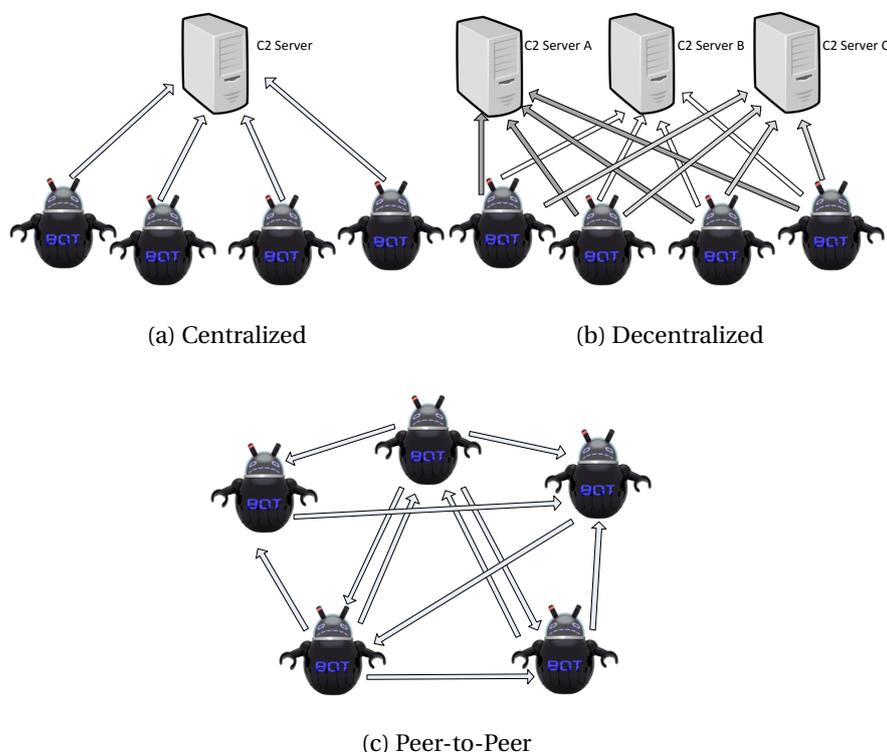


Figure 1: Comparison of botnet architectures.

### 2.1.1 *Centralized Botnets*

In the early days of botnets, the use of centralized C2s was common. The C2s are often deployed either on self-deployed Internet Relay Chat (IRC) servers or hacked webservers. The network topology of such botnets are depicted in Figure 1a. Centralized C2s are easy to deploy due to the simplicity of the network design and have low communication latency. Bots in centralized botnets regularly poll the C2 for newer updates from botmasters and apply them as soon as they are available. However, centralized C2 poses itself as a single point of failure which are frequently targeted by law enforcement agencies in botnet takedowns. The removal of the C2 node or server effectively renders the entire botnet incapable of retrieving newer commands or contacting the botmaster. Moreover, defenders are also able to enumerate all infected machines easily if they get access to the connectivity logs of the C2.

### 2.1.2 *Decentralized Botnets*

The weakness of centralized botnets has motivated the botmasters to design botnets that are not easily taken down. For that, the next generation of botnets were observed to adopt a decentralized architecture where the simplicity and efficiency of a centralized architecture is retained as much as possible but with improved resilience against botnet takedowns by adding redundant C2s as depicted in Figure 1b. Botmasters were observed to experiment with several strategies in the past to implement the redundant C2 feature. Most commonly, the bots have several hard-coded C2 address that are contacted one by one if an existing C2 is not reachable. More advanced strategies have also been seen with the implementation of Domain Generation Algorithm (DGA) or fast-fluxing mechanisms.

A DGA is an algorithm that generates time-sensitive domain names of C2s using a common seed, e.g., date of the day, across the different bots. This way, the botmaster is able to register many domain names in advance that would be contacted by the bots in the future. Even if some of the C2s are taken down, the botmaster is able to reestablish communication with the bots through a new C2 that uses future domain names. However, since the DGA is often hard-coded in the bot's binary, anyone with decent reverse-engineering skills can figure out future domain names that will be used in the bots. Therefore, defenders are able to find out and register these domain names, albeit temporarily, to hijack the botnet from the botmasters [Sg+09].

Another variant of the strategy to introduce redundancies in C2s is implemented using Domain Name System (DNS) fast-fluxing networks. In such networks, the IP addresses of multiple C2s is cycled rapidly via the usage of DNS records for a given domain name(s). In contrast to DGA-based mechanisms, if the botmasters are in control of the fluctuation of the IPs, defenders are not able to predict which IP address will be used to point bots to a C2 in the future. Moreover, an advanced variant of fast-fluxing is called double fast-fluxing: it cycles

a list of name servers which are utilized by the bots to resolve the C2's domain name. The following scenario describes the concept of double fast-fluxing. Whenever a bot needs to resolve a domain name, it sends a request to its name server, e.g., ns.botnet.my. However, each request from the bot may be handled by a different name server that is resolved using the identical name server domain name but different IP addresses. The name servers in turn resolve the requested domain name of the bot into IP addresses using the fast-fluxing concept described earlier. Therefore, such a design introduces an additional layer of fluctuation to increase resiliency, and to prevent successful botnet takedowns.

### 2.1.3 P2P Botnets

Recently, botnets have seen a paradigm shift through the adaptation of a P2P-based architecture which eliminates high vulnerabilities that were present in the centralized and decentralized architectures. Bots in P2P botnets are interconnected via an *overlay* that consists of neighborhood relationships between a bot and a subset of other bots as depicted in Figure 1c. This overlay is maintained using a *Membership Maintenance (MM)* mechanism that is specific to the botnet in scrutiny (cf. Section 4). For instance, like traditional P2P networks, P2P-based botnets also experience *node churn*, i.e., nodes joining and leaving the network at high frequency. To withstand churn, the *MM* mechanism ensures that participating bots remain connected to the botnet overlay by ensuring that unresponsive bots, e.g., offline bots, are removed from the *Neighborlist (NL)* of the bot and replaced by more responsive ones.

The botnet's overlay and MM is also leveraged for the botmaster's update and command dissemination. A botmaster can use any bot within the botnet to *inject* updates which are eventually disseminated to all bots. However, in contrast to the other architectures, P2P botnets experience a delay before all bots are able to successfully receive a new update. Nevertheless, the ability to inject commands from any part of the network is advantageous not only because it allows the botmaster to have an arbitrary number of entry points but it also cloaks the source of the command to prevent any traceback attempts.

In addition, P2P botnets are also more advantageous since they can only be taken down if all bots are disinfected or taken down simultaneously. This is usually done through sinkholing attacks that require a vulnerability within the botnet's design or communication protocols. However, such an attack requires enumeration of all bots in the botnet using some of the monitoring mechanisms presented in the following section.

## 2.2 P2P BOTNET MONITORING

Due to the lack of centralized infrastructures in P2P botnets, e.g., servers, it is difficult to obtain the complete representation of the participat-

ing bots. Hence, the MM mechanisms utilized by P2P botnets are often exploited to monitor botnets by sending request messages yielding results that disclose information about the participating bots.

The most common P2P botnet monitoring mechanisms are the *honeypot*, *crawlers*, and *sensors*. Honeypots are special systems that are purposely deployed to be infected by the botnet's binary so that subsequent communication of the honeypot can be logged and analyzed (cf. Section 3.3.1). Although honeypots are the easiest to be deployed, they can only gather limited information about the bots in a botnet.

In contrast, crawlers leverage the botnet's communication protocol to iteratively request neighbors of bots until all bots have been discovered (cf. Section 3.3.2). However, this monitoring mechanism requires a complete reverse engineering of the botnet's binary to understand and re-implement the botnet's communication protocols to send valid requests to bots. The major drawback of this mechanism is the inability to contact bots behind Internet access points (like routers) that implement the Internet Network Address Translation (NAT) concept [EF94]; such bots represent the majority segment of a botnet's population, i.e., up to 90% [Ros+13].

Sensor nodes are deployed to address the drawbacks of crawlers (cf. Section 3.3.3). They are usually deployed as *routable* nodes using public IP addresses to allow all bots to contact them. Due to the nature of non-routable nodes in P2P networks that remain connected to the botnet overlay through routable nodes, these sensor nodes are eventually contacted by bots behind NAT devices. Therefore, sensor nodes aim to be responsive to all bots contacting them to continue to remain in the NL of bots as a reliable neighbor, i.e., always online and responsive to all request messages. In the process of being responsive to incoming request messages, sensors are able to enumerate both routable and non-routable nodes based on their identities, e.g., IP address and/or botnet specific identifiers. However, a major drawback with sensors is that they are often not able to reconstruct the inter-connectivity between bots due to the inability to actively send requests to bots that are behind NAT-like devices.

However, monitoring of a live botnet is not always possible or straightforward, e.g., in the case of botnets being taken down or unexplained observations that could be caused by the myriad of network configurations and implementations in the Internet. Therefore, botnet research is also conducted using simulators to have a complete control over the investigated scenarios. In order to make the simulations more realistic, a churn model is also applied based on the real world churn measurements as detailed in the following section.

### 2.3 CHARACTERIZING CHURN IN P2P NETWORKS

The robustness and resilience of a P2P network is mainly influenced by the MM mechanism of a network as well as the churn behavior of participating nodes. While the parameters for the MM mechanism can be configured according to the needs of the system, it is not the same with the churn behavior of nodes participating in the network.

Moreover, simulation-based investigations on P2P networks also require accurate churn models to provide a realistic simulation scenario.

As a consequence, quite a number of works have been done in attempting to characterize the churn phenomenon in regular P2P file sharing networks such as *Gnutella* and *Bittorrent* [SR06; Yao+06; SRS05]. Most of these works reported the possibility of using the Weibull probability distribution to represent the observed churn in real world networks. To derive the accurate parameters for the Weibull distribution, several important metrics need to be measured in the P2P network as described in the following.

The metrics to describe churn characteristics of a network can be classified into (1) *group-level*, concerning the collective behavior for all nodes, and in (2) *peer-level*, which requires unique peers to be identifiable through different appearances in the network over time [SR06]. The latter highly depends on the definition of a unique peer, which could be the user, the machine or simply the IP address itself. Especially in the application domain of regular (structured) P2P file sharing networks, identifying peers by their unique identifier is possible, as peers are usually addressed by their identifiers. However, in the domain of (unstructured) P2P botnets, peers are often only addressed by the combination of their IP address and port number. Since IPs can be dynamically allocated using DHCP servers and IPs are shared using devices such as proxies, peer-level characterization based on IP address alone may be distorted as it cannot reliably map IPs to the particular bots. As such, this dissertation leaves out peer-level characterization and puts its focus on the group-level metrics as presented below.

- **Inter-Arrival Time:** The distribution of inter-arrival times captures the arrival pattern amongst peers. For that, the individual arrival of peers needs to be observed and time taken between two consecutive peers joining the network is measured.
- **Session Length:** One of the important property to characterize churn is the session length distribution. For that, the total duration of a peer being available in a given session while participating in the system is measured.
- **Age/Uptime:** Another important metric that needs to be measured is the distribution of the age or uptime of peers. For that, the measurement of the total duration since a peer first joined the system in the current session is required. This metric is useful in addition to the distribution of session length to characterize the possible combination of different session lengths that could coexist at the same time.
- **Remaining Uptime:** The final important metric for churn characterization is the distribution of the remaining time of an online peer to predict when a particular peer would go offline. For that, the remaining time of an online peer before it leaves the system in the current session is measured and correlated with the current uptime of the peer.

## 2.4 CHAPTER SUMMARY

This chapter presented some background that is necessary for the readers to understand and follow the remaining parts of this dissertation. Three subdivisions have been made, first presenting the three architectures often adopted by botnets, second going into monitoring of P2P-based botnets, before closing with a short description on characterizing churn in P2P botnets.

1. *Botnet Architectures*: By comparing the features offered by the common botnet architectures, i.e., centralized, decentralized, and P2P, the P2P architecture is motivated as an architecture that allows great flexibility to a botmaster to manage a botnet. The self-organizing and self-healing properties of the P2P botnet overlay allow the bots to remain connected among each other with minimal involvement from the botmaster. Most importantly, this architecture has an increased robustness and resiliency against botnet takedowns and random node failures compared to other architectures. The properties of this architecture seems to be preferred by botmasters; as seen from the increasing number of P2P botnets emerging in the wild.

2. *P2P Botnet Monitoring*: P2P botnets bring new set of challenges in monitoring bots participating in the botnet. Since each bot in a P2P botnet only maintain a localized view of the botnet overlay, a complete view of the overlay or the participating bots is not easily obtained. To address these issues, the MM protocols of botnets are leveraged to perform botnet monitoring using the mechanisms presented in Section 2.2: honeypots, crawlers, sensor nodes.

Out of the three monitoring mechanisms, crawlers and sensor nodes offer greater flexibility to perform botnet monitoring compared to honeypots. The background information presented on the monitoring mechanisms are useful to understand the works presented in Chapter 5 and 6.

3. *Characterizing Churn*: Botnet monitoring data provides valuable information to describe a botnet under scrutiny. Specifically, churn characterization of a botnet is useful to understand the dynamics that occur within the botnet. The characterization are often done based on churn metrics as introduced in Section 2.3.

These introduced metrics are later used in Chapter 7 to characterize churn in two botnets: Sality and ZeroAccess. Using these metrics, the dynamic nature of the botnets are studied and representative churn models are derived.

In the next chapter, the requirements and related work in the area of botnet monitoring is presented and thoroughly analyzed.

Chapter 2 provided the necessary background on P2P botnets and the different mechanisms that are often utilized to monitor them. This present chapter provides a detailed discussion on botnet monitoring mechanisms as well as the common challenges often faced in monitoring. In more detail, Section 3.1 presents the requirements of a botnet monitoring mechanism with emphasis on P2P botnets that aims at obtaining high-quality monitoring data. Then, Section 3.2 presents a formal model on P2P botnets that is used throughout this thesis.

In Section 3.3, the related work in botnet monitoring is discussed. After that, Section 3.4 discusses the challenges commonly faced in botnet monitoring. In particular, Section 3.4.1 introduces issues stemming from the dynamic nature of P2P networks. Section 3.4.2 elaborates on the pollution of monitoring results due to monitoring activities of unknown third parties. In addition, Section 3.4.3 discusses the various anti-monitoring mechanisms implemented in botnets and those proposed by related work. Finally, a thorough discussion sums up the chapter in Section 3.5.

### 3.1 REQUIREMENTS OF A BOTNET MONITORING MECHANISM

In the following, the functional and non-functional requirements of a botnet monitoring mechanism are presented.

#### 3.1.1 *Functional Requirements*

A botnet monitoring mechanism has to conform to the following functional requirements:

1. **Genericity:** In order for a monitoring mechanism to be easily adaptable to different botnets, the mechanism should be designed and developed in a generic manner. Although most of the recent P2P botnets utilize custom protocols to communicate among the bots or the botmaster, the steps to monitor them remain fairly similar. Therefore, a monitoring mechanism should be able to be easily adapted to any P2P botnets. Whenever a new botnet is discovered; only the communication protocol needs to be implemented and integrated to the monitoring mechanism. Such a design significantly reduces the required effort and time to monitor a new botnet.
2. **Protocol Compliance:** Monitoring mechanisms should comply with the protocols utilized by the botnet under scrutiny. The communication protocols used by the bots need to be understood and used when communicating with the bots. The pro-

tol compliance is important as most botnets only respond to requests or messages that adhere to their botnet protocols, e.g., encryption and decryption routines. In most cases, reverse engineering of the botnet's malware binary is required for obtaining a complete understanding of any botnet's inner workings.

3. **Enumeration of Bots:** The ability to enumerate infected machines within a particular botnet during monitoring is an important aspect of a monitoring mechanism. Based on the enumeration effort, it is possible to estimate the population size of the botnet. Furthermore, this information is also useful to measure the success of cleanup activities. Bot enumeration is usually done by leveraging on the botnet-specific request and reply messages. Every valid response for a sent request message indicates the presence of a bot that is identifiable by either a particular IP address and port number combination or a botnet specific identifier at a given point in time.
4. **Neutrality:** A monitoring mechanism should stay neutral during its monitoring activities in order to prevent introducing artificial noise that might influence the observed behavior of a botnet. Specifically, a mechanism should avoid executing command from the botmaster or disseminating them further to legitimate bots. This ensures that the monitoring mechanism does not aid the botnet in its malicious activities, e.g., binary updates or attacks.

Besides that, the mechanism should also avoid to intentionally introduce noise that may disrupt or hamper the normal activities of the bots. Disrupting the activity of the bots may introduce bias in the monitoring data of other researchers and may lead to inaccurate conclusion of the real nature of the botnet. Therefore, whenever in doubt, the best course of action is not to respond to any request that is not crucial for the monitoring activity itself.

5. **Logging:** All information gathered from a monitoring mechanism should be logged with the associated timestamps for further analysis. The logged information should include additional botnet-specific metadata such as the details of the latest command from the botmaster or unique identifiers of the bots (if available).

The logged information is particularly useful to informing the relevant stakeholders, i.e., Internet Service Providers (ISPs) and network administrators, about the infections as well as understanding the botnet itself. For instance, the logged information of a crawler can be used to reconstruct the botnet's network topology from the crawler's point of view. This information can then be further analyzed using graph analysis techniques to identify most influential bots in the botnet overlay. Moreover, in some cases, historical monitoring data may also be useful to ensure

a higher success rate in a botnet takedown attempt, i.e., identifying stable bots that are reliable in terms of their uptime or presence in the botnet overlay.

### 3.1.2 *Non-Functional Requirements*

The following non-functional requirements are directly related to the quality of a botnet monitoring mechanism as well as its collected data.

1. **Scalability:** A monitoring mechanism may be defined as scalable when its performance does not deteriorate with an increased number of bots in a botnet. In existing and earlier botnets, the total population ranged anywhere between several thousand to a few million bots. Scalability of a monitoring mechanism is not only about being able to handle the high volume of request and reply messages of all bots in a botnet, but also system resources regarding memory, bandwidth, computational resources and storage space.
2. **Stealthiness:** It is important to ensure that a monitoring mechanism is not identified by the botmasters or the bots in the process of monitoring the botnet. Since monitoring activities threaten the economy and the existence of the botnets themselves, botmasters may retaliate against such monitoring mechanisms. For instance, botmasters may launch DDoS attacks or blacklisting of IP addresses used for monitoring. The retaliation attacks may cause disruption to an ongoing monitoring activity or renders the IP address completely unusable for further monitoring. Hence, a monitoring mechanism should circumvent any mechanisms that may indicate an ongoing monitoring activity to the bots or the botmaster.
3. **Efficiency:** Efficiency is two-fold and can be divided in *probing* and *resource efficiency*. For probing activities, efficiency manifests on the ability of minimizing noise introduced in the resulting data in order to obtain high quality monitoring data. Since a botnet overlay also experiences *churn* and *diurnal* effects similar to traditional P2P networks(cf. Section 3.4.1), any delay in probing bots may lead to bias in the resulting data [SRS05]. Therefore, a monitoring mechanism should be designed to probe and respond to bots as quickly as possible.

The resource efficiency of a monitoring mechanism focuses on the ability to obtain high quality monitoring results but only with minimal resources, e.g., minimum number of request messages. For instance, botnets could be designed to send a probe message to verify the responsiveness of a bot before sending a different message for requesting neighborlists. A crawler could omit sending the probe messages and utilize the neighborlist request/reply message instead to assert the responsiveness of a bot as well as to obtain the neighbors of the bot.

4. **Accuracy:** Accuracy is two-fold and can be divided into *enumeration* and *connectivity accuracy*. For enumeration of bots, accuracy manifests in the ability to discover all bots that are online at a given point in time. This is important because the overall population of the botnet can be inferred from the number of online bots. In addition, information of which bots are offline is also important to assist any takedown attempts, i.e., ensuring no bots have been missed.

Besides bot enumeration, the (inter-)connectivity, i.e., overlay connectivity, among the bots is also important in botnet monitoring. This information is particularly very useful for botnet takedown operations that involve strategically invalidating the inter-connectivity among bots. For that, the connectivity accuracy manifests as the ability to capture the overlay topology as known to the bots participating in the botnet at a given point in time. Therefore, high accuracy of a monitoring mechanism in capturing the exact state of a botnet provides high-quality monitoring data.

The monitoring activities of unknown third parties may also introduce noise in the resulting monitoring data. For instance, a sensor deployed by a researcher may yield very high *uptime* compared to regular bots and consequently affect any measurements that are relying upon the *uptime* of bots. As such, a monitoring mechanism should also identify and remove such noise from the monitoring data to increase the accuracy in capturing and characterizing only the bots within the botnet.

5. **Minimal Overhead/Noise:** A monitoring mechanism ensures that its activities or footprints do not pollute the botnet with artificial data that could significantly alter the nature or behavior for both the botnet itself and other monitoring parties. Although it is evident that existing monitoring activities will introduce noise, it is crucial to ensure that necessary steps to reduce the noise are taken into consideration. Moreover, minimizing the generated overhead and noise also increases the stealthiness of a monitoring activity from being detected by the bots or the botmasters.

### 3.2 FORMAL MODEL FOR P2P BOTNETS

This section introduces a formal model for P2P botnets that will be very useful to understand the various work presented in Chapter 4–7.

A P2P botnet can be modeled by a directed graph  $G = (V, E)$ , which is a common practice [Dag+07; Dav+08; Ros+13], where  $V$  is the set of *bots* in the botnet and  $E$  is the set of edges or inter-connectivity between the bots, i.e., the neighborhood relationship. Bots  $V$  in a botnet can be further classified into two different categories of bots, i.e.,  $V = V_s \cup V_n$ , where *superpeers*  $V_s$  are bots that are directly routable and *non-superpeers*  $V_n$  for those that are not directly routable, e.g., behind stateful firewalls, proxies, and network devices that use NAT. Please

note that in the remainder of this thesis, the terms *bot*, *peer*, and *node* are used synonymously.

All bots use a MM mechanism (cf. Chapter 2.1.3) that establishes and maintains neighborhood relationships, i.e., a neighborlist, to ensure a connected botnet overlay. Hence, bots have connections to a subset of other bots, i.e., neighbors, 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 neighborlist NL of a bot  $v \in V$  is defined as  $NL_v = \{u \in V \mid \forall u \in V : (v, u) \in E\}$ . Hence, the *outdegree* of a bot  $v$  can be defined as the number of outgoing edges or neighbors maintained by the bot as:  $\text{deg}^+(v) = |NL_v|$ . The maximum outdegree is governed by the global botnet-specific value of maximum entries  $NL^{\text{MAX}}$  that can be stored at any given point in time, i.e.,  $|NL_v| \leq NL^{\text{MAX}} \leq |V|$ . Moreover, the *popularity* or *indegree* of a bot  $v$  in the botnet can be measured based on the number of bots that have  $v$  as an entry in their NLS:  $\text{deg}^-(v) = |\{(u, v) \in E\}|$

Bots in a botnet use a MM mechanism to maintain their NLS regularly following a botnet-specific interval that is often referred to as MM-cycle. In each cycle, a bot probes for the *responsiveness* of all of its neighbors by sending a *probe message* to each of them. The responsiveness of all neighbors can be verified based on a valid response to the sent messages. In this proposed model, this probe message is referred to as the *probeMsg*.

In addition, if the bot has low neighbors or many of the existing neighbors are not responsive, additional neighbors can be requested to fill up the NL. For this purpose, a peer  $v$  can request its neighbor  $u$  to select a subset of  $u$ 's neighbors  $L \subseteq NL_u$  and share them with  $v$  via a neighborlist request method that is referred to as the *requestL* in this proposed model. The decision on which exact entries are picked in the returned response message  $L$  depends on the *neighbor selection criteria* employed by the botnet.

Bots (re)joining the overlay often *announce* their existence to a subset of existing superpeers using a message that is referred to as *announceMsg* in this proposed formal model. The information of these superpeers can originate from a hard-coded list within the malware binary for new bots or from previous bot communications for bots rejoining the overlay. Superpeers receiving such a message will check if the new bot is a potential superpeer by sending a message to the port that is used by the sender for receiving incoming requests. This information about which port is often transmitted along with the initial message to the superpeer. A valid response to the message indicates the direct reachability of the new bot. Hence, it is a potential superpeer candidate as well. Therefore, the information of the new superpeer can be stored within the existing superpeer's NL and further propagated when additional neighbors are requested by other bots in need of new neighbors.

Non-superpeers would usually fail to receive the probe messages sent by the superpeers due to the presence of NAT-like devices that drop unsolicited messages, i.e., messages initiated remotely. Such bots are often not included in the neighborlists of the superpeers. These

bots rely upon existing superpeers to relay any update to or from the botmaster.

### 3.3 RELATED WORK ON BOTNET MONITORING

The open nature of P2P botnets allows anyone with knowledge of the botnet communication protocols to participate within the network and communicate with any bots as explained in Section 2.2. This openness is often leveraged to conduct monitoring on the P2P botnets by disguising as yet another bot. Monitoring in P2P botnets is often done with the aim of identifying and enumerating all infected machines. Besides using the monitoring data to perform cleanup activities, the data can also provide valuable information in understanding and tackling P2P botnets.

In addition, monitoring also enables the possibility of understanding the malicious operations of the botnets themselves. For instance, by analyzing the commands that are regularly injected in the botnet, it is also possible to identify the operators or botmasters. All of this valuable information is also particularly useful and important in the event of a botnet takedown attempt, to ensure a higher success rate [Ros+13].

In the following, the state of the art of three commonly used botnet monitoring mechanisms: *honeypots*, *crawlers*, and *sensor nodes*, are presented with respect to the requirements presented in Section 3.1.

#### 3.3.1 Honeypots

In the early days of botnet monitoring, researchers used *honeypots* to monitor centralized botnets. Honeypots or *honeynets* are machines or a network of machines designed to appear as lucrative targets in the eyes of malware and attackers. Such machines or networks aim at being infected to monitor any subsequent malicious activities [Spi03]. Using honeypots is easy and straightforward as no prior knowledge of the malware or its communication protocol is needed to conduct monitoring.

Malware that infected a honeypot would contact its C2, e.g., IRC, to report back to the botmaster. By monitoring the network traffic generated by the honeypot, it is possible to discover the details of the C2, e.g., IRC server, that is being used. Moreover, the information of other infected machines that are reporting back to the C2 can also be obtained by inspecting the C2 communication logs. For instance, McCarty presented results of their deployed honeypot being recruited by a botnet that uses an IRC server to control its bots [McC03].

The main drawback of this approach is the fact that the user has minimal control over the actions that are taken by the bots within the honeypot environment, e.g., participating in an ongoing attack. In [McC03], McCarty reported his efforts to rate-limit the generated network traffic and manual blocking of certain ports to minimize the damage that may be done by the malware. These efforts are taken

mainly because it may be illegal if a user knowingly *volunteers* to be part of the botnet activities or participate in an ongoing attack. Moreover, if the traffic generated by the infected machines is encrypted, users can only obtain communication metadata.

The limitations associated with using honeypots for botnet monitoring have led to the development of more advanced monitoring mechanisms like crawlers and sensor nodes. These advanced monitoring mechanisms allow the user to have a full control over all monitoring activities. For instance, a monitoring mechanism can selectively refuse to respond or forward certain messages like new command dissemination to other bots. Furthermore, these mechanisms can communicate with any bots and increase their coverage according to the purpose of monitoring. Therefore, in the remainder part of this thesis, the focus is only on the usage of these two advanced mechanisms in the context of monitoring P2P botnets.

### 3.3.2 Crawlers

Due to the self-organizing nature of P2P botnets, bots can actively request additional neighbors when the number of responsive neighbors in their NL is low. This observation is leveraged by the *crawler* which is a computer program that is implemented to mimic the behavior of a bot that is low on neighbors and to request neighbors. However, since bots in a botnet only respond to communication that conforms to their protocol, a crawler needs to implement parts of the botnet protocol for sending *requestL* request messages and parsing the replies accordingly (Functional Requirement 2 and 4).

Starting with a list of *seed* nodes, i.e., superpeers, that is retrieved by reverse engineering the malware binary, a crawler requests the neighbors of this node and iteratively sends *requestL* to all newly discovered bots to obtain entries from their NL. The goal of crawling is to obtain an accurate *snapshot* of the botnet by obtaining information of all infected bots as well as the inter-connectivity among them. Each snapshot consists of a directed graph that represents each discovered bot along with its neighborhood relationship as described in the formal model presented in Section 3.2 (cf. Functional Requirement 3 and Non-Functional Requirement 4). A visual representation of such a snapshot is presented in Figure 2.

The steps of requesting the NL of all bots can be implemented using graph traversal techniques such as DFS or BFS. These techniques can be easily implemented within the crawlers by using either a stack or a queue-based implementation as the node selection/crawling strategy respectively. Finally, the information of *who knows whom* can be stored for further analysis (Functional Requirement 5). Take note that it is important for a crawler to request the NL of bots in quick successions to reduce the network bias, e.g., address aliasing, churn effects introduced in the resulting botnet snapshot [Kar+14; Wan+09] (cf. Section 3.4.1). Accurate snapshots are particularly important to conduct an effective botnet take-down attempt or to analyze the re-

silience of the botnet against potential attacks (Non-Functional Requirement 3 and 4).

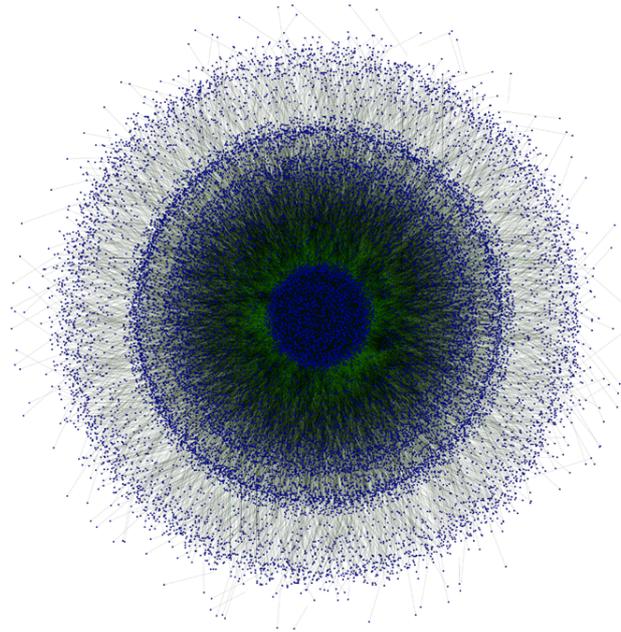


Figure 2: GameOver Zeus network connectivity graph between 23,196 nodes reconstructed via crawling. The blue dots indicate the nodes (systems infected with GameOver ZeuS) and the green lines indicate the edges between nodes. (Source: Dell SecureWorks)

However, crawling fails to enumerate *all* bots in the botnet. Depending on the MM mechanism of the respective botnets, crawlers are only able to contact the *superpeers*, i.e., bots that are directly reachable. Bots that are behind network and security devices such as NAT, proxies, and firewalls, i.e., non-superpeers, are not directly reachable by crawlers, but make up the majority of the entire botnet population (according to [Ros+13] 60 – 90%). These bots often rely on the superpeers to remain connected to the botnet overlay and to receive new updates from the botmasters.

One of the first P2P botnets that have caught the attention of the public and media was *Storm* [Hol+08]. This botnet initially coexisted together with the OVERNET P2P file sharing network, and eventually moved on to a bot-only network which is referred to as *Stormnet*. Following the discovery of the botnet, many researchers attempted to monitor Storm and presented their analysis [Hol+08; KLE08; Enr+08; Wan+09]. The results presented by the researchers were interesting due to discrepancies stemming from differing approaches and assumptions in monitoring the botnet [Yan+14a; Ros+13].

Since Storm initially coexisted with OVERNET, one of the main challenges was to distinguish bots from benign clients of OVERNET. Wang et al. presented a methodology to identify bots based on the observation that the DHT IDs of the Storm bots are not persistent over reboots and, therefore, change frequently compared to benign users [Wan+09].

One of the earliest work on monitoring Storm was performed by Holz et al. using a BFS-based crawler called *StormCrawler* [Hol+08]. This crawler iteratively queries each bot starting from a *seedlist* and sends 16 route request messages consisting of carefully selected DHT IDs, i.e., evenly spaced around the DHT space, to increase the chances of retrieving undiscovered peers. Mainly contributed by the open nature of P2P botnets, many researchers started actively monitoring Storm and experimented with it. Kanich et al. reported the presence of many unknown third parties that monitor the botnet in parallel by using a built-in heuristic on their *Stormdrain* crawler to identify non-bots. This heuristic is based on a design flaw of the OVERNET ID generator within the binary that is capable of only generating a small range of IDs [KLE08]. However, the authors admitted that they were not able to identify researchers that could have chosen IDs that fall within the range that is used by bots in Storm.

Most crawlers on Storm have been reported to conduct crawling by randomly *searching* for IDs around the DHT space in the hope of eventually discovering all participating bots. However, as reported by Salah and Strufe [SS13], a more accurate snapshot can be obtained using their *KAD Crawler* that crawls the *entire* KAD network in a distributed manner by leveraging the design of KAD itself, which is also the design adopted by OVERNET. Their distributed crawling approach is not dependent on *online* nodes, unlike existing Storm crawlers, instead splits the KAD ID space into multiple zones and assigning crawlers to dedicated zone to retrieve the routing tables of all nodes within each zone. The results from the different zones can then be aggregated as the snapshot of the botnet.

Other P2P botnets have also attracted the attention of researchers in monitoring them. For instance, Dittrich and Dietrich deployed a DFS-based crawler to crawl *Nugache* [DD08]. Their crawler conducts *pre-crawls* and utilizes that information as an input for their priority-queue based implementation that prioritizes nodes which have been observed more often available and responsive in the pre-crawls. More recently, Rossow et al. presented their analysis on the resiliency of P2P botnets, namely *GameOver Zeus*, *ZeroAccess*, and *Salinity*, using their BFS-based crawler that starts crawling from a *seednode* and appends newly discovered nodes from previously crawled bots at the end of a queue. In 2014, Yan et al. introduced *SPTracker* to crawl the three botnets mentioned above [Yan+14b; Yan+14a]. Instead of using only conventional crawling, *SPTracker* includes *node injection* (cf. Section 3.3.3) as a complementary mechanism to obtain better crawl results.

From the domain of unstructured P2P file sharing network, Stutzbach et al. presented *Cruiser* to crawl *Gnutella* [SRS05]. This crawler prioritizes *ultrapeers* from the two-tier design of the *Gnutella* network and can capture an accurate representation of the P2P network through quick crawling. The authors also report of an observed connectivity bias among peers which are most likely connected to peers with higher uptime. Similar observations in unstructured P2P botnets have also been reported by other researchers [Yan+14b; Ros+13].

Although most of the proposed botnet crawlers meet all functional requirements presented in Section 3.1.1, not much focus have been given on the non-functional requirements such as stealthiness, accuracy and the amount of noise being introduced within the footprint of the botnet (cf. Table 2). These unmet requirements will become more apparent upon discussing the challenges often encountered in monitoring P2P botnets in Section 3.4 where the challenges affect the accuracy of the crawl data (cf. Non-Functional Requirement 4).

### 3.3.3 *Sensor Nodes*

Due to the presence of network devices that allow sharing of IP addresses across many machines, e.g., NAT, botmasters, too, experience the same problem of regular P2P networks which often have two distinct classes of devices: superpeers and non-superpeers. As peers behind NAT are not directly reachable, botnets follow a two-tier network structure to enable all bots participating in the overlay management remain connected among themselves. The non-superpeers rely upon the superpeers to remain connected to the botnet and to receive new updates or commands from the botmaster. Commands from the botmaster is retrieved by polling the superpeers for newer updates. The superpeers can relay any information from the botmasters to requesting bots, and therefore circumventing the NAT traversal issues.

Fortunately, this two-tiered network design can also be exploited to monitor the botnets. Kang et al. were the first to propose a mechanism called *sensors* to enumerate *structured* P2P botnets [KCTL09], e.g., Storm. The sensors are directly routable and are deployed using strategic DHT IDs intended to intercept route requests of other bots. Since the requests were initiated by the bots themselves, the sensors can identify the non-superpeers based on the intercepted request messages. In contrast to crawling, sensor nodes are able to enumerate both superpeers and non-superpeers (cf. Section 3.3.2). This idea has also been extended and applied for monitoring other existing unstructured P2P botnets [Ros+13].

By exploiting the node announcement mechanism that is required in each P2P botnet, a sensor node can be *announced* to existing superpeer bots (cf. Section 3.2) using the *announceMsg* method. When a non-superpeer requests additional neighbors from a superpeer, information about the sensor node may also be returned. Eventually, non-superpeers will insert the sensor into their NL and from there on will regularly probe the sensor for its responsiveness. Therefore, sensors can identify and enumerate bots that are not directly routable and thus cannot be discovered by crawling.

As explained in Section 2.1.3, entries within an NL are only removed or replaced if the associated bot has (consistently) remained unresponsive when being probed. To avoid being removed from the NL of the bots, sensors must always be responsive when being probed by other bots. Moreover, the high availability of a sensor also directly influences its popularity [Kar+14], i.e., the number of bots that have the sensor in their NL. The more superpeers can verify the responsive-

ness of a sensor at all time, the more bots will propagate the sensor's information to other bots. Hence, this improves the coverage of the sensor.

A variation of a sensor node is also often used in *sinkholing* attacks on P2P botnets. Such an attack requires a vulnerability within the botnet protocol that is exploitable to overwrite information in a bot's NL. By invalidating all entries of a bot, except those of the sensor(s), bots will only be able to communicate to and through the sensor(s). As such, all requests for newer commands to or from the botmaster can be dropped and thus the communication between the botmaster and its bots is disrupted.

However, although sensors can enumerate bots that are not discovered by crawlers, they are not able to retrieve the connectivity information of the bots. It is usually not possible to actively request the NL of the non-superpeers, except in rare cases in which a botnet design allows UDP hole punching techniques [Ros+13].

Although both, crawlers and sensors, have their set of advantages and disadvantages, in most cases they complement each other. A combination of both mechanisms provides a better monitoring results. For instance, some researchers have augmented their sensors with crawling capabilities in their monitoring activities [Yan+14a; Yan+14b].

Now that the state of the art in botnet monitoring is discussed, the following section looks at the challenges associated with monitoring activities.

### 3.4 CHALLENGES IN BOTNET MONITORING

Although botnet monitoring can be easily conducted by anyone with sufficient knowledge of the respective botnet protocol, some challenges need to be taken into consideration while monitoring to ensure the reliability and the quality of the collected data. These challenges, if not addressed carefully, can result in distorted or impartial monitoring results that may consequently lead to wrong interpretations of the data in comparison to the real botnet (Non-Functional Requirement 3–5).

These challenges are described in the following section and are mainly due to the dynamic nature of P2P (bot) networks itself (Section 3.4.1), noise resulting from monitoring activities by unknown third parties (Section 3.4.2), and anti-monitoring countermeasures that are deployed by the botnets (Section 3.4.3).

#### 3.4.1 *The Dynamic Nature of P2P Botnets*

The dynamic nature of the overlay of a P2P botnet which is similar to a regular P2P file-sharing networks, poses several challenges to botnet monitoring mechanisms. These challenges are discussed in detail in the following:

1. **Churn and Diurnal Effects** : A botnet overlay experiences high *churn* rate of nodes joining and leaving the network at high fre-

quency [SRS05]. Therefore, crawlers that crawl bots with either a low frequency or taking a longer period to complete a full crawl may introduce a significant network *bias*, i.e., in considering bots to be online that have already went offline, within the produced snapshot. In addition, newly arrived peers might also be missed from being captured by the crawler [SRS05].

Moreover, bots within the overlay also experience *diurnal effects* where significant portions of bots go offline and come online based on geographical timezones [SRS05], e.g., computers that are turned on/off during or after working hours. This observation suggests that any short-term measurement of a botnet, i.e., less than a week, would be heavily influenced by such diurnal effects.

2. **IP Address Aliasing :** In addition to the dynamic nature of the overlay, IP address *aliasing* occurs in P2P botnets. Primarily contributed due to the shortage of IPv4 addresses and security concerns, ISPs and organizations use various network devices such as NAT and proxies to share the limited number of IP addresses that are available for their network(s). As such, from the Internet viewpoint, any network traffic generated by machines behind such devices seems to be originated from only a single IP address. Thus, measurements that rely upon IP addresses alone may underestimate the total number of infected machines.

In contrast, ISPs or organizations that run a DHCP service for a dynamic allocation of IP addresses to their users may also influence ongoing measurements. For instance, traffic originating from a single infected machine can be observed from several IP addresses due to different addresses (re)allocated to existing bots by the DHCP servers, e.g., after reboots or the expiry of a lease period. This address aliasing issue, may lead to overestimation of the number of infected machines.

The issue of IP address aliasing can also occur due to the presence of load-balancing infrastructures that may be used by local network administrators or ISPs. In such cases, network traffic from an infected machines may seem to originate from different IP addresses (in the viewpoint of a sensor node).

One way to overcome the IP address aliasing issue is to use persistent and unique botnet-specific identifiers (UID), if applicable, to enumerate and associate the infections accurately [Ros+13]. As depicted in Table 1, although many botnets use some kind of UIDs, not all of them can be used to uniquely distinguish the bots. For instance, the UID of Sality is not reboot-persistent and as such unreliable to be used to distinguish bots uniquely. Moreover in some extreme cases like the Miner botnet, there is no UID at all, hence rendering a more accurate estimation of the botnet population difficult or even impossible. Therefore, bots should be at least distinguished by using the combination of the

IP address and port number used by them in the absence of a reliable UID.

Botnet	UID
Kelihos V1	16 bytes
Kelihos V2	16 bytes
Kelihos V3	16 bytes
Miner	None
Nugache	(not shared)
Sality V3	4 bytes (non-persistent)
Sality V4	4 bytes (non-persistent)
Storm	16 bytes
Waledac	20 bytes
ZeroAccess V1	(not shared)
ZeroAccess V2	4 bytes (not shared)
GameOver Zeus	20 bytes

Table 1: UIDs of existing P2P botnets that are retrievable from crawling [Ros+13].

### 3.4.2 Noise from Unknown Third Party Monitoring Activities

To address the dynamic nature of P2P networks, mediation steps or new measurement techniques can be designed that take them into account and to obtain a more realistic and accurate measurements. However, the second challenging aspect of botnet monitoring is the presence of unknown third party monitoring activities. As explained in Section 3.3, botnet monitoring activities generate a considerable amount of noise that is inserted into the botnet’s footprint. However, when the third party is unknown, footprints of their monitoring activities will, unfortunately, be attributed as those originating from the bots [KLE08]. For instance, consider the scenario of a third party interested in conducting a churn measurement in a particular botnet. As discussed in Section 3.3.3, sensors deployed aim to be highly responsive for a prolonged period to ensure high popularity. However, the presence of sensor nodes with longer session lengths may skew the churn measurements as most benign bots have significantly shorter session lengths. Hence, any derived churn model may not be representative of the real churn in the botnet.

Kanich et al. monitored the Storm botnet and took the active pollution into account by leveraging upon a flaw within the botnet’s ID generator [KLE08] (cf. Section 3.3.2). In addition, the authors classified the different type of sensors that were deployed based on their responses to sent request messages [Enr+08]. Besides that, they also described how they had to stop relying on the nature of the botnet’s participants and carefully handle all received request and response

messages. Their crawler often crashed within the first few minutes of crawling the network due to the presence of many malformed packets and the ongoing pollution attack within the botnet which uses bogus source IP addresses. Therefore, they had to put in much engineering effort, to enable their crawler to be fault-tolerant and crawl successfully. However, they also argue future monitoring mechanisms may become more stealthy and indistinguishable from benign bots. As a consequence, stealthy monitoring mechanisms will influence the botnet's footprint and skew the monitoring results one way or another.

### 3.4.3 *Anti-Monitoring Mechanisms*

The third challenge in monitoring P2P botnets is the most interesting aspect of them all: anti-monitoring mechanisms deployed by botmasters. Botnets are an important asset to their botmasters due to the high profit that can be generated by them. Consequently, their malicious activities also attract the attention of many researchers and law enforcement agencies.

In the past, botnet monitoring activities have resulted in several botnets to be successfully taken down [Ros+13; Naz07]. For these reasons, botmasters are aware of the monitoring activities and have equipped recent botnets with anti-monitoring mechanisms. At the same time, further research contributed additional countermeasures to impede crawling and deployment of sensor nodes. However, to the best of knowledge, none of these proposals have been seen to be adopted by existing botnets (as at time of writing).

Anti-monitoring mechanisms from the perspective of a botmaster can be classified as follows: 1) *Prevention*, 2) *Detection* and 3) *Response*. The first category of anti-monitoring mechanisms can be deployed in advance to impede or prevent monitoring activities. The second category focuses on detecting ongoing monitoring activities and the last category addresses response actions to detected monitoring activities.

#### 3.4.3.1 *Prevention*

Anti-monitoring mechanisms within this category aim to impede monitoring activities by design of the botnets themselves. Mechanisms of this category are commonly implemented in many of recent botnets. However, a majority of them are focused on impeding performance of botnet crawlers. In the following, mechanisms targeting crawlers will be first discussed, and followed by those targeting the sensors.

A majority of anti-monitoring mechanisms against crawlers are primarily focused on the neighborlist return mechanism of the botnets (cf. Section 2.1.3) as detailed in the following:

1. **Restricted neighborlist replies** : Many P2P botnets restrict the size of the returned *NL* when being requested, by handing out only a small fraction of their overall neighbors. In addition, botnets also have a neighbor selection strategy to decide which neigh-

bors are to be picked and returned upon request. For instance, bots in Sality returns only one random (but active) neighbor [Fal11] and ZeroAccess [Wyk12] return the 16 most-recently probed neighbors when requested. Meanwhile, botnets such as the GameOver Zeus [CER13; And+13] come with a mechanism that returns only ten neighbors that are "close" to a botnet-specific ID specified within the *requestL* message and the ID of the neighbors. This restriction mechanism utilizes the *Kademlia*-like XOR-distance metric [MM02] to calculate the notion of closeness between two bots.

However, despite the presence of such restriction mechanisms, crawling is still possible. In Sality and ZeroAccess, a crawler needs to send *requestL* messages continuously to all discovered bots until the results converge, i.e., no newly discovered bots [Yan+14a; Yan+14b; Kle15]. Similarly for the GameOver Zeus, a crawler needs to repeatedly query bots in GameOver Zeus for their neighbor lists by spoofing different IDs chosen randomly as reported by Rossow et al. [Ros+13]. Nevertheless, the restriction mechanisms of all these botnets implies that crawlers can only achieve a limited accuracy and are not able to provably retrieve or discover the *complete* neighborlist of a bot. As a consequence, the accuracy of the obtained monitoring data may be low or poor.

2. **Ratbot** : A theoretical and DHT-based structured P2P botnet called *RatBot* was proposed by Yan et al. that returns spoofed non-existing IP addresses when requested to hinder attempts to enumerate the botnet [YCE11]. This mechanism makes the crawling process difficult and inefficient due to additional nodes that do not respond. This idea of Ratbot can also lead to an over-estimation of the botnet size, which may be a preferred feature for botmasters, e.g., publicity among potential clients. Although crawlers may still work in RatBot, the introduction of excessive noise to the monitoring data may adversely affect botnet take-down attempts.
3. **Overbot** : Starnberger et al. [SKK08] proposed a botnet called *Overbot*, which does not disclose the information of other bots or the botmaster if compromised by security researchers. The idea of Overbot is to let the infected machines to communicate to the botmaster using DHT keys that are generated by encrypting a *sequence number* with the public key of the botmaster. For this, bots in Overbot utilize the DHT space within an existing P2P file sharing network like Overnet to publish *intentions* to communicate with the botmaster.

By deploying several *sensor* nodes that listen for and decrypt search requests or intentions, a botmaster can identify bot-originated requests and communicate to the infected machines individually. However, since the sensors are assumed to have a copy of the botmasters private key, they pose themselves as a single point of failure if the sensors are compromised. Moreover, since

bots continuously search for keys that could be found by the botmaster, such a pattern and noise or overhead can easily raise suspicions.

#### 4. **Rambot :**

Focusing on unstructured P2P botnets, Hund et al. proposed *Rambot* which uses a credit-point system to build bilateral trust amongst bots and to use it as a *proof-of-work* scheme to protect against exploitation of its neighborlist exchange mechanism [HHH08]. For that, they request all nodes including crawlers to complete some computationally intensive tasks like crypto-puzzles before returning their neighborlists. However, with the advancement of computing resources that are available today, this proof-of-work mechanism can be easily circumvented.

5. **Manual neighborlist update:** Another interesting approach was proposed by Wang et al. in [WSZ10] to allow botmasters to manually update the neighborlist of *all* bots from time to time. However, the design of the botnet requires frequent interactions from botmasters to instruct bots to *report* to a specific node with the information of their respective neighborlists and IDs. Considering that this could pose as a single point of failure if the machine being used for aggregating the information is being monitored, the authors also provided several suggestions to overcome this issue.

Nevertheless, the design of the proposed botnet requires the botmaster to participate actively in the management of the botnet. This design not only increases the risk of the botmaster being exposed but also do not scale in the long run.

Very little work has been seen in the context of anti-monitoring mechanisms targeting sensors. Andriess et al. reported that is often difficult to identify sensors in comparison to crawlers that are in nature more aggressive in monitoring [ARB15]. The authors also mentioned that due to the passive characteristics of sensor nodes and the difficulty in distinguishing them from bots, sensors often remain undetected. However, aggressive sensor popularization strategies such as *Popularity Boosting* [Yan+14b] can be easily detected by mechanisms to detect crawlers (cf. Section 3.4.3.2).

It is worth mentioning that there are mechanisms deployed within existing botnets that are presumably aimed at preventing potential sinkholing or takeover attacks. Sinkholing attacks often require the entire neighborlist of existing bots to be invalidated or filled up with only sinkhole servers. Since the sinkhole servers can also be generalized as sensor nodes, those mechanisms aimed at preventing such attacks are listed in the following for completeness.

1. **IP-based filtering :** Most botnets, including Sality and ZeroAccess, have IP filtering mechanisms that prevent multiple sensors sharing a single IP to infiltrate a botnet. Therefore, this mechanism prevents an organization that does not have a large pool

of unique IP addresses, e.g., IPv4 addresses, from carrying out sinkholing attacks on the botnet. GameOver Zeus implements a more strict filtering mechanism that ensures that there is only a single entry within any /20 subnet [And+13].

2. **Local reputation mechanism** : Bots in Sality use a local reputation mechanism that keeps track of the behavior of their neighbors based on the number of valid replies received when probed (cf. Section 4.2.2). This mechanism slows down the rate of deployment of sensor nodes throughout the botnet as bots in Sality prefer existing and responsive neighbors over new bots.
3. **High-frequency swapping of neighborlist entries** : ZeroAccess uses a very small MM-cycle interval (cf. Section 4.3.2) to ensure that all entries in its primary neighborlist are probed and cycled at a high rate. As a consequence, on the one hand, a sensor node loses its popularity unless it continuously announces itself to other bots in ZeroAccess [Yan+14b]. On the other, sensor that continuously announce itself would incur a high overhead in terms of the amount of messages that need to be generated and processed by the sensor.

#### 3.4.3.2 Detection

Anti-monitoring mechanisms does make it more difficult to monitor botnets. However, since this field is an arms race between the researchers and botmasters, it is only a matter of time before workarounds to circumvent or tolerate such mechanisms are available [Kar+15]. Therefore, it is important that a detection mechanism is also in place to detect ongoing monitoring activities.

In the following, detection mechanisms for crawlers are presented:

1. **Rate-limitation of requesting neighborlists** : GameOver Zeus introduced a simple rate-limitation mechanism to detect crawling activities locally [And+13]. For that, a bot keeps track of the number of connections or messages from each observed IP address within a sliding window of 60 seconds. If any IP address contacts a bot more than six times within an observation period, the IP address is flagged as a crawler and remediation actions like blacklisting (cf. Section 3.4.3.3) are taken immediately. It is believed that the threshold value is set high enough, i.e., 6, to take into account possible false positives due to multiple bots sharing the same IP, e.g., as a result of NAT.

Crawling is still possible although it requires more effort and longer delays in between successive crawls of the same node. However, significant network noise and bias is introduced in the resulting botnet snapshot as more time is required for obtaining it. To circumvent this detection mechanism, a distributed crawling from a pool of unique IP addresses is required. The available IP addresses can be rotated among the crawlers to allow parallel crawling of bots without triggering the detection mechanism.

2. **Collaborative detection mechanism**: Andriesse et al. proposed a crawler detection mechanism that detects protocol anomalies that may result from improper protocol (re)implementations of existing P2P botnets [ARB15]. However, this can be easily circumvented by strictly following the botnet protocol.

In addition, the authors also proposed a crawler detection approach that uses multiple colluding sensors to detect a crawler. This approach correlates the number of sensor nodes being contacted by a node and classifies one as crawler if the number of contacted sensors exceeds a certain threshold, e.g., maximum neighborlist size. The authors evaluated this mechanism with a deployment of up to 256 sensors in existing P2P botnets like GameOver Zeus and Sality. Hence, this detection mechanism can be easily implemented by botmasters on a large scale in the whole botnet to detect crawlers.

Andriesse et al. reported that sensors are a more stealthy monitoring mechanism than crawlers due to their indistinguishableness from benign bots [ARB15]. Although many sensors are observed to be highly popular, i.e., known by many bots and highly reliable in the botnet [Böc+15], this behavior is indistinguishable from popular bots. Due to this reason, not much work has been done previously in the scope of detecting sensors that are deployed in botnets. In contrast, as one of the major contribution of this dissertation, Section 6 will introduce three mechanisms to detect sensor nodes.

#### 3.4.3.3 *Response*

After the detection of monitoring activities a response can be initiated. There are several possible actions to be implemented in botnets:

1. **Static blacklisting**: Many botnets, e.g., GameOver Zeus, are shipped with a list of entries which consists of IP addresses of organizations known to monitor botnets. Based on this list, bots will refuse to communicate to any requests originating from those blacklisted IPs.
2. **Automated blacklisting**: GameOver Zeus also deploys an additional variation of the blacklisting mechanism. It does not rely upon the botmaster to pro-actively update the list of IPs to be blacklisted. The botnet uses a rate-limiting mechanism (cf. Section 3.4.3.2) instead to identify crawler activities and subsequently blacklist that IPs automatically [And+13]. Take note that the blacklisted entries are only maintained locally and not propagated further to other bots or the botmaster.
3. **DDoS attack**: A more aggressive response is the denial-of-service attacks on the IP address or network of detected monitoring node. This has been observed with the GameOver Zeus botmasters who retaliated in response to the sinkholing attempts of

their botnet [CER13]. Similar observations have also been reported by researchers on the *Storm* botnet [Ste07].

### 3.5 CHAPTER SUMMARY

This chapter has introduced requirements to botnet monitoring and presented a formal model for botnets that is used throughout this dissertation, i.e., Chapter 4–7. In addition, this chapter thoroughly analyzed the state of the art in existing botnet monitoring mechanisms (cf. Section 3.3); namely honeypots, crawlers, and sensors. From the analysis, it can be concluded that although honeypots can be deployed quickly to monitor botnets and require only minimal effort, their monitoring coverage are limited compared to the other mechanisms. In particular, crawlers are able to enumerate and capture the inter-connectivity information of many bots but with a drawback of missing out some portion of bots, e.g., those behind NAT. Meanwhile, sensors are more effective in enumerating bots compared to crawlers but often fail in capturing the inter-connectivity among bots. In order to obtain best results, both crawlers and sensors can be deployed to complement each other.

However, many of the previously proposed monitoring mechanisms were found to be not compliant with most of the non-functional requirements proposed in Section 3.1.2. Table 2 provides a summary of the existing monitoring mechanisms with respect to the compliance of the proposed requirements (cf. Section 3.1.1 and Section 3.1.2).

This chapter has also presented a thorough discussion in Section 3.4 on the three major challenges that need to be considered in botnet monitoring. These challenges, if left unaddressed, could impede the effectiveness of a monitoring mechanism. This would also not comply to the non-functional requirements proposed in Section 3.1.2.

Firstly, the dynamic nature of P2P botnets introduces significant amount of noise which may skew measurement results due to the presence of churn, diurnal effects, and IP address aliasing issues. State of the art reported that the usage of high-frequency crawling in combination with a long-term crawling can minimize bias introduced by churn and diurnal effects in P2P botnets. In addition, bias from IP address aliasing can be further reduced (or eliminated) using UIDs of botnets; if the UIDs are persistent and unique.

Secondly, the noise introduced by unknown third party monitoring activities (cf. Section 3.4.2) could also distort the resulting monitoring data. This particular challenge has seen very little attention from the research community in the context of P2P botnets. Analysis of the state of the art indicated that most of the existing botnet monitoring mechanisms were implemented only on a *best-effort* basis, and, as such, may have had their measurements tainted by noise originating from unknown monitoring activities, e.g., abnormally high uptime of sensor nodes.

Thirdly, anti-monitoring mechanisms which impede the performance of the monitoring mechanisms (cf. Section 3.4.3) also pose itself as a major hurdle for botnet monitoring. Fortunately, most of the exist-

P2P Botnet Monitoring Mechanisms											
	Requirements	Honeypots	Crawlers					Sensors			
			Glovenet [Wan+09]	StormCrawler [Hol+08]	Stormdrain [KLE08]	Nugache Crawler [DD08]	Rosow et al. [Ros+13]	SPTracker [Yan+14b]	Kang et al. [KCTL09]	Rosow et al. [Ros+13]	SPTracker [Yan+14b]
Functional	<b>Generic</b>	✗	ϕ	ϕ	ϕ	ϕ	✓	✓	ϕ	✓	✓
	<b>Protocol Compliance</b>	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	<b>Enumerator</b>	ϕ	✓	✓	✓	✓	✓	✓	✓	✓	✓
	<b>Neutrality</b>	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	<b>Logging</b>	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Non-Functional	<b>Scalability</b>	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
	<b>Stealthiness</b>	✓	-	-	-	-	✓	✓	-	✓	✓
	<b>Efficiency</b>	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
	<b>Accuracy</b>	✗	ϕ	ϕ	✓	✓	ϕ	ϕ	ϕ	✗	✗
	<b>Min. Overhead/Noise</b>	✓	✓	✗	✓	✓	ϕ	ϕ	✓	✓	✓

Table 2: Summary of P2P botnet monitoring mechanisms regarding their compliance with the requirements specified in Section 3.1. Checkmark symbols ✓ indicate the fulfillment of these requirements, crossing symbols ✗ indicate their non-fulfillment, average symbols ϕ indicate a partial fulfillment, and dash symbols - indicate not applicable.

ing anti-monitoring mechanisms observed deployed in the wild are still in their infancy in terms of their effectiveness. Although some of them can be circumvented or tolerated by the existing monitoring mechanisms, it may just be a matter of time before more advanced countermeasures are implemented by botmasters to raise the stakes. Along that line, this dissertation will propose advanced countermeasures from the perspective of a botmaster in Section 5.2 and 6.1 to anticipate the retaliation of the botmasters against botnet monitoring.

Concluding, crawlers and sensors seemed to be the only viable solution to monitor P2P botnets in an effective and efficient manner. However, more effort has to be taken to ensure the stealthiness, efficiency, and accuracy of the monitoring mechanisms is improved to obtain high quality monitoring data. Therefore, future monitoring mechanisms need to also carefully consider and address the various challenges discussed in Section 3.4 to ensure an effective botnet monitoring that can be useful for further steps such as botnet takedown attempts or malware cleanup campaigns.

Most P2P-based botnets implement similar Membership Maintenance (MM) mechanisms to ensure bots remain connected among one another in a distributed manner. MM mechanisms differ mainly in the parameters initialized for each of the botnets. Nevertheless, the mechanisms themselves are often implemented within the botnet's malware binary using custom encryption and encoding. Due to nature of existing botnets in using highly customized communication protocols and designs, the botnets need to be reverse engineered in order to understand their MM mechanism and the utilized communication protocols. The reverse engineering information is later utilized by botnet monitoring mechanisms to interact with bots in a botnet.

Although there are many available resources describing the anatomy of existing P2P botnets, they are only described on a very coarse-grained level. However, a detailed understanding of a botnet is often required to perform botnet monitoring activities. Therefore, own reverse engineering work is also often required to complement the existing literature for a better understanding of the inner-workings of a botnet.

For the purpose of this dissertation, three botnets were picked as case studies: GameOver Zeus, Sality, and ZeroAccess. These botnets are selected not only because they are some of the most prevalent P2P botnets but also because they deployed anti-monitoring strategies as discussed in Section 3.4.3 to impede botnet monitoring. The first three sections of this chapter (Section 4.1–4.3) describes the MM mechanism of GameOver Zeus, Sality, and ZeroAccess using the formal model presented in Section 3.2. In addition, botnet-specific details that are useful for discussion in the later part of this dissertation are also highlighted based on own reverse engineering results. The results not only managed to validate the findings of other work on these botnets, but also provided new insights that were important foundation for the works presented in this dissertation. The parts on Sality and ZeroAccess draw from a Master thesis supervised by the author of this work [Haa15]; the part on GameOver Zeus draws from a collaboration with Dr. Christian Rossow at Saarland University, Germany [Kar+15]. Finally, Section 4.4 summarizes this chapter.

#### 4.1 DISSECTING GAMEOVER ZEUS

GameOver Zeus or also known as *P2P Zeus* is a variant of the infamous banking trojan Zeus first observed in the wild around September 2011 [And+13]. GameOver Zeus has also been dubbed as one of the most sophisticated P2P botnets that have been seen in the wild [Abu11]. Detailed technical descriptions of this botnet are available as published technical reports and scientific articles in [CER13; Ros+13; And+13; Abu11].

In the following, Section 4.1.1 describes the bootstrapping process of bots in GameOver Zeus. Section 4.1.2 discusses in depth the membership management mechanism of this botnet. Finally, Section 4.1.3 introduces the blacklisting mechanism used by this botnet.

#### 4.1.1 Bootstrapping Process

Upon infecting a new machine, the malware of GameOver Zeus generates a 160-bits unique identifier (UID) based on the hash value of the concatenated strings of the operating system's *ComputerName* and the *VolumeID* of the first hard-drive in the infected machine. Since the same UID is always reproducible as long as the mentioned variables do not change, the UID is persistent through reboots as well. This UID is stored and heavily used throughout the communication between other bots in GameOver Zeus as described in Section 4.1.2 and can be represented also as a 40-hexadecimal characters string.

No	IP Address	Port	UID
1	123.100.12.201	25235	45d5f530d28f49...<truncated>
2	214.86.57.2	15687	c89d3abf771315...<truncated>
...	...	...	...
50	150.80.86.87	29001	d1649c62b94280...<truncated>

Table 3: Example of a GameOver Zeus bootstrap/neighborlist

The bots are supplied with a *bootstraplist* embedded in their binary which consists of 50 entries of other existing infected machines or bots. The information in this list consists of a tuple of *IP Address*, *Port* number and a *UID* for each entry. After successfully infect a machine, a bot utilizes this list to bootstrap itself into the botnet overlay as described in Section 4.1.2.3. This list also effectively becomes the initial NL of the bot as depicted as an example in Table 3 with a maximum of 50 entries, i.e.,  $NL^{MAX} = 50$ . Next, the mechanism to maintain this list is detailed.

#### 4.1.2 Membership Maintenance Mechanism

Bots in GameOver Zeus carry out their maintenance activities periodically every 30 minutes. Within each membership maintenance cycle, i.e., MM-cycle, a bot probes for the responsiveness of its neighbors for up to five times using the *probeMsg* as introduced in the formal model (see Section 3.2). This message is used by a bot to probe the responsiveness or availability of its neighbors. Take note that the *probeMsg* message is also commonly referred to as the *VersionRequest* message in other literature [Ros+13; CER13] as the message is also used to query and exchange the latest botmaster update(s) from the neighbors.

A valid response that is received to a sent *probeMsg* indicates that a particular neighbor is responsive, i.e., being online. If a neighbor

remained unresponsive for five consecutive attempts, it is discarded from the NL and the probing process is continued with the next neighbor in the NL. If a bot has less than 50 responsive neighbors at the end of the MM-cycle, it looks into a queue that contains information about all senders of unsolicited request messages, i.e., request messages initiated by other bots during their MM-cycle, that were successfully processed by the bot. Then, the bot sends a probe message to each of the candidate that is not already in the NL of the bot. If a valid response is received and the NL is not full, i.e.,  $|NL_v| < NL^{\text{MAX}}$ , the probed bot is added into the bot's NL.

In addition, another mechanism to refresh the NL kicks in after every sixth MM-cycle or 180 minutes once if the NL is low on entries, i.e.,  $|NL_v| < 25$ . After considering all the senders of unsolicited requests as potential neighbors, the bot actively requests for new neighbors from its responsive neighbors using the message *requestL* (see Section 3.2). Bots in GameOver Zeus include their UID or *key s* in every sent *requestL* message. Upon receiving such a request, a bot replies the message by returning ten entries from their NL, which are selected based on a *neighbor selection criteria* as described in the following.

#### 4.1.2.1 Neighbor Selection Criteria

---

**Algorithm 1** : *processRequestL(s)*

---

```

1 for  $i = 0; i < l \ \&\& \ i < |NL|; i++$  do
2    $L[i] \leftarrow NL[i]$ 
3 for  $i = l; i < |NL|; i++$  do
4   for  $j = 0; j < l; j++$  do
5     if  $XOR(NL[i], s) < XOR(L[j], s)$  then
6        $L[j] \leftarrow NL[i]$ 
7       break
8 return L

```

---

Parts of the details presented in the following has appeared in [Kar+15]. Bots that need information about other bots in the network use the *requestL(s)* method to ask their neighbors, where *s* is the key of the requesting bot. However, it is important to note that *s* can also be generated or spoofed, as long as it is a valid key, i.e., a 160-bit key. On receiving a valid NL request, a bot returns a subset of its NL of size *l*, in GameOver Zeus usually  $l = 10$ , which are closer to key *s* of the query with regards to the Kademia-like XOR-distance using the method *processRequestL(s)* [Kar+15]. More precisely, as detailed in Algorithm 1, the queried node replies as follows: it first constructs a list *L* containing up to the first ten elements listed in its NL *NL* (Line 2). Then, it iterates over all remaining elements in *L* (Line 3). The key of each element *L*[*i*] is compared to the elements of *NL* under consideration (Line 5). As soon as the algorithm finds an element *L*[*i*] with a *smaller* XOR-distance to *s* than *NL*[*j*], *NL*[*j*] is replaced with *L*[*i*] (Line 6).

In this fashion, entries or keys with closer XOR-distance are more likely to be returned (Line 5), but only the entry with the *closest* key to  $s$  is *guaranteed*. The algorithm may not return the second-closest key if it is the first element in the initial list NL (cf. Line 2); in this case, if the closest key is stored at an index larger than 9, it will be definitely compared to the second largest key at index 0 and, used as a replacement for the second largest key at index 0, and not considered further. Obviously, other constellations may exist which may lead to some of the closest-ten nodes to be discarded. In summary, the algorithm will return close-by nodes but not necessarily the closest ten. Please take note that the order of the entries stored in a bot's NL is non-deterministic, e.g., they can be sorted by the XOR-distance of the neighbor to the bot or by the timestamp an entry was last updated.

This particular neighbor selection criterion introduces bias in the entries within a bot's NL towards a bot's key. Although this selection criterion is similar to the DHT implementation in Kademlia, GameOver Zeus remains an unstructured P2P botnet. Next, the mechanism to facilitate insertion of new entries within a bot's NL is elaborated further.

#### 4.1.2.2 *Inserting New Entries*

Whenever a bot decides to add new neighbors, from either the queue consisting of senders of unsolicited messages as explained in Section 4.1.2 or by actively requesting new neighbors as described in Section 4.1.2.1, each new neighbor candidate goes through a two-step *sanitization* phase (for the newest variant of GameOver Zeus). The candidates need to satisfy the following conditions:

1. **Port Range:** A candidate's source port is required to be within the range 10,000 – 30,000.
2. **Sub-network Range:** Only one IP entry is allowed in the NL for every /20 sub-network.

The algorithm discards candidates that failed to fulfil any of the conditions and appends the rest in the NL.

#### 4.1.2.3 *Node Announcement and Update of Existing Entries*

Since a botnet needs to include new infections within the botnet overlay, the botnet MM uses the *announceMsg* to announce the arrival of the new bots to others in the overlay (see Section 3.2). In the case of GameOver Zeus, the purpose of a *announceMsg* is incorporated within the *probeMsg* sent by bots. A bot that receives a valid *probeMsg* will consider the sender of the message to be added directly into the NL if the sender is unknown and the NL is not full. Alternatively, if the NL is full, the bot adds the sender to a queue which consists of potential candidates that can be considered as neighbors if the size of the NL falls below a threshold value of  $|NL| \geq 25$ . Therefore, when a newly infected machine attempts to probe entries in the bootstraplist (see Section 4.1.1), the bot possibly also inserts itself as a potential candidate for future consideration. If a bot inserts this newly infected machine in

its NL, information about the new infection will be propagated further when other bots request for new neighbors.

To address the IP address aliasing issue (see Section 3.4.1), GameOver Zeus uses a reboot-persistent UID to uniquely identify bots. This UID is transmitted as a field within all communication messages of bots in GameOver Zeus. Whenever a bot receives a *probeMsg* that consists of a UID already known to the bot (in the NL) but with a different IP address and/or port number, the bot updates the entry with the new information. This update mechanism ensures that a bot eventually gets to update the entries of neighbors that rejoin the network with new IP addresses or port numbers.

#### 4.1.3 Blacklisting Mechanism

GameOver Zeus implements a two-fold blacklisting mechanism in the bots to deter aggressive crawling activities of known and unknown researchers. Firstly, all bots have a static list of IP addresses that they refuse to communicate to, e.g., security organizations known to perform botnet monitoring [CER13]. Secondly, bots also implement a simple local rate-limiting mechanism to detect and blacklist aggressive crawlers [CER13]. For this, a bot that receives more than six request messages from a single IP address within a sliding window of 60 seconds, automatically adds the IP address into a list of blacklisted IP addresses and will permanently refuse to communicate further [And+13]. This list is maintained locally and not exchanged with other bots.

## 4.2 DISSECTING SALITY

Sality is a botnet family that propagates through file-infection. It has been around since mid-2003 [Fal11]. This botnet family has evolved throughout the years from traditionally communicating to the botmaster via emails, to a complete P2P-based communication in early 2008 [Fal11]. Based on the initial reporting on the P2P variant of Sality, there could have been up to four versions of this P2P botnet. Many researchers distinguish different variants of Sality by the specific version numbers in transmitted communication messages. However, none have reported the existence of any *Version 1* of Sality.

The first version of P2P Sality observed in the wild transmitted "Version 2" in its communication messages. Around early 2009, Sality version 3 has been first seen and said to be the largest variant of the P2P Sality [Fal11]. This variant still remains active at the time of writing. Falliere reported that differences between the protocols implemented in version 2 and 3 are minimal [Fal11]. Around late 2010, version 4 of this botnet was first seen. This variant introduces new features, leading to improved security and robustness, by addressing some of the weaknesses found in the earlier versions of the botnet. Nevertheless, most of the communication protocols of the botnets as mentioned earlier remain the same, except for the transmitted version number.

Since the information that is needed to understand the various work within the scope of this dissertation is common across these different botnets, i.e., communication protocols, all variants of P2P Sality are henceforth referred to as Sality in the remainder part of this dissertation, unless mentioned otherwise. Other detailed technical description of this botnet for interested readers is available as published technical reports and scientific publications in [Fal11; Kle15].

In the following, Section 4.2.1 describes the bootstrapping process of bots in Sality. Meanwhile, Section 4.2.2 discusses in depth the membership management mechanism of this botnet.

#### 4.2.1 Bootstrapping Process

Upon infecting a new machine, a Sality malware first starts to listen on a User Datagram Protocol (UDP) socket for incoming request messages following the node announcement procedure explained in Section 4.2.2.1. This socket or port number is derived based on the operating system's ComputerName using a simple built-in algorithm. Also, bots in Sality utilize UIDs for some of the inter-bot communication messages. However, the UIDs are not persistent over reboots. Instead of generating the UIDs themselves, bots in Sality obtain their UID through a process that involves an existing superpeer assigning it one (explained later in Section 4.2.2). The assigned UIDs are integer values between  $0 - 2.0 \times 10^7$ . In the bootstrap phase, this UID is initialized to a default value of 0.

No	IP Address	Port	UID	GoodCount	LastOnline
1	123.100.12.201	25235	$1.8 \times 10^7$	65	0
2	214.86.57.2	15687	$1.9 \times 10^7$	45	2356
...	...	...	...	...	...
1000	150.80.86.87	29001	$1.1 \times 10^7$	21	5561

Table 4: Example of a Sality bot's NL

Next, the bootstraplist which is passed on from the previous file infector, i.e., bot, is used to initialize the NL of the new bot. This list typically consists of up to 1,000 entries. A bot executes this initializing step only if there is no existing NL present within the machine's registry file. The bot's NL can hold up to a maximum of 1,000 entries, i.e.,  $NL^{MAX} = 1,000$ , and has a structure as depicted in an example in Table 4. Initially, the values for all fields in this list is set to a default value of 0. Then, the bot copies the *IP Address* and *Port Number* of entries from the bootstraplist into the NL. After initialization of the NL, the bot executes the bootstrapping process by directly invoking the first membership maintenance cycle. The following section details the membership maintenance mechanism, as well as the purpose of the different fields within the NL of a bot.

### 4.2.2 Membership Maintenance Mechanism

Each bot in *Salinity* utilizes a membership maintenance mechanism to ensure connectivity amongst bots with an interval of 40 minutes. Within each MM-cycle, a bot  $v$  probes all neighbors in its  $NL_v$ , one after another, in a sequential manner by executing three different but related processes.

Firstly, the bot probes the responsiveness of a neighbor using the *probeMsg* method (cf. Section 3.2) which utilizes a *Salinity*-specific *Hello* message. When a bot receives a valid response to the sent *Hello* message, the corresponding neighbor's *LastOnline* value is set to the *current* timestamp. Otherwise, the value is set to zero if an invalid reply is received or the request timed out, before the next entry in the  $NL$  is probed. The *LastOnline* is widely used within *Salinity* as a flag to indicate that a particular neighbor was responsive within the previous MM-cycle. Also, for each successful verification of a neighbor's responsiveness, the *GoodCount* of the corresponding entry is incremented by one. Similarly, when a timeout occurred, or an invalid reply is received, the bot decrements the neighbor's *GoodCount* accordingly. Over time, neighbors that are more often responsive have a higher *GoodCount* value compared to those that are not.

The *Hello* message exchange is also leveraged to ensure the latest command from the botmaster is disseminated to all bots, i.e., from bot-to-bot. An update command from the botmaster comes in the form of a digitally signed and encrypted file which is also known as a *URLPack* [Fal11]. A *URLPack* consists of a list of Uniform Resource Locators (URLs) which usually host additional malicious binaries that needs to be downloaded and executed frequently by the bots in their local machine, i.e., approximately twice in each hour. Within each *Hello* message, the newest *sequence number* of the *URLPack* known to a bot is always transmitted.

By comparing the sequence number transmitted in the received messages, bots in *Salinity* can update themselves with the latest update from the botmaster. Consider the following scenarios where  $Bot_x$  is sending a *Hello* message to  $Bot_y$ :

1.  **$Bot_x$  has an older *URLPack* than  $Bot_y$ :** Upon inspecting the sequence number of the received message,  $Bot_y$  will notice that  $Bot_x$  has an older *URLPack* installed. Therefore,  $Bot_y$  responds to the received message by attaching the latest *URLPack* which would be applied by  $Bot_x$  upon receiving the reply.
2.  **$Bot_x$  has a newer *URLPack* than  $Bot_y$ :** Upon inspecting the sequence number of the received message,  $Bot_y$  will notice that  $Bot_x$  has a newer *URLPack* installed. Therefore,  $Bot_y$  responds with a message by stating the sequence number of its currently installed *URLPack*. Upon receiving the message and noticing that  $Bot_y$  has an older *URLPack*,  $Bot_x$  sends an additional *Hello* message to  $Bot_y$  that attaches the latest *URLPack*.  $Bot_y$  can then apply the newer update accordingly.

3. **Both bots have the same *URLPack*** : No steps are taken if both bots have the same *URLPack*.

After verifying the responsiveness of a neighbor, the bot checks if it has a status of either a superpeer or non-superpeer (see Section 4.2.2.1). Bots in Sality use the default UID value of  $UID = 0$  as an indicator that a bot's status has not been tested. Similarly, a non-zero value indicates that a bot's capability is tested and this second process within an MM-cycle is omitted. Section 4.2.2.1 elaborates this testing process in detail.

Finally, if the number of entries in the NL is low, i.e.,  $|NL_v| \ll 980$ , at the beginning of the maintenance cycle, a Sality-specific *Neighborlist Request* message ( $NL_{Req}$ ) is sent to the responsive neighbor using the method *requestL* (cf. Section 3.2). Bots receiving an  $NL_{Req}$  will respond with a *Neighborlist Reply* ( $NL_{Rep}$ ) message containing information on one randomly picked bot from a list of *only* responsive neighbors. For this, a temporary list is first constructed from the main NL but consisting of only entries that have non-zero *LastSeen* values, i.e., responsive neighbors. Upon receiving an  $NL_{Rep}$  reply message, the returned entry can be considered as a potential candidate for the bot's NL as elaborated in Section 4.2.2.2. It is also worth noting that all entries within a Sality's NL are only superpeers. After completing the three processes for the picked neighbor, the next neighbor is probed until all neighbors within the NL have been probed.

After all neighbors are probed, a bot conducts an additional clean-up step on the NL if the size of the list was at least 500 at the beginning of that particular MM-cycle. This clean-up process discards all entries that have low *GoodCount* values, i.e.,  $GoodCount < 30$ , or UIDs that are not within a superpeer's assigned range, i.e.,  $UID > 1.6 \times 10^7$  (see Section 4.2.2.1). Finally, the responsible thread for the maintenance sleeps for the next 40 minutes before invoking a new MM-cycle.

Also take note that Sality utilizes a new OS-allocated socket or port for each request that is sent out. This design could have been chosen to prevent any potential replay attack against Sality.

#### 4.2.2.1 Testing Superpeer Capability and Node Announcement

Since there is no centralized infrastructure in Sality, the testing of superpeer capability is done with the help of other existing superpeers within the overlay. For that,  $Bot_x$  first sends a botnet-specific *Node Announcement Request* message via the *announceMsg* method (cf. Section 3.2) to  $Bot_y$ , i.e., a responsive neighbor. Within the sent message,  $Bot_x$  includes information of which UDP port it listens for unsolicited requests (cf. Section 4.2.1). Upon receiving this request message,  $Bot_y$  sends a *Hello* message to the IP address of  $Bot_x$  using the port specified within the received request message.

The premise of this decision is; if  $Bot_x$  is publicly reachable from the Internet, it should be able to respond successfully to the received *Hello* message. Hence, a valid response to the probe message indicates  $Bot_x$ 's superpeer capability and the failure to respond, e.g., timeout occurred, signifies its incapability. Based on the verification,  $Bot_y$

responds to the initial *Node Announcement Request* with a reply that includes a value: a random value between  $[0 - 1.6 \times 10^7)$  when  $Bot_x$  is not contactable on the listening socket, i.e., non-superpeer, or  $[1.6 \times 10^7 - 2.0 \times 10^7]$  otherwise. If  $Bot_x$  is identified as a superpeer candidate,  $Bot_y$  additionally adds  $Bot_x$  into its NL as well (cf. Section 4.2.2.2). This way, to further propagate the information about itself being a potential superpeer to other bots that may need additional neighbors,  $Bot_x$  relies only upon  $Bot_y$ .

Finally, upon receiving the *Node Announcement Reply* message from  $Bot_y$ ,  $Bot_x$  uses the returned value as its own UID and omits to repeat this process in subsequent MM-cycles until the next reboot. However, in case a reply was not received from  $Bot_y$ , i.e., a timeout occurred,  $Bot_x$  sets its UID to zero and repeats this testing procedure with the next responsive neighbor.

#### 4.2.2.2 Inserting New Entries

There are two different scenarios where a bot attempts to add a new neighbor: 1) through NL exchange when having a low number of neighbors (cf. Section 4.2.2) and 2) when testing a bot for superpeer capability (cf. Section 4.2.2.1). Algorithm 2 describes the method *insertNeighbor()* for both scenarios with an input parameter *entry*, i.e., IP address and port number. To distinguish the different scenarios, Sality uses a flag parameter *isTested* to indicate if the entry is being considered after being tested for its superpeer capability or not.

First, the bot checks if the current NL is already full (Line 1) and returns if the method was invoked within Scenario 1. Otherwise, the algorithm proceeds and checks if the IP address of *entry* is already present within the NL (Line 3). In case the IP address is present, it is checked whether the corresponding port in the reply matches with the existing entry in the NL (Line 4). If the ports do not match, the bot additionally checks (Line 5) if the entry in the NL was marked *offline*, i.e., *LastSeen* = 0, during the last MM-cycle or if the method was invoked within Scenario 2. If either one of the conditions is satisfied, the old port of the entry is replaced with that in the *entry* (Line 6) and the method returns. This entry updating feature allows an existing bot that reappeared on a different port to be updated by existing bots by retaining the old entry along with its corresponding *GoodCount* value as well.

However, if the address is unknown and the method was invoked from within Scenario 2 when having a full NL, the bot additionally removes one entry from its NL that has the lowest *GoodCount* value to make room for this new candidate (Line 10). Finally, *entry* is appended to the end of NL (Line 11).

### 4.3 DISSECTING ZEROACCESS

ZeroAccess is a malware dropper family that is used to distribute additional malware. It primarily focuses on financial fraud through pay-per-click (PPC) advertising [NG13]. The botnet utilizes *plugins* as dropped

**Algorithm 2:** insertNeighbor(entry, isTested)

---

```

1 if NL.isFull() && isTested  $\neq$  True then
2   | return // NL is full
3 if entry.IP  $\in$  NL.getAllIPs() then
4   | if NL[entry.IP].Port  $\neq$  entry.Port then
5   |   | if NL[entry.IP].Status  $\neq$  Online || isTested then
6   |   |   | NL[entry.IP].Port = entry.Port // Update Port
7   |   |   | return // Nothing else to do
8 if NL.isFull() && isTested then
9   | // Make room for a new entry
10  | NL.popEntryWithLowestGoodCount()
    | // Append entry at the end
11 NL.append(entry)

```

---

modules to enable bots conducting the above-mentioned malicious activities. As of the time of writing, researchers have reported the discovery of two versions of ZeroAccess [Ros+13; NG13]: *Version 1* in May 2011 and *Version 2* in April 2012. In contrast to the former that uses TCP protocol for communications between bots, the latter version adopts UDP protocol instead. In the newer version, the set of commands utilized for inter-communication has also been reduced. This includes the removal of a command that can be exploited to launch sinkholing attack on the botnet. The removal of the command enhanced the efficiency and resiliency of the botnet [NG13]. Since it was difficult to find active bots to bootstrap in the *Version 1* of this botnet, the remainder part of this dissertation focuses only on the *Version 2* [Ros+13].

ZeroAccess *Version 2* primarily performs two types of malicious activities: Bitcoin mining and Click-Fraud. For each activity, there exist two separate networks of bots distinguished by the OS architecture of the infected machines, i.e., 32-bit or 64-bit. As such, there are four distinct networks of ZeroAccess *Version 2* as detailed in Table 5. Each of the networks distinguishes itself by the usage of distinct UDP hard-coded ports for communications.

	Ports for 32-bit	Ports for 64-bit
Bitcoin Mining	16464	16465
Click-Fraud	16471	16470

Table 5: Distinct botnets distinguished by ports in ZeroAccess *Version 2*

Around mid-2013, the botmaster issued an update to the Bitcoin mining networks that addressed a weakness highlighted by researchers which could be exploited to conduct a sinkholing attack [Ros+13]. Considering that the new update was not completely disseminated within the Click-Fraud networks, Symantec initiated a sinkholing attempt on both 32-bit and 64-bit networks in July 2013 and liberated about 500,000 bots from their botmaster [Sym13]. However, as at time of

writing, all four networks were observed to have received the new update and there are still bots unaffected by the sinkholing attack. These bots, which are still interconnected and operational in all four networks, survived like GameOver Zeus and Sality due to the resiliency and robustness of thoroughly designed P2P botnets.

Since the information that is needed to understand the various work within the scope of this dissertation requires only the understanding of the communication protocols that is common across these different networks, all networks of ZeroAccess *Version 2* are henceforth referred to simply as ZeroAccess in the remainder part of this dissertation unless mentioned otherwise. Detailed technical description of this botnet is available as published technical reports and scientific publications in [NG13; Sym13; Wyk12; Ros+13].

In the following, Section 4.3.1 describes the bootstrapping process of bots in ZeroAccess. Section 4.3.2 discusses in depth the membership management mechanism of this botnet. Finally, Section 4.3.2 elaborates on a particular design (flaw) of this botnet that enables UDP-hole punching to be conducted to contact non-superpeers.

#### 4.3.1 Bootstrapping Process

Upon infecting a new machine, a ZeroAccess malware first starts to listen on a fixed UDP socket for incoming request messages as depicted in Table 5. In addition, the malware also uses a dedicated OS-allocated UDP socket to send out requests. Bots in ZeroAccess also generate a UID upon initialization that is not persistent over reboots. Next, the bots download all available *plugins* that enable add-on features as intended by the botmasters, e.g., Click-fraud and Bitcoin mining, and store them on the infected machine [NG13].

Index	IP Address	LastOnline
0	123.100.12.201	2564
1	214.86.57.2	1082
...	...	..
255	150.80.86.87	220

Table 6: Example of a ZeroAccess *Primary* NL

According to the reverse-engineering results of recent malware variants<sup>1</sup>, the bots have three types of NLs: *primary*, *secondary* and *backup* list. The bot's *primary* NL can hold up to a maximum of 256 entries, i.e.,  $NL^{MAX} = 256$ , and has a structure as depicted in an example in Table 6. The *primary* NL only maintains the IP address of a neighbor as all bots listen on a dedicated port that is unique to the network they reside in. The bootstrapping list of the malware that typically has up to 256 entries is used to initialize the *primary* NL of the new bot. The remaining lists which have a significantly larger length, i.e.,  $16 \times 10^6$  entries, are initialized empty during infection and information of all respon-

<sup>1</sup> md5 = ea039a854d20d7734c5add48f1a51c34

sive and known bots throughout the lifetime of the particular bot is continuously added/updated in both of the lists. However, only the *backup* list is persistent over reboots. The presence of this list can allow a bot to recover from any potential sinkhole attempts using bots that were responsive in the past (cf. Section 4.3.2).

After initialization of the NLs, the bot executes the bootstrapping process by directly invoking the first membership maintenance cycle. The details of the membership maintenance mechanism, as well as the purpose of the *LastOnline* field in the *primary* NL of a bot, is detailed next.

#### 4.3.2 Membership Maintenance Mechanism

Each bot in *ZeroAccess* utilizes a membership maintenance mechanism to ensure connectivity amongst bots with an interval of 256 seconds. Within each MM-cycle, a bot  $v$  sequentially probes each neighbor in its *primary*  $NL_v$ , and optionally an additional two bots from the *secondary* and *backup* lists, every one second using the *probeMsg* method (cf. Section 3.2). In other words, a bot starts probing entries from index 0 up to 255 with an increment of one every one second. Upon reaching the final entry, i.e.,  $index = 255$ , the iteration process wraps up to start again from  $index = 0$ .

Maintenance of the NL in the bots relies only on the exchange of two *ZeroAccess*-specific messages between bots: *getL* and *retL*. As explained above in Section 4.3.1, each bot listens on the botnet-specific port for unsolicited requests, the so-called *server port* and sends such requests from an OS-allocated but fixed UDP port or a so-called *client port*. As such, all probing messages originate *only* from the client port. Figure 3 depicts such a probing process between two bots:  $Bot_X$  and  $Bot_Y$ .

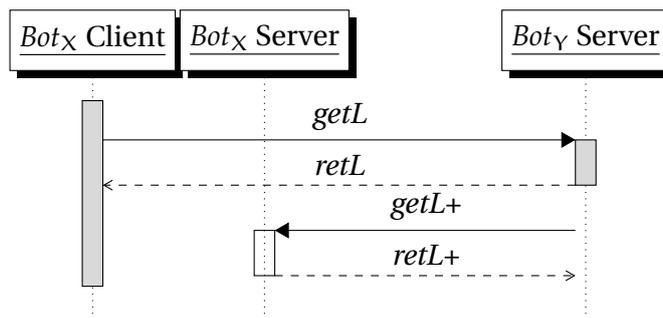


Figure 3: Message exchange for  $Bot_X$  probing  $Bot_Y$  in *ZeroAccess*.

Let's assume  $Bot_X$  probes  $Bot_Y$  for its responsiveness. Upon receiving a *getL* message,  $Bot_Y$  responds with a *retL* message that consists of a subset of its own *primary* NL and a list of all *plugins* available for download from itself, i.e., using TCP connections. Section 4.3.2.2 discusses the details on the neighbor selection criteria utilized to pick the neighbors to be included in a reply. In addition to the reply, a probe message is sent to the *server port* of  $Bot_X$ , i.e., lower part in

Figure 3. This process which serves as a method to verify if  $Bot_X$  is a superpeer requires  $Bot_X$  responding to the probe message of  $Bot_Y$  using the server port as the source port in the reply. A successful response indicates that  $Bot_X$  is publicly reachable on the Internet, i.e., superpeer.

---

**Algorithm 3** : processGetL(sender, msg)

---

```

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))

```

---

To avoid the exchange of messages to be continuously looped, a *flag* in the messages is set when being probed to indicate *no further probing is needed* as depicted in Algorithm 3. Such messages are differentiated by using *getL+* and *retL+* when referring to the set of messages that do not require additional probing.

Upon receiving either a *retL* or *retL+*, bots process the message by performing the following steps:

1. Inserts the sender in the *primary* NL as described in Section 4.3.2.1.
2. Inserts the sender and the returned neighbors (if any) in the *secondary* NL.
3. If there is, at least, one *plugin* information in the reply, inserts the sender in the *backup* NL.

Although the sender is added into all three NLs, for the remainder part of this dissertation, we only focus on the *primary* NL, which is used to select neighbors to be returned in *retL* and *retL+* messages. Take note that due to the communication design adopted by this botnet, researchers have reported that they were able to leverage them to conduct UDP hole-punching to communicate continuously to non-superpeers that first reached out to a sensor node [Ros+13].

#### 4.3.2.1 Inserting New Entries

Upon receiving a valid response, a sender's IP is added or updated within the *primary* NL of each bot. For that, the IP is used as a parameter to invoke Algorithm 4 that is responsible for handling this process. Firstly, bots check if the sender's IP exists in their *primary* NL (Line 2). If the IP exists, this entry is removed from the NL (Line 4). After removing this entry, all subsequent entries are shifted up one position to bridge the gap originating from the deletion process. Finally, the sender's IP is used to create an entry at the beginning of the NL, i.e.,  $index = 0$ . Afterward, the *LastSeen* value is set to the value of the current timestamp.

---

**Algorithm 4**: insertInPrimaryList(sender)

---

```

1 // Check if sender known
2 if sender ∈ NL.getAllIPs() then
3   | // Remove existing entry and close the gap
4   | NL.pop(sender)
5 // Push entry at the beginning of list
6 NL.push(sender)
7 // Set sender's LastSeen to now
8 NL[sender].LastSeen = getCurrentTimestamp()

```

---

Thanks to the nature of the process of inserting neighbors, the *primary* NL of ZeroAccess bots is sorted by *most-recently* responsive neighbors. Moreover, if the NL is full and a new (but non-existing) entry needs to be added, the last entry, i.e., `index = 255`, is discarded when the new entry is pushed at the beginning of the NL.

#### 4.3.2.2 Neighbor Selection Criteria

Bots need to respond to each of received probe messages, i.e., *getL* and *getL+* messages, with a *retL* message that includes a subset of neighbors that have their responsiveness most recently verified. According to the protocol of ZeroAccess, a bot can include up to 16 entries in a resulting reply but an empty reply is also valid. Since the *primary* NL of the bots is always sorted by most responsive neighbors first, a bot only needs to return the first-16 entries from the NL in the reply message.

## 4.4 CHAPTER SUMMARY

This chapter described the anatomy of three P2P botnets: GameOver Zeus, Sality, and ZeroAccess, based on the work of malware reverse engineering. Particularly, the bootstrapping and membership maintenance mechanism of each botnet was thoroughly analyzed, and the differences among them were outlined. From the analysis, existing P2P botnets are found to share similarities in terms of the common MM designs and anti-monitoring mechanisms aimed at impeding botnet monitoring.

However, some of the parameters used by the botnets differed greatly among one another, e.g., MM-interval or the size of the NL used by the botnets, as described in the following:

- **MM-interval** : The MM-interval of a botnet directly influences the rate of stale information in the NL of bots, and also the communication overhead generated by the bots. For instance, a very long interval may cause a bot to be isolated from the botnet overlay due to many of its neighbors being non-responsive, e.g., bots gone offline. However, the communication overhead caused by such long intervals are lesser and may be helpful to stay below the radar of the network administrators monitoring the net-

work traffic of the network the bot is in. Sality and GameOver Zeus utilized such long intervals with 40 and 30 minutes respectively.

In contrast, a very short interval may ensure non-responsive neighbors are quickly replaced with responsive ones. Therefore, chances of a bot with short interval to be isolated from the botnet overlay is much less than those with higher intervals. However, a short interval also implies that high communication overhead is generated in the process of frequently probing the responsiveness of the neighbors of a bot. This in turn may easily raise suspicions to the network administrators. ZeroAccess is an example of such botnet with a shorter interval, i.e., 256 seconds.

- **Size of NL:** Ensuring a bot remains connected to the botnet overlay is also often influenced by the availability of sufficient number of responsive neighbors to communicate with. Moreover, the size of the NL utilized by botnets also influences how quickly a command issued by a botmaster is disseminated throughout the botnet. In the context of the analyzed botnets, GameOver Zeus utilized the smallest size for its NL, i.e.,  $|NL| = 50$ . This is followed by ZeroAccess with a size of  $|NL| = 256$  and Sality with the biggest NL size, i.e.,  $|NL| = 1,000$ .
- **Anti-monitoring mechanisms:** Each of the anti-monitoring mechanisms implemented by the analyzed botnets are unique and has its special purpose in protecting the botnet either from being monitored or taken down. GameOver Zeus is by far the most advanced botnet among the three botnets analyzed in this chapter. It utilized an NL restriction mechanism that returns only a subset of its NL following a special node selection criteria as described in Section 4.1.2.1. In addition, the botnet also uses IP address-based filtering to ensure sensors or sinkhole server entries are not able to easily fill up the NL of a bot. Finally, GameOver Zeus also utilizes blacklisting mechanisms to refuse communicating to known or aggressive crawlers.

Sality also introduced an NL restriction mechanism that returns one random entry out of the 1,000 entries for each received NL request. Moreover, it also implemented a local reputation mechanism whereby older neighbors are preferred to new neighbors. This reputation mechanism prevents sinkholing attacks that aim to invalidate all entries easily within the NL of a bot.

Finally, ZeroAccess utilizes the short MM-interval to make it difficult for sinkholing attacks by cycling entries within the NL with a high frequency. Since most sinkholing attack requires the existing entries in the NL of a bot to be invalidated, this mechanism quickly flushes away the invalidated entries. As a consequence, a successful sinkholing attack requires a lot of resources to continuously invalidate the entries in the NL of all bots in the botnet. In addition to this design, the botnet also maintains two additional NLs on top of the main NL used for regular MM activi-

ties. These two NLs keep track of all responsive bots ever discovered since the last reboot of an infected machine. Entries within these NLs are also contacted from time to time to allow the botnet to recover even from a powerful ongoing sinkholing attack.

Many of the detailed analysis presented in this chapter serve as a foundation to other work presented in this dissertation. For instance, the neighbor selection criteria of GameOver Zeus that is presented in Section 4.3.2.2, is important to understand the work on circumventing this particular mechanism (see Section 5.1.1). In addition, the MM design of Sality and ZeroAccess as presented in this chapter is leveraged in the work on autonomously detecting crawlers in P2P botnets (see Section 5.2.2). Moreover, the communication and MM designs utilized by both Sality and ZeroAccess are also leveraged to present several novel sensor detection mechanisms in Chapter 6.

The next chapter presents work on circumventing the anti-crawling countermeasures of GameOver Zeus and further introduces advanced anti-crawling countermeasures that can be expected in the near future.

Crawlers are widely used in botnet monitoring (see Section 3.3.2) to enumerate all bots and to discover the interconnectivity among them. Such information is vital to law enforcement agencies in botnet take-down attempts. In response, botmasters introduced several anti-crawling mechanisms within existing botnets to impede monitoring activities. In particular, GameOver Zeus can be considered as the most sophisticated P2P botnet seen to date [Ros+13] due to the significant efforts that are taken to prevent monitoring activities. As discussed in Section 3.4.3, such anti-crawling mechanisms can introduce a significant amount of noise and distortion to the data gathered by monitoring. Hence, it is important to circumvent existing anti-crawling mechanisms effectively to obtain monitoring data of better quality and to anticipate future botnet advancements.

Section 5.1 presents work on circumventing the NL restriction mechanism and the automated blacklisting mechanism of GameOver Zeus from an attacker perspective, i.e., researchers and legal enforcement agencies. Section 5.2 introduces advanced anti-crawling mechanisms that aim at impeding crawling activities and on the detection of ongoing crawling activities from the perspective of botmasters. Section 5.3 presents the evaluation results of the proposed mechanisms on the newly introduced countermeasures and the advanced crawling mechanisms. Finally, Section 5.4 provides a brief discussion and summarizes this chapter. Please take note that some passages in this chapter are quoted verbatim from the following publications [Kar+14; Kar+15; Kar+16a].

## 5.1 CIRCUMVENTING ANTI-CRAWLING MECHANISMS

In this section, the anti-crawling mechanism of GameOver Zeus (see Section 4.3.2.2) are thoroughly analyzed, and methods to circumvent them are presented. The anti-crawling mechanism of GameOver Zeus is of special interest due to the fact that this mechanism is more advanced than the mechanisms of Sality and ZeroAccess.

The remainder part of this section is organized as follows: First, an algorithm to exploit and circumvent the botnet's NL restriction mechanism is presented in Section 5.1.1. This algorithm aims to retrieve the complete NL of a given bot deterministically. Second, Section 5.1.2 presents a novel crawling algorithm that heuristically attempts to enumerate all bots in a botnet by contacting only a minimum number of bots.

### 5.1.1 *Restricted NL Reply Mechanism of GameOver Zeus*

GameOver Zeus is one of the most sophisticated botnet known to date, due to the effective mechanisms adopted by the botnet to impede monitoring activities and to recover in a case of a potential take-down [Ros+13; Kar+15]. One particularly interesting aspect among the botnet defense mechanisms is the NL restriction mechanism that deterministically picks and returns a subset of neighbors, specific to the requester, when being requested (see Section 4.1.2.1). This subsection will introduce a novel algorithm called ZEUSMILKER that circumvents this restriction mechanism. For that, background information on the restriction mechanism is detailed in Section 5.1.1.1. Section 5.1.1.2 introduces ZEUSMILKER algorithm. Finally, Section 5.1.1.3 provides an in-depth analysis on the correctness and complexity analysis of ZEUSMILKER.

#### 5.1.1.1 *Background*

Each bot in GameOver Zeus maintains an NL that contains a subset of other active bots in the network (see Section 4.1.2). Bots regularly exchange subsets of these lists on a request basis to maintain and improve the connectivity of the botnet. The exchanged subsets are selected based on an *XOR-distance* metric between the unique keys of the requesting bots that are included in request messages and the neighbors in the NL (see Section 4.1.2). Hence, two legitimate bots with two different keys that request an NL from a bot may receive a totally different set of entries. Thus, a botnet crawler has to query each node multiple times using distinct spoofed keys, which decreases the performance of a crawler considerably [Kar+14].

Rossow et al. first proposed a method to circumvent this mechanism by spoofing the querying keys randomly and to hope to obtain all neighbors eventually [Ros+13]. In the following, a reliable method to provably obtain all neighbors from a bot by strategically spoofing requester UIDs or keys is presented. This method is the only work known to provide a working solution in successfully circumventing the restriction mechanism of GameOver Zeus.

#### 5.1.1.2 *ZeusMilker Algorithm*

The ZEUSMILKER algorithm that is presented in Algorithm 5 leverages the design of the GameOver Zeus restriction mechanism itself (see Section 4.1.2.1). Before elaborating the algorithm, important notations to understand the algorithm is introduced in the following.

**NOTATIONS** Each bot in GameOver Zeus is assigned a unique *key* in the form of a  $b$ -bit string.  $\mathbf{1}(i)$  is used to denote a bit string of  $i$  1s and analogously  $\mathbf{0}(i)$  denotes a string of  $i$  0s. Furthermore,  $|s|$  denotes the length of a string  $s$ , and  $\parallel$  is the concatenation operator. For two  $b$ -bit keys  $x$  and  $y$ , the function  $\text{cp}(x,y)$  returns their common prefix. An order on the set of  $b$ -bit keys is defined by associating

the key's bits  $b_{b-1} \dots b_0$  with an integer value  $\sum_{i=0}^{b-1} 2^{b_i}$ . In particular, a key  $y$  is defined bigger, smaller or equal than a key  $x$  by comparing the integer values. The operators  $+$  and  $-$  are then defined as the respective operators in  $\mathbb{Z}$ , the set of all integers. In particular, two keys  $x$  and  $y$  are called *consecutive* if  $y = x + 1 \pmod{2^b}$ . Finally,  $I(x, y) = \{x + 1, \dots, y - 1\}$  is used to denote the set of all possible keys 'between'  $x$  and  $y$ . Note that the set is empty if  $y \leq x$ .

ZEUSMILKER aims on retrieving the complete NL of a bot using the method  $requestL(s)$  as described in the formal model (see Section 3.2). Algorithm 5 achieves this goal by subsequently discovering pairs of keys  $(x, y)$  such that the NL is guaranteed not to contain any keys in  $I(x, y)$  and thus  $NL \cap I(x, y) = \emptyset$ . The algorithm terminates if no sets of  $I(x, y)$  can contain additional and yet unknown keys, guaranteeing that the list of returned keys  $L$  is identical with NL. The NL of bots is assumed to remain static during crawling. This assumption is valid because the periodic MM cycle during which NLs stay constant is 30 minutes in GameOver Zeus (see Section 4.1.2). As GameOver Zeus has a rate-limiting mechanism (see Section 4.1.3) in place, several unique IP addresses are assumed to be utilized to circumvent this mechanism, i.e., crawling bots in a distributed manner using multiple unique IP addresses.

The neighbor selection mechanism of GameOver Zeus (see Section 4.1.2.1) returns ten entries that are close to the supplied key  $s$  through the method  $requestL(s)$ . The selection of the returned neighbors is based on a well-known XOR-distance metric that was introduced in Kademia [MM02]. However, the design of the GameOver Zeus mechanism is such that only the closest key is always guaranteed to be returned. This observation along with the nature of an XOR operation is leveraged by ZEUSMILKER to circumvent the GameOver Zeus restriction mechanism. Before discussing Algorithm 5 in detail, a short explanation is provided on how spoofing with two consecutive keys  $s_1, s_2 \in I(x, y)$  results in a set  $I(x, y)$ , such that all keys in  $I(x, y)$  are *not* contained in NL. Consider the left-hand side of Figure 4: Here, all possible  $b$ -bit keys are represented in the form of a ring. Note that all the keys in the right half of the ring are closer to  $\mathbf{o}(b)$  than  $\mathbf{1}(b)$  with regard to the XOR-distance, whereas all keys on the left half are closer to  $\mathbf{1}(b)$ . Similarly, when considering only the keys on the right half, the keys in the upper right quarter are closer to  $\mathbf{00}\|\mathbf{1}(b-2)$  than to  $\mathbf{01}\|\mathbf{o}(b-2)$ , whereas the keys in the lower right quarter are closer to  $\mathbf{01}\|\mathbf{o}(b-2)$ . In this manner, one can successively divide keys into sets according to their closeness in their XOR-distances to the supplied or spoofed keys. This division is leveraged to identify keys *not* contained in the NL and the keys *possibly* contained in NL as follows. Let

$$s_1 = c\|\mathbf{0}\|\mathbf{1}(i), \quad s_2 = s_1 + 1 = c\|\mathbf{1}\|\mathbf{o}(i) \quad (1)$$

for some common prefix  $c$  and  $i \geq 0$ , i.e.,  $s_1$  is a key ending with a string of 1s, and  $s_2$  is the next higher key, thus ending with a string of 0s. First note that for any keys  $id_1$  and  $id_2$ ,  $XOR(id_1, id_2)$  starts with a string of 0s of the length of their common prefix. So, if  $id_1$  shares a longer common prefix with  $id_2$  than with a key  $id_3$ ,  $id_1$  is closer to



$id_2$  than to  $id_3$  with regard to the XOR distance. Now, assume that the NL contains keys  $k_1$  and  $k_2$  starting with  $c||0$  and  $c||1$ , respectively. As a consequence,  $x$  and  $y$ , the closest keys in NL to  $s_1$  and  $s_2$  with respect to the XOR distance, have to start with  $c||0$  or  $c||1$ , respectively. So,  $requestL(s_2)$  returns a list containing a key  $y = c||1||r_y = s_2 + r_y$  for some  $i$ -bit string  $r_y$ . Similarly,  $requestL(s_1)$  returns a list containing a key  $x = c||0||r_x = s_1 - \mathbf{1}(i) + r_x$  for some  $r_x$ . Therefore, it is shown that indeed  $x$  and  $y$  are such that  $NL \cap I(x, y) = \emptyset$ . By the definition of  $I(x, y)$ ,  $I(x, y) = I(x, s_2) \cup I(s_1, y)$ . The claim that  $NL \cap I(x, y) = \emptyset$  follows from showing that all  $z_x \in I(x, s_2)$  and  $z_y \in I(s_1, y)$  have a lower XOR distance to  $s_1$  or  $s_2$  than  $x$  or  $y$ , respectively, and hence cannot be contained in NL. Note that all  $z \in I(x, y)$  share the prefix  $c$ . Consider  $z_y = c||1||q \in I(s_1, y)$  for an  $i$ -bit string  $q$ , so that  $XOR(z, s_2) = q = z - s_2$ . As a consequence,  $XOR(z_y, s_2) < XOR(y, s_2)$  for all keys  $z_y \in I(s_1, y)$ , so that  $z_y \notin NL$  if  $y$  is the closest key to  $s_2$  in NL. Similarly, for any  $z_x = c||0||q \in I(x, s_2)$ ,  $r_x - q \leq \mathbf{1}(i)$ , so that  $XOR(z_x, s_1) = \mathbf{1}(i) - q$  and hence  $XOR(z_x, s_1) < XOR(x, s_1)$ . Hence,  $z_x \notin NL$  if  $x$  is the closest returned key to  $s_1$ . In summary, all keys in  $I(x, y)$  are not contained in NL, and thus a method to reliably identify sets of keys that are guaranteed not to be contained in NL is found. However, without further queries, it is not possible to say which keys in  $I(k_1, x)$  and  $I(y, k_2)$  are contained in NL.

**Example 5.1.1.** As an example consider the neighborlist  $NL_{ex} = \{00000, 00100, 01010, 01100, 10010, 11000\}$  and assume for simplicity that each query via  $requestL()$  only returns  $l = 1$  key. Assume it is already discovered that  $k_1 = 00000$  and  $k_2 = 01100$  with common prefix  $c = 0$ . The next step is to query with  $s_1 = 0||0||111 = 00111$  and  $s_2 = 01000$ .  $requestL(s_1)$  is guaranteed to return  $x = 00100$  and  $requestL(s_2)$  returns  $y = 01010$ . However, the reply does not tell if any keys in  $I(k_1, x) = \{00001, 00010, 00011\}$  or  $I(y, k_2) = \{01011\}$  are contained in  $NL_{ex}$ .

Algorithm 5 now subsequently identifies sets of keys which cannot be contained in NL, while at the same time finding new keys  $k_1$  and  $k_2$  that are used for determining the keys  $s_1$  and  $s_2$ . Initially, the list of discovered keys  $L$  is empty (Line 1). Then  $s_1 = \mathbf{o}(b)$  and  $s_2 = \mathbf{1}(b)$  are used as keys for the first two queries with the returned list  $requestL(s_1)$  and  $requestL(s_2)$  added to the set of discovered keys (Lines 2 - 7). In particular,  $requestL(s_1)$  has to contain the smallest key  $k_{first}$  and largest  $k_{last}$  in NL, i.e., the closest keys to  $\mathbf{o}(b)$  and  $\mathbf{1}(b)$ . Hence, the set  $I(k_{last}, k_{first})$  is the first detected set of keys that are not contained in NL. However,  $I(k_{first}, k_{last})$  potentially contains undiscovered keys, given that it is non-empty, i.e., the two keys are not equal or consecutive. So, the pair  $(k_{first}, k_{last})$  is the first element in  $R$  (Line 9), which is implemented as a queue. Hence,  $R$  contains pairs  $(k_1, k_2)$  whose common prefix defines the spoofed keys in future iterations. In each iteration of the while loop (Lines 10 - 26), such a pair  $(k_1, k_2)$  from the front of the queue is considered. The common prefix  $c$  of  $k_1$  and  $k_2$ , determines the two spoofed keys  $s_1$  and  $s_2$ , such that  $s_1 = c||0||\mathbf{1}(b - \text{length}(c) - 1)$ , which consists of the common prefix  $c$ , 0,

**Algorithm 5 : ZeusMilkер()**


---

```

// Initialization
1 L ← ∅ // Crawled keys
  // Get smallest key
2 M ← requestL(o(b))
3 L ← L ∪ M
4 kfirst ← getClosestKey(M, o(b))
  // Get largest key
5 M ← requestL(1(b))
6 L ← L ∪ M
7 klast ← getClosestKey(M, 1(b))
8 if kfirst ≠ klast && kfirst ≠ klast - 1 then
9   | R.push((kfirst, klast)) // Push undiscovered range
  // While not fully discovered
10 while not R = ∅ do
    // Get keys for spoofing
11   (k1, k2) ← R.pop()
12   c ← getCommonPrefix(k1, k2)
13   s1 ← c||0||1(b - length(c) - 1)
14   s2 ← c||1||o(b - length(c) - 1)
    // Execute queries and add new sets
15   if k1 s1 then
16     | M ← requestL(s1) // query with s1
17     | L ← L ∪ M
18     | x ← getClosestKey(M, s1)
19     | if x ≠ k1 then
20     | | R.push((k1, x))
21   if k2 s2 then
22     | M ← requestL(s2) // query with s2
23     | L ← L ∪ M
24     | y ← getClosestKey(M, s2)
25     | if y ≠ k2 then
26     | | R.push((y, k2))
27 return L

```

---

and a string of 1s achieving a total length of  $b$ , is the largest key closer to  $k_1$  than to  $k_2$  (in terms of the XOR-distance). Analogously,  $s_2 = c||1||o(b - \text{length}(c) - 1) = s_1 + 1$  is the smallest key closer to  $k_2$  than  $k_1$  (Lines 12-14). If  $s_1$  is not bigger than  $k_1$ ,  $I(k_1, s_1)$  is empty, hence it is not necessary to query with  $s_1$ . Analogously, if  $s_2$  is not smaller than  $k_2$ ,  $I(s_2, k_2)$  is empty. If  $s_1$  is bigger than  $k_1$ , the method call  $\text{requestL}(s_1)$  is executed, the returned list  $M$  added to  $L$ , and the key  $x$  is chosen as the closest key to  $s_1$  in  $M$  (Lines 16-18). Similar, if  $s_2$  is smaller than  $k_2$ ,  $y$  is chosen as the closest key to  $s_2$  in the set returned by  $\text{requestL}(s_2)$  (Lines 22-24). As discussed above, keys in  $I(x, y)$  are guaranteed to be not contained in  $NL$ , hence only the sets  $I(k_1, x)$  and  $I(y, k_2)$  can contain undiscovered keys if they are non-empty. Hence,

the pairs  $(k_1, x)$  and  $(y, k_2)$  are added to the end of  $R$  (Line 20 and 26, respectively).

**Example 5.1.2.** The exemplary neighborlist

$NL_{ex} = \{00000, 00100, 01010, 01100, 10010, 11000\}$  from Example 5.1.1 is used, which is sorted for simplicity and indexed by  $id_j = NL_{ex}[j]$ , for  $j = 0 \dots 5$ . The ring on the left of Figure 4 depicts how these keys map onto the whole key space. For simplicity, it is assumed that only  $l = 1$  keys are returned per query. However, for larger  $l = |NL_{ex}|$ , the same number of steps is required to *guarantee* that all keys in  $NL_{ex}$  are returned, though individual keys might be discovered much earlier. Initially, two queries are conducted, one with key 11111 (Line 5, Algorithm 5) and one with key 00000 (Line 2, Algorithm 5), which will return two entries from  $NL$ , namely  $k_{first} = id_0 = 00000$  (Line 4, Algorithm 5) and  $k_{last} = id_5 = 11000$  (Line 7, Algorithm 5), respectively. Hence, it can be deduced that there are no keys in  $I(id_5, id_0)$ . Then, as described in the following and as can be seen on the right of Figure 4, five iterations of the loop are executed as follows:

- 1) The pair of keys  $k_1 = id_0 = 00000$ , and  $k_2 = id_5 = 11000$  is retrieved from  $R$ . They do not share a common prefix, so spoofing with  $s_1 = 01111$  and  $s_2 = 10000$  discovers  $x = id_3 = 01100$  and  $y = id_4 = 10010$ . The pairs  $(id_0, id_3)$  and  $(id_4, id_5)$  are added to the set  $R$ . After this step, it can be guaranteed that  $NL_{ex}$  does not contain keys in  $I(id_3, id_4)$  since they would have been returned when spoofing IDs  $s_1$  or  $s_2$ .
- 2) The pair  $(id_4, id_5) = (10010, 11000)$  is retrieved, sharing common prefix 1. The spoofed keys are thus  $s_1 = 10111$  and  $s_2 = 11000$ . Because  $s_2$  is identical to  $id_5$  and hence there are no keys in  $I(s_2, id_5)$ , it is not necessary to spoof with  $s_2$ . Spoofing with  $s_1$  does not result in any closer key to  $s_1$  than  $id_4$ . No new pairs are added to  $R$ , and it is guaranteed that  $NL_{ex}$  does not contain keys in  $I(id_4, id_5)$ .
- 3) The pair  $(id_0, id_3) = (00000, 01100)$  is retrieved. Spoofing with  $s_1 = 00111$  and  $s_2 = 01000$  leads to the discovery of  $id_1 = 00100$  and  $id_2 = 01010$ . Therefore, the pairs  $(id_0, id_1)$  and  $(id_2, id_3)$  are added to  $R$ . As a consequence, it is known that  $NL_{ex}$  does not contain keys in  $I(id_1, id_2)$ .
- 4) The pair  $(id_2, id_3) = (01010, 01100)$  is retrieved, but spoofing with  $s_1 = 01011$  (spoofing  $s_2 = 01100$  not required) reveals that  $NL_{ex}$  does not contain keys in  $I(id_2, id_3)$ .
- 5) The pair  $(id_0, id_1) = (00000, 00100)$  is retrieved, but spoofing with  $s_1 = 00011$  (spoofing  $s_2 = 00100$  not required) reveals that  $NL_{ex}$  does not contain keys in  $I(id_0, id_1)$ .

The example indicates that in each step, Algorithm 5 discovers a pair of keys  $x$  and  $y$ , such that it is guaranteed that the  $NL$  does not contain keys in  $I(x, y)$ . In the following, an analysis is presented to show that the observation holds for all steps and utilize it to derive the complexity of Algorithm 5.

### 5.1.1.3 Correctness and Complexity Analysis of ZeusMilker

In the following, it is first shown that at least  $2n$  queries are needed to guarantee that an  $n$ -elemental  $NL$  is retrieved, regardless of the choice of spoofed keys. Second, Algorithm 5 is analyzed, and it is shown that

the algorithm indeed terminates in at most  $2n$  steps. The results presented here are mainly concerned with the *worst-case complexity*, defined as the maximal cost, i.e., the number of queries, required by any input, i.e., NLs, for the algorithm to terminate.

More precisely, the optimality of the proposed algorithm is shown in the following:

1. The complexity of the problem to obtain a provable complete NL is  $2n$ , i.e., there exist some NLs for which at least  $2n$  keys need to be spoofed regardless of the algorithm for choosing these keys.
2. The proposed algorithm needs at most  $2n$  steps to obtain a provable complete NL and is hence optimal with regard to the worst-case complexity.

As a result, the algorithm achieves optimal performance with regard to the worst-case complexity, which is not necessarily optimal for all inputs. However, it is later shown that while there are lists that require fewer steps, most NLs require at least  $2n$  spoofed keys, regardless of how these keys are chosen. Hence, the performance bound is indeed of practical relevance. To present a more general result,  $b$ -bit keys are considered rather than the concrete value of  $b = 160$  as used in GameOver Zeus.

**Proposition 5.1.3.** There exist NLs NL with  $n$  distinct keys, such that the number of queries needed to guarantee the complete retrieval of NL is, at least,  $2n$  for any choice of spoofed keys.

The proof is presented in Appendix A.1 for interested readers. The idea of the proof is to divide the set  $I(\text{id}_j, \text{id}_{((j+1) \bmod n)})$ ,  $\text{id}_j$  being the  $j$ -th smallest key in NL, into two sets  $F_{j-}$  and  $F_{j+}$ , such that keys in  $F_{j-}$  are closer to  $\text{id}_j$  and keys in  $F_{j+}$  closer to  $\text{id}_{j+1}$ . Then it is shown that each non-empty  $F_{j-}$  or  $F_{j+}$  requires, at least, one query, adding up to a total of  $2n$  queries.

It is shown that NLs exist such that  $2n$  spoofed keys are needed by any algorithm. However, as indicated by Example 5.1.2, it is possible that a NL can be discovered using less than  $2n$  queries. The existence of such examples raises the question if the worst-case complexity is a suitable measure, or if it is only relevant for few constructed examples.

In addition, an edge-case exists where all entries in NLs might be empty or shorter than the minimum reply size, i.e.,  $|\text{NL}| \geq 1$ . For these, it requires exactly one query to retrieve the full list.

Proposition 5.1.4 and the subsequent calculation of Eq. 2 for realistic botnet sizes  $m$  show that the vast majority of NLs require at least  $2n$  spoofed keys for crawling, such that the algorithm is not only optimal with regard to the worst-case complexity, but also for nearly all inputs.

**Proposition 5.1.4.** The probability that the guaranteed retrieval of an  $n$ -elemental NL in a GameOver Zeus botnet with  $m$  bots and  $b$ -bit keys requires less than  $2n$  spoofed keys is at most

$$\frac{m(m-1)}{2^{b+1}} (3 + 4(b-1)\ln 2). \quad (2)$$

The proof is provided in the Appendix A.3 for interested readers. The proof idea is to provide a bound on the probability that any  $F_{j-}$  or  $F_{j+}$ , as defined in the proof of Proposition 5.1.3, is empty. The result is obtained using basic probability theory.

In GameOver Zeus, typically,  $b = 160$ -bit keys are used, while the size of most of such botnets is usually not bigger than several ten thousand. For such parameters, Eq. 2 is negligible, being less than  $10^{33}$ , so that indeed the vast majority of NLs require  $2n$  crawl requests to reveal all entries. A more detailed discussion of Eq. 2 can be found in Appendix A.2.

Now, Algorithm 5 is analyzed and later shown that its complexity is indeed  $2n$ .

**Proposition 5.1.5.** Algorithm 5 guarantees the complete retrieval of any  $n$ -elemental NL and requires at most  $2n$  queries.

*Proof.* Denote the keys in NL by  $id_0, \dots, id_{n-1}$ , sorted in ascending order. In the first step of Algorithm 5 (Lines 2 - 7), two keys are spoofed to guarantee that the NL does not contain keys in  $I(id_{n-1}, id_0)$ . In each iteration of the loop, Algorithm 5 requires at most two queries with spoofed keys  $s_1$  and  $s_2 = s_1 + 1$  to guarantee that NL does not contain keys in  $I(id_j, id_{j+1})$  for some  $0 \leq j \leq n - 2$  (Lines 10-26),  $id_j$  being the closest key to  $s_1$  and  $id_{j+1}$  being the closest key to  $s_2$ . So,  $n - 1$  iterations of the loop are required to provably retrieve the NL, since each pair  $(id_j, id_{j+1})$  is only considered once. Hence, the two initial queries in addition to the maximum of  $2(n - 1)$  queries executed during the loop results in an upper bound of  $2n$  on the number of queries.  $\square$

It is now shown that Algorithm 5 achieves the lowest possible worst-case complexity. Note that the number of required steps can—in principle—be much lower in some rare cases: If the keys contained in the list are consecutive, i.e.,  $id_{j+1} = id_j + 1$ , the algorithm terminates in  $n$  steps. Furthermore, there are some NLs, for which the lowest possible number of steps cannot be achieved, e.g., for an NL consisting of only  $\mathfrak{o}(b)$  and  $\mathfrak{1}(b)$ , Algorithm 5 requires four steps, but a variant of Algorithm 5 that starts using  $0\|\mathfrak{1}(b - 1)$  and  $1\|\mathfrak{o}(b - 1)$  for spoofing terminates within two steps. The example clearly shows that for any choice of initially spoofed keys and hence for any algorithm, there is some NL for which the minimal number of steps is not achieved. However, Proposition 5.1.4 shows that the algorithm is optimal for the vast majority of cases.

Take note that although ZEUSMILKER is specifically designed for GameOver Zeus, the algorithm can be easily generalized and also applied to other P2P botnets like Storm, and/or Kademia-based P2P file-sharing networks. As long as the XOR-metric is utilized as a notion of distance between two values, e.g., node IDs, in a particular domain, ZEUSMILKER could be used to manipulate a mechanism to return entries to deduce additional information, e.g., all neighbors known to a particular node. However, it is required that the domain allows a user or client to pick values or keys arbitrarily. ZEUSMILKER

requires this feature in order to be able to manipulate the input parameters of the mechanism that calculates the XOR-distance.

### 5.1.2 *Less Invasive Crawling Algorithm (LICA)*

Crawling introduces a significant amount of network activity that is easily observable and may disclose a crawler to the botmasters. Although the simplest solution in crawling a botnet is to request NLs from all known nodes iteratively, this is neither stealthy nor efficient (cf. Section 3.4). For example, current BFS and DFS-based crawling algorithms need to crawl all possible nodes to provide a snapshot at any particular time. The unnecessary activity of frequently requesting NLs 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 [SRS05]. This argument is especially true if the reason for crawling is only to perform bot enumerations, i.e., identifying infected machines, and not to discover the full interconnectivity between bots. In the case of the former, it is desirable to minimize the necessary amount of interaction between crawler and the botnet.

The idea behind is only to 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. 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 (cf. Section 3.2) as follows:

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

However, this problem is proven to be  $\mathcal{NP}$ -hard, and all known approximation algorithms require a global view of the graph, e.g., the algorithm of Bar-Yehuda [BYE85].

For this reason, this particular work intends to *approximate* the minimum vertex cover during the crawl of a botnet. For that, this work tries to identify the *stable core* of a botnet and to crawl those bots first, which is inspired by the work of Stutzbach et al. [SRS05] on the *Gnutella* P2P network. Unstructured P2P networks, like other botnets presented in Section 4, maintain their overlay connectivity via a membership management mechanism. This mechanism ensures the robustness of the overlay by exchanging fresh information about active peers in the network. Consequently, the entry of a *stable* or an *important* peer is frequently shared by bots and will stay longer in the NL of many others. To exploit this observation, an iterative crawling algorithm named *Less Invasive Crawling Algorithm (LICA)* is proposed that employs a heuristic to plan the next crawling steps iteratively and to establish a *vertex cover* in the botnet that operates with such sparse graph information. This crawling algorithm attempts to optimize the coverage of subsequent crawling steps and thus decreases the required overall number of steps for crawling a botnet. Hence, this algorithm intends not to discover the full botnet interconnectivity,

but to extend the monitoring coverage to have the best, i.e., largest, snapshot of a botnet overlay along with the superpeers in it. It is assumed that bots in the botnet are all online at the same time; hence, diurnal patterns and churn effects are ignored in this work. Furthermore, it is also assumed that the NLs of other bots can be requested, so that either all neighbors are retrievable, e.g., GameOver Zeus (cf. Section 5.1.1), or subsets of the neighbors of a particular peer, e.g., Sality and ZeroAccess.

---

**Algorithm 6** : LICA(*seedpeer*, *R*, *w*, *t*)
 

---

```

// Initialization
1  $V_{\text{known}} \leftarrow \text{seedpeer}$ 
2  $c_{\text{crawl}} \leftarrow 0$ 
   // Maximum allowed requests (per node)
3 for  $i=0, i \leq r, i=i+1$  do
   // Utilize previous crawl
4    $V_{\text{crawl}} \leftarrow V_{\text{known}}$ 
5    $V_{\text{visited}} \leftarrow \emptyset$ 
6   do
   // Reset gain if necessary
7   if  $c_{\text{crawl}} \bmod w = 0$  then
8      $\text{gain} \leftarrow 0$ 
   // select the next node to crawl
9   Choose  $u \in$ 
      $\text{argmax}_{\forall v \in V_{\text{crawl}}} \sum_{\forall y \in V_{\text{known}}} |NL_y| - |NL_y - v|$ 
   // Crawl + get neighborlist of u
10   $NL_u \leftarrow \text{crawl}(u)$ 
   // Update list of visited nodes
11   $V_{\text{visited}} \leftarrow V_{\text{visited}} \cup \{u\}$ 
   // Update list of nodes to crawl
12   $V_{\text{crawl}} \leftarrow (V_{\text{crawl}} \cup NL_u) - V_{\text{visited}}$ 
13   $c_{\text{crawl}} \leftarrow c_{\text{crawl}} + 1$ 
   // Calculate gain
14   $\text{gain} \leftarrow \text{gain} + |NL_u| - |V_{\text{known}} \cap NL_u|$ 
   // Update visited nodes
15   $V_{\text{known}} \leftarrow V_{\text{known}} \cup NL_u$ 
16  while  $V_{\text{crawl}} \neq \emptyset \ \& \ (c_{\text{crawl}} \bmod w \neq 0 \ \parallel \ \text{gain} \div w \ t);$ 

```

---

LICA which is described in Algorithm 6, not only aims at crawling efficiency but is also configurable for an adaption to a specific environment or a specific botnet via parameters *seedpeer*, *r*, *w*, and *t*.

The *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 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.,  $\geq 0$ , 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  $t, \geq 0$ .

LICA utilizes the initial *seedpeer* for bootstrapping itself into the botnet overlay. Then, starting with the *seedpeer*, the algorithm obtains the NL  $NL_u$  from  $u$  (line 10) and further extends its knowledge by iteratively requesting NLs from the discovered peers. For each request sent by LICA, the counter  $c_{crawl}$  is incremented by one.

Upon receiving an NL from  $u$ , it is immediately added to  $V_{visited}$  (line 11) and the undiscovered peers in the received entries are added to  $V_{crawl}$  (line 15) as potential candidates for the crawl.

Line 9 in the algorithm selects the next candidate for the crawl. The algorithm goes through all received NLs of peers in  $V_{known}$  and ranks all remaining peers in  $V_{crawl}$  based on their popularity, i.e., their in-degree. Since the exact in-degree is not known, an approximation is obtained by counting the number of the occurrences a candidate or peer is seen among the NLs of other 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 one among them.

At every window interval, i.e., after  $w$  requests, the algorithm checks the accumulated gain (line 14) within the past window and terminates the current crawl iteration if the ratio of the observed gain drops below threshold  $t$  (line 16). Depending on the value of  $t$ , 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, the number of maximum allowed iterations is exceeded, or gain is below  $t$ .

## 5.2 ADVANCED ANTI-CRAWLING COUNTERMEASURES

ZEUSMILKER and LICA as presented in Section 5.1 circumvents existing anti-crawling mechanisms. However, as usual in the race between a botmaster and researchers, it can be expected that botmasters will introduce newer countermeasures to send the researchers back at trying to circumvent them yet again. Therefore in this section, by assuming the role of a botmaster, some advanced anti-crawling countermeasures that can be expected in the near future are proposed. Although mechanisms presented here can be utilized to fortify or improve future botnets, the introduction of these advanced countermeasures is necessary to allow researchers to anticipate and be prepared before such mechanisms are seen in the wild.

In the following, two advanced anti-crawling countermeasures are proposed to undermine some of the non-functional requirements proposed in Section 3.1.2. First, improved NL restriction mechanisms over the GameOver Zeus' mechanism (cf. Section 4.1.2.1) as a crawling prevention mechanism are presented. Second, a lightweight crawler detection mechanism that is easily deployable in existing P2P botnets is presented.

### 5.2.1 *Enhancing GameOver Zeus' NL Restriction Mechanism*

The NL restriction mechanism of GameOver Zeus had one major weakness: the requester can manipulate the returned entries. The existing mechanism accepts any input, i.e., key, from the sender's message (cf. Section 4.1.2.1) without validating the input or key. This *feature* allowed ZEUSMILKER to manipulate the mechanism by deterministically spoofing keys to retrieve the bot's entire NL. Hence, it is important that any future mechanism that aims to *prevent* crawling activities (cf. Section 3.4.3.1) also ensures that the requester is not able to manipulate the selection of returned neighbors. As such, a crawler can be prevented from obtaining an accurate representation of the overlay topology (Non-Functional Requirement #4).

Take note that one important aspect of an MM is to ensure a robust botnet overlay. Therefore, all NL restriction mechanisms need to ensure that the botnet's overlay connectivity is not adversely affected by the restriction techniques. In the following, two countermeasures to improve the existing *GameOver Zeus* NL restriction mechanism are presented along with another countermeasure that simply returns random nodes from the NL. The evaluation results of the countermeasures along with a discussion will be presented in Section 5.3.1.5.

#### 5.2.1.1 *Random Node Return*

In Sality [Fal11], bots return exactly one entry that is randomly chosen from their respective NLs to the requesting bot (cf. Section 4.2.2). Hence, the requesting bot has no influence on the returned entries at all. However, all entries have an equal likelihood to be returned to any requesting bots. This approach from Sality is considered as one of the potential countermeasures in this work.

#### 5.2.1.2 *Bit-XOR+*

This countermeasure adds additional randomness at the side of the recipient of an NL request. The recipient, i.e., bot, generates a random key uniformly for each IP address it receives a request from and stores it. This key is then *XOR*-ed with the key of the requesting node, and the resulting key is then used as an input for Algorithm 1 to return the neighbor entries. Hence, the set of keys that is returned is now biased towards the new *XOR*-ed key, and an attacker loses its ability to strategically spoof keys. By including a randomly generated key into the selection process, each entry in the NL has the equal likelihood to be returned, such that *Bit-XOR+* is expected not to affect the connectivity between the bots negatively.

#### 5.2.1.3 *Bit-AND*

*Bit-AND* is a variation of the *Bit-XOR+* countermeasure that executes a bit-wise AND operation between the stored key and the requesters key before using the resulting key to return NL entries. However, due

to the nature of the AND operation whereupon each bit of the resulting key has a tendency to be 0 with a probability  $3/4$ , the set of returned keys for *Bit-AND* is likely to be biased towards keys starting with 0s. On the one hand, such a bias can considerably decrease the performance of all crawlers because the returned sets are expected to have a larger overlap in contrast to uniformly selected sets. On the other hand, keys starting with 1s are expected to be present in fewer NLs, potentially damaging the connectivity and thus the resilience of the botnet. Therefore, while *Bit-AND* is expected to achieve the best performance out of the three countermeasures, its disadvantages likely outweigh its benefits.

### 5.2.2 *BoobyTrap: Detecting Persistent Crawlers*

This section focuses on *detecting* crawlers that may still be able to tolerate the proposed NL restriction mechanisms in Section 5.2.1 from the perspective of a botmaster. From the observations of crawlers deployed in existing P2P botnets [Kar+16a], most of the crawlers exhibit similar characteristics, i.e., greedy and aggressive in contacting all bots, compared to bots. Moreover, since researchers are required to manually re-implement some of the botnet's protocol to crawl the botnet, it is also observed that many crawlers have incomplete or simplified (re)implementations of the botnet protocols. Such characteristics of a crawler can be leveraged to detect them within the botnets.

For that, Section 5.2.2.1 introduces a set of lightweight crawler detection techniques called *BoobyTrap (BT)*, which identify crawlers exhibiting such characteristics compared to bots. Section 5.2.2.2 and 5.2.2.3 presents the adaptation of BT to Salty and ZeroAccess. Although similar adaptations of BT are possible for GameOver Zeus, they were not focused in this work as it was not possible to evaluate the mechanisms on a real world scenario compared to the other two botnets, i.e., GameOver Zeus is being sinkholed since 2014.

#### 5.2.2.1 *BoobyTrap (BT)*

A BT node can be a regular bot or a sensor node (cf. Section 3.3.3) that is enriched with detection mechanisms or '*traps*' to identify autonomously misbehaving nodes that are contacting it. As a proof-of-concept, a sensor is utilized through the remainder part of this work. All communication with the BT node is logged in a relational database for future reference along with additional metadata: 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 (described later). Take note that in this work, the deployed BT nodes only return non-bot entries, i.e., other sensor nodes, to avoid participating in the regular botnet maintenance activities due to legal constraints. Therefore, the

BT-enhanced sensor nodes deployed in this work do not participate in any activities that may, directly or indirectly, aid the botnet in any manner.

The BT detection mechanism leverages upon the following assumptions of crawlers that are derived from the observations in real-world botnets.

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.

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

**DEFIANCE** Bots that defy the botnet-specified *MM* protocols can be classified as crawlers. An example of such defiance 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 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 *GameOver Zeus* (cf. Section 4.1.2.2). 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 node, this behavior can be classified as a crawler.

**ABUSE** In P2P botnets, bots may request the NLS of their neighbors to add additional neighbors to their NL. The ability to request new neighbors 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 (cf. Section 3.3.2). 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 (cf. Section 3.4.3.1). 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 (cf. Section 3.4.1). Therefore, crawlers typically crawl bots with higher frequencies [Kar+15]. In contrast, bots usually probe their neighbors only once per *MM* cycle, i.e., between 1 sec and 40 minutes, depending on the botnet. 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 *GameOver Zeus* botnet (cf. Section 4.1.3).

**AVOIDANCE** The MM of a botnet specifies the sequence of message exchange as well as the structure of the messages. Hence, nodes that do not respond to messages (periodically) or return non-consistent responses, e.g., empty or invalid, might be crawlers too. This premise is reasonable as the designs of most monitoring nodes are often minimalistic and does not aim at aiding the botnet in its day-to-day operations such as command dissemination, attack execution, or management of the overlay. This is usually enforced by the need to adhere to research ethics or cyber laws in many countries. 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 would often generate a verifiable reply, bots that are refusing to respond (or ignore the requests) can be classified as crawlers.

### 5.2.2.2 *Adaptation of BoobyTrap for Sality*

In the following, three crawler detection mechanisms or *traps* are adapted for Sality, adhering to two out of the three misbehavior classes presented in Section 5.2.2. Each trap's name for Sality is prefixed with an *S*, and the abbreviation of the respective detection class. There are two traps for Sality in the class of *Defiance*, i.e., *SD1-IgnoreTrap* and *SD2-BaitTrap*, and one trap from the class of *Abuse*, i.e., *SAB-BurstTrap*. Take note that a trap can also be setup from the class of *Avoidance* based on the appended *URLPack*, i.e., newer update, within the *Hello* message exchange process. However, such a trap for Sality is omitted in this work as it may induce DDoS on the BT node itself due to continuously requesting *URLPack* responses that will generate heavy network traffic towards the node itself [Ros14].

**SD1-IGNORETRAP** The MM protocol of Sality dictates that a bot utilizes a *Hello* message to probe the responsiveness of its neighbors (cf. Section 4.2.2). If the neighbors are responsive, and the probing bot requires additional neighbors, only then, it sends an additional  $NL_{Req}$  message. As such, an  $NL_{Req}$  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.

**SD2-BAITTRAP** The MM protocol of Sality also ensures that an IP address can only be present once in a bot's NL (Line 3, Algorithm 2). When a bot discovers a potential neighbor with the same IP but different port (Line 5, Algorithm 2), it prefers an existing and responsive entry, i.e., IP address. This trap exploits this behavior by deliberately responding to all received  $NL_{Req}$  with an entry that points back to the

BT node's secondary port. 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 would 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.

**SAB-BURSTTRAP** Bots in Sality probe the responsiveness of their neighbors once every 40 minutes (cf. Section 4.2.2). In addition, bots can (optionally) request neighbors of their neighbor by sending an  $NL_{Req}$ . 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 requests within a short interval, i.e., 40 minutes, this behavior triggers the detection mechanism.

### 5.2.2.3 Adaptation of BoobyTrap for ZeroAccess

In the following, three crawler detection mechanisms or *traps* are adapted for ZeroAccess from the different misbehavior classes presented in Section 5.2.2. Each trap's name is prefixed by an abbreviation of the botnet's name and the respective detection class.

**ZD-NONCOMPLIANCE**TRAP The MM protocol of ZeroAccess as described in Algorithm 3 allows bots to identify if a *getL* message was received. This is done by checking the flag value of the received request message (Line 4), i.e.,  $flag == 0$ . In reply, legitimate bots *always* send a *getL+* message that has its flag set to 1. However, it would still be 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 *retL* or *retL+* messages that have the *flag* values *copied* from the received request messages (Line 2). In addition, due to the possibility of UDP hole punching in ZeroAccess [Ros+13], 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. Any crawlers that are non-compliant to the MM protocol by not copying the exact flag that has been received in the reply will be detected.

**ZAB-BURSTTRAP** The MM mechanism of ZeroAccess indicates that a bot would only contact a particular neighbor at most three times within a duration of 256 seconds (cf. Section 4.3.2). This observation is exploited in this trap that triggers when any bot attempts to contact the BT node aggressively in quick successions, i.e., more than three requests within 256 seconds. The maximum number of *getL* messages that can be expected from a bot with a full NL, i.e., 256 neighbors, is three messages within a maximum interval of 256 seconds (probed

once from each of the three NLs). Similar to *SAB-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.

**ZAV-IGNORETRAP** This trap works in combination with the *ZD-NonComplianceTrap*. Crawlers that received the 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.

### 5.3 EVALUATION

This section presents the evaluation results and analysis of the mechanisms proposed in Section 5.1 and 5.2 in three parts. In the first part (Section 5.3.1), a thorough analysis of ZEUSMILKER is presented in the context of circumventing the NL restriction mechanism of GameOver Zeus as described in Section 5.1.1. In addition, the evaluation of the enhanced restriction mechanisms as introduced in Section 5.2.1 is also presented.

The second part of this section (Section 5.3.2) presents the evaluation results of the Less Invasive Crawling Algorithm (LICA) as described in Section 5.1.2. The final part (Section 5.3.3) provides an analysis of the ability to detect crawlers in existing botnets through the mechanisms introduced in Section 5.2.2.

#### 5.3.1 Evaluation of ZeusMilker

The evaluation of ZEUSMILKER is outlined below and has also been published in [Kar+15]. First, the dataset utilized for evaluating ZEUSMILKER is discussed in Section 5.3.1.1. Then, Section 5.3.1.2 elaborates on the setup for the experiments and Section 5.3.1.3 introduces the metrics used in the evaluations. After that, the investigated research questions are listed along with the expectations of the outcome in Section 5.3.1.4. Finally, the results of the experiments are presented in Section 5.3.1.5.

##### 5.3.1.1 Dataset

A real-world GameOver Zeus dataset is used in the evaluation that consists of crawled information collected for a duration of approximately five hours from the botnet on 25th April 2013. The sanitized dataset contains information of 900 bots that have between 10 to 70 entries in their respective NL. The median of the dataset is 34 entries with a standard deviation of 18.37.

### 5.3.1.2 Experimental Setup

The membership management protocols of GameOver Zeus was implemented in *OMNeT++*<sup>1</sup> by making use of *OverSim*<sup>2</sup>[BHK07] as the simulation framework. OMNeT++ is a discrete event simulator that allows simulation of networks. Meanwhile, OverSim adds the required functionality for overlays that is leveraged in implementing the membership management mechanism of the botnets. For ZEUSMILKER, the implementation includes the NL restriction mechanism as described in Algorithm 1. As for the generation of random keys within *OverSim*, the *OverlayKey* class is utilized to generate keys following a uniform distribution to investigate the effects of key distribution within the NLs.

For each iteration of the experiment, data for each bot in the simulation is uniformly selected at random from the dataset described in Section 5.3.1.1 depending on the investigated NL size. The bots are then assigned the selected key and have their NL filled with the associated NL entries. Since the order of entries in a bot's NL is non-deterministic, a random permutation is applied to the contained entries before storing them as the NL.

The following monitoring approaches were also implemented in the simulation framework, to evaluate not only the effectiveness of the NL restriction mechanism in GameOver Zeus but also the efficiency of ZEUSMILKER in comparison to other approaches:

- **ZeusMilker** is the proposed approach for strategically spoofing keys to *milk* all entries from a bot's NL implemented as per Algorithm 5.
- **Random** is the only other known monitoring approach than ZEUSMILKER used for monitoring GameOver Zeus [Ros+13] that is known within the research community. The spoofed keys are 160-bit in length and generated uniformly at random for each request.
- **BinaryHalving** spoofs keys by halving the ID space in the manner of a binary search algorithm. For each iteration of the algorithm, two keys are derived between two previously crawled keys. This halving process is repeated until the maximum number of permitted requests is reached. For that, *BinaryHalving* initially spoofs with  $\mathbf{o}(b)$  and  $\mathbf{1}(b)$ , and adds the pair  $(\mathbf{o}(b), \mathbf{1}(b))$  to a FIFO queue  $Q$ . Then it executes the following statement  $T$  times:
  1. Remove the head  $(K_1, K_2)$  of  $Q$  and determine the keys  $h_1 = \lfloor \frac{K_1 + K_2}{2} \rfloor$  and  $h_2 = h_1 + 1$ ,
  2. Crawl using spoofed keys  $h_1$  and  $h_2$ , and
  3. Add  $(K_1, h_1)$  and  $(h_2, K_2)$  to  $Q$ .

<sup>1</sup> <http://www.omnetpp.org>

<sup>2</sup> <http://www.oversim.org>

Please note that, to the best of knowledge, this approach has not been utilized in any real botnet monitoring system. It is introduced in this work to investigate and evaluate the case where keys are distributed evenly throughout an ID space.

For each experiment, the results were averaged over 50 independent trials with confidence intervals of 95%. Furthermore, for each iteration of the experiments, a unique seed value has been used to initialize the simulation models and to choose a random node from the dataset. In all of the experiments, the maximum number of requests is limited to  $2n$  requests, where  $n = |\text{NL}|$ , in agreement with the worst-case complexity for retrieving a complete NL (see Proposition 5.1.3).

#### 5.3.1.3 Evaluation Metric

The success of anti-crawling countermeasures and the performance of ZEUSMILKER are measured by the *discovery ratio*. It is defined as the unique fraction of an NL that is retrieved during crawling. Hence, the discovery ratio is an assessment of both the efficiency of the crawling algorithm as well as the effectiveness of the botnet's countermeasures, allowing the comparison of different crawling and anti-monitoring strategies.

#### 5.3.1.4 Research Questions and Expectations

One of the countermeasures to hinder successful botnet monitoring is to restrict the number of entries that are returned after receiving NL request (cf. Section 3.4.3.1). Hence, the following research question needs to be answered in the evaluation:

- *What is the influence on different NL sizes  $n$  and NL return sizes  $l$ ?*

In this investigation to answer the research question, ZEUSMILKER is expected to obtain all entries successfully with at most  $2n$  requests in every scenario as shown in Proposition 5.1.5. Meanwhile, *Random* and *BinaryHalving* are expected to miss some entries. The *Random* crawling strategy retrieves a randomized set of entries and has a high probability of missing one or more keys. *BinaryHalving*, in contrast, divides the search space strategically, but does not make use of knowledge gained in previous steps and as such may continue to query regions with few or no keys intensively. Furthermore, the performance of all algorithms is expected to increase with increasing  $l$ , because more keys are discovered in each step. The increase should be particularly strong for *Random* as the probability to be successful when spoofing randomly is highly dependent on the number of trials.

The distribution of keys within a real world GameOver Zeus bot's NL is observed biased to the key of the bot itself [Ros+13]. However, due to the botnet's NL return mechanism (cf. Section 4.1.2.1), different key distributions may influence the number of requests needed to be able to retrieve the entire NL. Examples of other distributions that could occur in a bot's NL are:

1. **Random Distribution:** A node's NL contains only randomly generated keys.
2. **Consecutive Entries:** A node's NL contains only consecutive keys, e.g.,  $k_{j+1} = k_j + 1 \pmod{2^{160}}$ .

Therefore, the following research question needs to be answered in the evaluation:

- *How do the different distributions of keys within a bot's NL influence the performance of the spoofing algorithms?*

ZEUSMILKER is expected to retrieve all entries within a bot's NL independent of the chosen distribution. However, in the case of *Consecutive Entries*, it is expected to retrieve all unique entries in only  $n$  requests instead of  $2n$ , as it is not necessary to check for additional keys between two neighboring keys. In contrast, *Random* and *BinaryHalving* are expected to require more crawling requests in this setting. Especially, *BinaryHalving* is expected to perform worst, as it spoofs many keys that yield no new knowledge in the *Consecutive Entries* setting. However, in the *Random Distribution* setting, both are expected to be closer, but still inferior to the crawling performance of ZEUSMILKER, as a result of the uniform key distribution.

The existing NL restriction mechanism of GameOver Zeus is exploitable as the requesting bot can manipulate the choice of returned entries based on the supplied key. Newer countermeasures presented in Section 5.2.1 attempt to either deny the possibility of manipulating the entries at all or allow restricted degree of manipulation on the returned entries. As such, the following research question needs to be answered in the evaluation:

- *How does the crawling algorithms perform in the presence of countermeasures like Random Node Return, Bit-XOR+, and Bit-AND?*

For *Random Node Return*, both *Random* and *BinaryHalving* algorithms are expected to be comparable in performance since the returned entries are not influenced by their key spoofing algorithm. However, *ZeusMilker* is expected to perform poorly if the choice of keys returned by the bot mislead the algorithm into believing that there are no more keys left undiscovered.

For *Bit-XOR+*, *Random* is expected to perform best compared to the other two algorithms as it produces higher entropy of keys used in selecting neighbors to be returned. Finally, for *Bit-AND*, all algorithms are expected to perform poorly since the bits of the returned entries are highly biased to 0 with a probability of  $3/4$ .

### 5.3.1.5 Results

In the following, the evaluation findings are summarized on the impact of parameters  $l$ ,  $n$ , the assumed key distribution, and effectiveness against advanced countermeasures on the three different crawling algorithms.

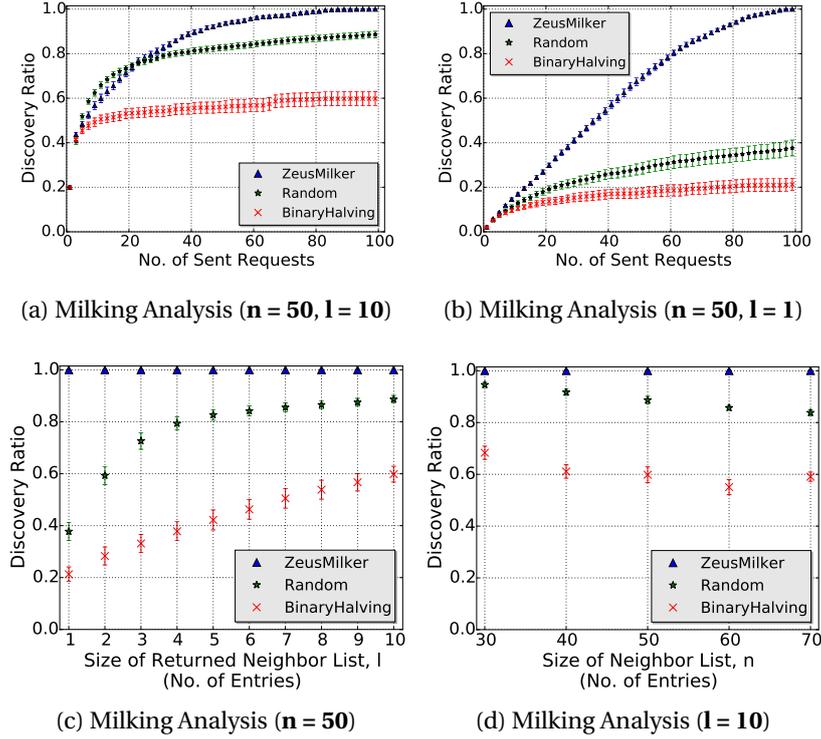
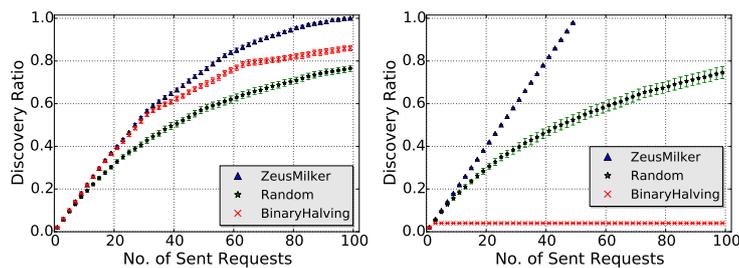


Figure 5: Performance analysis of ZEUSMILKER, *Random*, and *BinaryHalving* on GameOver Zeus for various NL sizes  $n$  and returned NL sizes  $l$

**IMPACT OF THE SIZE OF THE RETURNED NEIGHBORLISTS  $l$**  First, the impact of the size of the returned NL  $l$  is discussed. Figure 5a summarizes the discovery ratio for a default parameter setting of a GameOver Zeus botnet with  $n = 50$  and  $l = 10$  in dependence on the number of requests for all three crawling strategies. As can be observed, ZEUSMILKER can successfully retrieve all entries in a bot's NL within 100 requests as guaranteed by Proposition 5.1.5. At the same time, *Random* discovers only 92% and *BinaryHalving* even only 53% of all entries in a bot's NL. Thus, the results confirm the expectation that *BinaryHalving* is not suitable for such biased NLs. *BinaryHalving* performs poorly because of retrieving many duplicate entries as a result of spoofing keys within a range of the key space that provides no additional new information. For all algorithms, the number of initially retrieved entries increases fast with only a few queries. Later on, when only a few keys are left undiscovered, the slope of the performance curve decreases. Note that during the first few queries, the *Random* crawling algorithm even manages to discover more number of unique keys than ZEUSMILKER. A potential reason for the initially weaker performance of ZEUSMILKER is the choice of the two spoofed keys  $s_1, s_2$  (see Eq. 1), which are potentially very close and hence can result in returned sets with a high overlap. However, ZEUSMILKER is clearly superior to *Random* and *BinaryHalving* in discovering larger portions or even the complete NL.

Figure 5b shows the discovery ratio in dependence on the number of crawling requests for  $n = 50$  and  $l = 1$ . ZEUSMILKER still retrieves all entries within the predicted 100 requests, though at a lower speed than for  $l = 10$ . As only one entry per request can be obtained, the number of retrieved keys initially increases linearly and then converges slowly to a discovery ratio of 1. The decrease in performance is more apparent for *Random* and *BinaryHalving*: The discovery ratio for both approaches decreases drastically compared to  $l = 10$ , to 19% for *BinaryHalving* and 37% for *Random* at 100 requests. A more detailed analysis of the impact of  $l$  is given in Figure 5c showing the discovery ratio in dependence on  $l$  for  $n = 50$ . ZEUSMILKER successfully obtains all neighbor entries within  $2n = 100$  queries independent of  $l$ , whereas the discovery ratios after  $2n$  queries of the other two strategies are significantly affected by  $l$ . Since both algorithms are unable to strategically spoof keys, the fraction of retrieved keys drastically decreases when the number of returned keys  $l$  is reduced. Hence, the results of this analysis match the initial expectation that smaller values of  $l$  restrict the amount of new knowledge the crawling algorithms could obtain. However, since ZEUSMILKER can strategically spoof keys to discover all entries in a NL, its ability to retrieve the complete list remains unaffected by different values of  $l$ .

**IMPACT OF THE SIZE OF THE NEIGHBORLISTS  $n$**  Next, the impact of the size of the NL  $n$  on the crawling performance is analyzed. Figure 5d shows the discovery ratio of the different algorithms in dependence on  $n$  for  $l = 10$ . Independent of  $n$ , ZEUSMILKER successfully discovers all nodes in an NL. In contrast, the performance of *Random* slowly decreases with increasing  $n$ , because it is harder to discover large sets simply by random trials than on smaller sets. The slight decrease in performance of *BinaryHalving* is not significant.



(a) GameOver Zeus Milking Analysis for **Random Distribution** (b) GameOver Zeus Milking Analysis for **Consecutive Entries**

Figure 6: Performance analysis of ZEUSMILKER, *Random*, and *BinaryHalving* for different key distributions in NLs ( $n = 50$ ,  $l = 1$ )

**INFLUENCE OF DIFFERENT KEY DISTRIBUTIONS** Apart from  $n$  and  $l$ , different key distributions in NLs may also influence the performance of crawling algorithms. Figure 6a shows the discovery ratio in dependence on the number of requests for all three crawling strategies in the *Random Distribution* setting. As expected, ZEUSMILKER

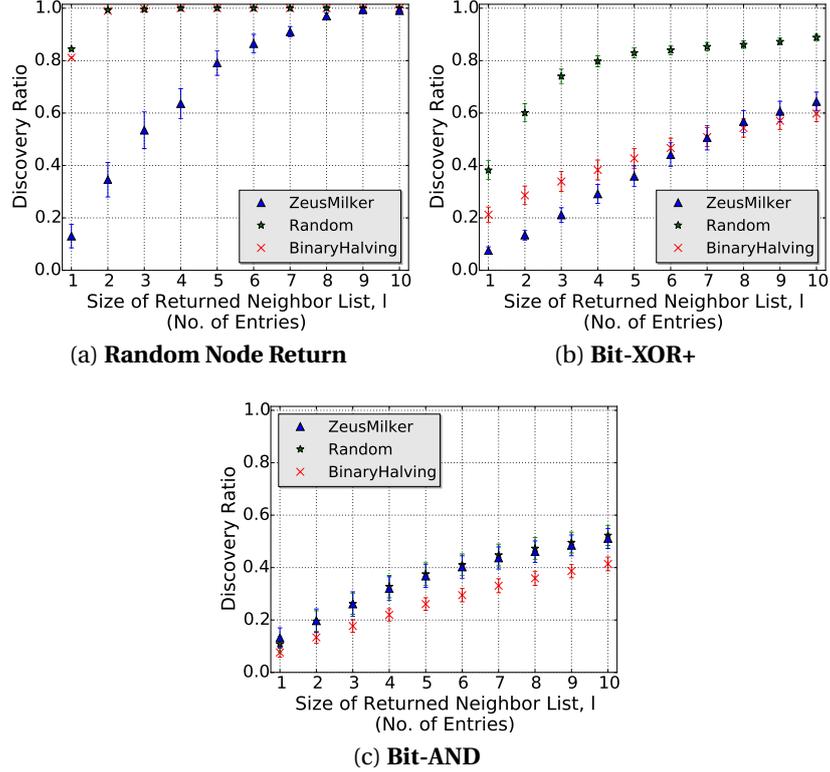


Figure 7: Performance analysis of *ZEUSMILKER*, *Random*, and *BinaryHalving* on the presence of different advanced countermeasures ( $n = 50$ )

can obtain all entries with at most  $2n = 100$  requests whereas *Random* and *BinaryHalving* only discover about 80% and 90% of all NL entries, respectively. Both strategies perform considerably better than the results for the real-world data set, increasing their discovery ratio by more than a factor of 2 and 4, respectively. This improved performance is contributed by the well-distributed keys, resulting in fewer duplicates during successive crawling attempts. As expected, *BinaryHalving* also performs much better than *Random* when the uniform key distribution assumed by *BinaryHalving* is indeed given.

The inability of *BinaryHalving* to deal with non-uniform key distributions becomes evident when considering *Consecutive Entries*. Most of the time, *BinaryHalving* discovers only two keys, i.e., discovery ratio is about 4%. A potential reason is the repeated spoofing of keys at distances away from all keys in the NL. As a result, the same two keys are returned repetitively. *ZEUSMILKER*, in contrast, can successfully discover all entries with only  $n$  requests instead of  $2n$  because the sets  $I(K_j, K_{j+1})$  are empty so that no additional  $n$  keys need to be spoofed to verify that  $I(K_j, K_{j+1}) \cap NL = \emptyset$ . In contrast, the performance of the *Random* crawling is similar to its performance when considering randomly distributed keys.

**ENHANCED RESTRICTION MECHANISMS** The evaluation results of the proposed anti-crawling countermeasures in Section 5.2.1 as depicted in Figure 7 is presented in the following. The *Random Node*

*Return* analysis in Figure 7a indicates the inefficiency of this countermeasure in restricting the information gained by a crawler. Both *Random* and *BinaryHalving* were able to retrieve more than 80% of a bot's NL in all parameter settings after 100 requests. However, ZEUSMILKER performed poorly as expected due to the inherent incorrect assumptions made on the returned keys that led the algorithm to falsely assume there are no more keys left undiscovered.

The results for the *Bit-XOR+* countermeasure indicate that *Random* performs best with an average of about 80% of nodes discovered for  $l \geq 4$ , as displayed in Figure 7b. Hence, the performance of *Random* is largely not influenced by bits flipping, as can be seen from comparing Figure 7b and Figure 5c, showing the performance of the crawling for the unaltered GameOver Zeus. Although *BinaryHalving* initially performs better than ZEUSMILKER for  $l \leq 7$ , its performance degrades for  $l > 7$ , as a result of spoofing keys that yield more duplicate entries. However, ZEUSMILKER's strategy of deriving keys based on previous knowledge provides more randomness, i.e., a variety of key prefixes, in the spoofed keys, hence obtains a slight improvement than *BinaryHalving* towards the end.

*Bit-AND*, as displayed in Figure 7c, presents a better restriction mechanism than *Random Node Return* and *Bit-XOR+* as the discovery ratio of all crawling algorithms is kept below 50% for  $l \leq 10$ . The discovery ratio increases with the size of the returned NLs  $l$  in a close to linearly manner. Although the poor performance of all strategies in terms of discovery ratio indicates the effectiveness of this countermeasure, the bias resulting from this strategy may negatively affect the robustness of the resulting overlay.

### 5.3.2 Evaluation of The Less Invasive Crawling Algorithm (LICA)

The evaluation of LICA is outlined below and the results have been published in [Kar+14]. First, the dataset utilized for evaluating LICA is discussed in Section 5.3.2.1. Then, Section 5.3.2.2 elaborates the setup for the experiments and Section 5.3.2.3 introduces the evaluation metric. After that, the investigated research questions are listed along with the expectations of the outcome in Section 5.3.2.4. Finally, the results of the experiments are presented in Section 5.3.2.5.

#### 5.3.2.1 Dataset

Two different real-world unstructured P2P network datasets in the form of directed graphs, i.e., GameOver Zeus and *Gnutella*, were used to evaluate the performance of crawling algorithms.

The GameOver Zeus dataset used in this evaluation consists of crawling information collected in approximately five hours from the GameOver Zeus botnet on 25th April 2013. It has been obtained from previous work in analyzing GameOver Zeus [Ros+13]. From the initial 1,061,402 edge entries in the database, 667,704 edge entries were removed that consist of biases that were made known by the authors: sinkholed nodes (identified by an out-degree  $> 10$ ), sensor nodes (identified by

an in-degree 500), and duplicated edges. Take note that, since the crawl data consists of multiple continuous crawls over a longer period, some bots may have reported a lot of neighbors than usual, i.e., 50. This is mainly due to churn dynamics within the botnet (cf. Section 3.4.1).

The second dataset is the crawl data of the unstructured P2P file-sharing network *Gnutella*, in August 2002 that was obtained from the SNAP repository<sup>3</sup>. This dataset was used as it is for the evaluation, i.e., without any sanitization. A summary of the dataset properties is provided in Table 7. Please note that properties listed for GameOver Zeus in the table are the post-sanitize properties of the dataset.

Dataset Name	GameOver Zeus	Gnutella
Nodes	82,471	62,586
Nodes (out-degree 10)	10,794	16,387
Avg. NL Size	4.6	2.4
Highest NL 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 7: Graph properties of the datasets.

### 5.3.2.2 Experimental Setup

The analysis was conducted using *Python* and the *NetworkX* module [HSC08], with all crawling algorithms discussed in Section 5.1.2, i.e., Less Invasive Crawling Algorithm (LICA), Breadth-First Search (BFS), and Depth-First Search (DFS), implemented as Python scripts. To address the issue of some nodes having NL size of more than 50, all entries of a bot are shuffled and split into a sequence of chunks, i.e., 50 entries in each chunk. This value is chosen to resemble closely the GameOver Zeus' implementation of the NL size.

For every received NL request from the 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 the experiments, a full crawl ends when there are no more new peers to crawl. In addition, LICA also ends its crawl when the maximum allowed iterations have exceeded (cf. Section 5.1.2).

Fifty (50) independent experiments were executed on each of the experiments and the final results were averaged over them. For each iteration of the experiment, the simulation uniformly chooses a common seed peer to begin crawling for all the algorithms. Furthermore, for the clarity of the resulting plots, all algorithms terminate their crawling as soon as 95% (indicated by the horizontal dashed lines) of nodes

<sup>3</sup> SNAP: <http://snap.stanford.edu/data/>

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.

### 5.3.2.3 Evaluation Metric

To evaluate the performance of a crawling algorithm, a *ratio of discovered peers* is used as an evaluation metric. This metric represents the ratio of *unique* peers discovered dependent on the number of sent request messages.

Meanwhile, to evaluate the efficiency of a crawling algorithm, the local-ratio approximation of *minimum vertex cover* presented by Bar-Yehuda et al. [BYE85] is utilized. This approximation provides a value of the number of minimum nodes to be crawled to obtain a snapshot of the botnet graph, i.e., a vertex cover. 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, it has been modified to be applied to directed graphs. In the remainder of this evaluation, this modified algorithm is referred as *Approximative Minimum Vertex Cover (AMVC)*.

### 5.3.2.4 Research Questions and Expectations

LICA is a flexible crawling algorithm that requires a combination of parameters to perform well. Depending on the choice of the parameters, an ongoing LICA-based crawl may terminate quickly or much later. However, the process of selecting the best combination of parameters  $w$ ,  $r$ , and  $t$  is not very intuitive. For that, the following research question needs to be answered:

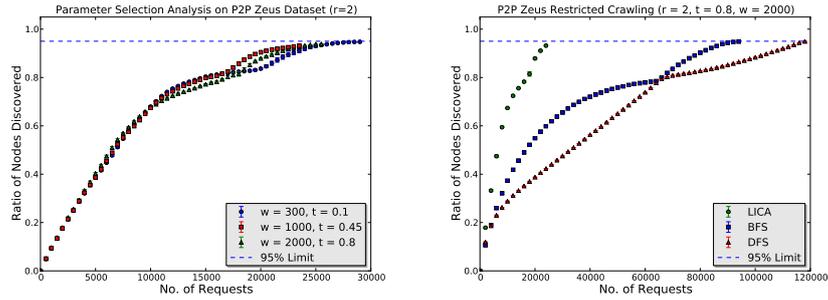
- *What is the best combination of parameters for LICA in GameOver Zeus and the Gnutella dataset?*

DFS and BFS-based crawling algorithms attempt to discover all bots in a greedy manner. However, not much has been discussed on the performance of these algorithms in a real world botnet scenarios. Therefore, the following research question needs to be answered:

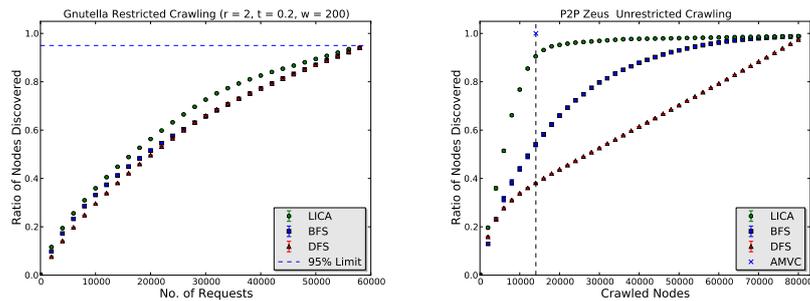
- *How well do the different crawling algorithms perform on real world datasets?*

LICA is expected to perform better compared to the other algorithms as it prioritizes backbone nodes and terminates as soon as the ratio of discovery falls below the threshold value. BFS is expected to perform better than DFS due to candidate prioritizing strategy adopted by this algorithm, i.e., first come first serve, which discovers more number of new peers than its counterpart.

In addition, since the best results – relative to the goals of LICA – in enumerating all bots in a botnet is to obtain a *minimum vertex cover*, it is of interest to identify which crawling algorithm perform closest



(a) Analysis of LICA's Parameter Selection (b) P2P Zeus Restricted Crawl Analysis on P2P Zeus Dataset



(c) Gnutella Restricted Crawl Analysis (d) P2P Zeus Unrestricted Crawl Analysis

Figure 8: 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) GameOver 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 GameOver Zeus dataset without any NL restrictions, plotted by the ratio of nodes discovered depending on the total number of nodes crawled.

to the *AMVC* value. Therefore, the following research question needs to be answered:

- Which crawling algorithm performs with the best efficiency?

LICA is once again expected to perform the best due to the presence of a natural backbone that can be leveraged by the crawlers. Meanwhile, *BFS* is expected to perform also slightly better than *DFS* due to the candidate selection strategy that discovers more number of unique peers in the beginning.

### 5.3.2.5 Results

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

**BEST COMBINATION OF PARAMETERS FOR LICA** First, it is 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 GameOver Zeus<sup>4</sup>, 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. The value  $R$  is set to 2 in all experiments because it is known from the GameOver Zeus datasets, that all algorithms need to request the NLs from any node at most twice, i.e., two chunks, to obtain the full NL.

Furthermore, by deciding combinations of values for the window size  $w$  and threshold  $t$ , the resulting crawl can also be shaped. For example, a 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. The values of the window size and threshold used in this work were obtained through a parameter study with various pair of combinations. The effects of different combinations of promising values are analyzed on the GameOver Zeus dataset with the value  $R = 2$  as presented in Figure 8a.

From the analysis, it is 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% fewer 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% despite requiring 1,800 requests more than the previous combination. Therefore, it is decided that the combination,  $t = 0.8, w = 2000$  is more reasonable to crawl the GameOver Zeus dataset efficiently. Unfortunately, the criteria for selecting the best combination of parameters are not straightforward, i.e., the combination of parameters can only be selected based on the basis of a trial and error. Although some general combination of values can be utilized, based on the requirement of the crawl, LICA performs better if fine-tuned with more appropriate parameters to improve the crawling efficiency, i.e., based on results from previous crawls.

**PERFORMANCE ANALYSIS OF CRAWLING ALGORITHMS** The results of the crawl performance analysis are presented in Figure 8b and 8c. LICA was executed on the GameOver Zeus dataset using the

<sup>4</sup> At the time of this work, GameOver Zeus allowed 12 requests to be received within a sliding window of a minute. This value was later changed in a subsequent botnet binary update to only 6 requests. (cf. Section 4.1.3)

previously chosen parameter combination:  $r = 2$ ,  $t = 0.8$ , and  $w = 2000$ . The crawl performance in Figure 8b indicates a much better performance of LICA in comparison to the other methods. Note that for the plot of LICA, the points in which there were results less than 16 individual experiments were omitted as a confidence interval cannot be obtained from that. LICA required 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 NLs from previously known nodes, i.e., subsequent chunks of their NL. 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 NLs or the returned subset of the list, the effort to crawl the entire network increases proportionally for all crawling algorithms. This is verified by running another set of experiment with a returned NL of size 30 and  $R = 3$ . The observed performance between the crawling algorithms remain relatively similar to the results in Figure 8b.

The experiment is repeated 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 8c is not as significant as in the GameOver Zeus dataset. Further investigation revealed that this behavior is due to the diameter of this dataset being very high, i.e., 31, with an average path length of 9.2. Moreover, nodes in this dataset have a rather low average size of the NL, e.g., 2.4 entries. Hence, due to the inherent network structure in this dataset which is unlike the structure of most P2P botnets, the gain is much lower, as all crawlers need to go through almost every available node to obtain a full view. Nevertheless, the performance of LICA is better compared to the other two algorithms as presented in Figure 8c. For example, with 31,941 requests, LICA discovered 75.5% of nodes that is about 7% more than the other algorithms.

**EFFICIENCY ANALYSIS OF CRAWLING ALGORITHMS** For this purpose, the performance of all the three crawling algorithms on the GameOver Zeus dataset are compared with respect to the *AMVC* value in Figure 8d. For simplicity, the simulation settings were modified 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 to obtain the full view of the network.

Based on this analysis, it is demonstrated 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 it indicates that by crawling and prioritizing the '*popular*' peers, the backbone of the network is being leveraged and crawled. This corresponds to the finding of Stutzbach et al. [SRS05] that reports the existence of biased connectivity with peers with higher uptime, i.e., *popular* nodes in this work. This allowed LICA to exploit the feature and outperform existing crawling algorithms.

In this work, the influence of the NL restriction mechanism of GameOver Zeus is not studied. The main reason for this is the fact that crawling algorithms discussed in this chapter do not address on the problem of retrieving all neighbors of a particular bot. Instead, the algorithms only focused on prioritizing bots to be crawled with the aim of discovering as many bots as possible with the least effort. Furthermore, the ZEUSMILKER algorithm that was proposed in Section 5.1.1 can be implemented in all crawling algorithms discussed in this chapter to circumvent the restriction mechanism. Therefore, the influence of the restriction mechanism can be ignored in this work.

### 5.3.3 Evaluation of The BoobyTrap Mechanism

The evaluation of the BoobyTrap (BT) mechanisms as proposed in Section 5.2.2 is outlined next, and the results have also been published in [Kar+16a]. First, the dataset utilized for evaluating BT is discussed in Section 5.3.3.1. Then, Section 5.3.3.2 elaborates the setup for the experiments. After that, the investigated research questions are listed along with the expectations of the outcome in Section 5.3.3.3. Finally, the results of the experiments are presented in Section 5.3.3.4.

#### 5.3.3.1 Dataset

The datasets that were used for evaluation were obtained using a real deployment of BT nodes, i.e., sensors, in Sality *Version 3* (cf. Section 4.2) and ZeroAccess *Network 2* (port 16470) (cf. Section 4.3). Each BT node was popularized for two weeks before the measurements were obtained. After that, the measurements were collected for a duration of one week 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). Table 8 presents the summary of the datasets.

Table 8: Statistics of the collected data

	<b><i>Sality (Version 3)</i></b>	<b><i>ZeroAccess (Port 16470)</i></b>
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

### 5.3.3.2 Experimental Setup

The experiments were conducted with BT-enhanced sensors that were implemented in *Python* language for both botnets, i.e., Sality and ZeroAccess. All detection mechanisms were triggered by the type and contents of the responses received (or missing) from a node. However, only for the frequency-based detection mechanisms, i.e., *Abuse* class, a configurable sliding window-based detection mechanism was implemented to help identify IPs aggressively contacting the BTs. This detection mechanism takes two input parameters: length of the sliding window  $t$  (in seconds) and the minimum number of messages  $n_{\min}$  to trigger the detection mechanism. If a particular node from an IP address sent more than  $n_{\min}$  messages within any observed sliding window, a detection will be triggered.

To evaluate the performance of the proposed mechanisms, the amount of IP addresses that triggered the BTs were considered and manual verification was conducted on the log data to identify if the behavior of a node behind an IP matched with that of a possible crawler. Some of the characteristics that are inspected and considered are listed in the following:

- Rate of consecutive request messages along with the pattern of utilized source ports (if any)
- Fixed values of certain fields within a message that would otherwise be random
- Not exchanging neighbors or botnet-specific update/attack payloads

Since manual checking cannot always yield a binary answer, i.e., yes or no, 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. 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 bot. In such cases, an explanation of why those bots were initially flagged is also provided. In some cases, it may also be possible to identify the organizations behind the detected crawling activities. However, in this work, identities of the parties responsible for crawling are not disclosed to prevent targeted attacks from the botmasters.

For the evaluation, an analysis to identify the best threshold values for the parameters  $t$  and  $n_{\min}$  in the frequency-based detection mechanisms, i.e., *SAB-BurstTrap* and *ZAB-BurstTrap*, is first done 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, the performance of the detection mechanisms described in Section 5.2.2 is evaluated based on the research questions presented

in Section 5.3.3.3. Finally, the common characteristics exhibited by the detected crawlers are discussed.

#### 5.3.3.3 *Research Questions and Expectations*

As mentioned in the previous section, BT detection mechanisms assume an IP address could only be associated to a single crawler or bot. However, the threshold value to trigger a detection in the observed sliding window can be configured in the detection mechanisms (when applicable) to take into consideration influences of bots behind NAT and proxies. Hence, the following research question needs to be answered in the evaluation:

- *What are the suitable threshold values to minimize false positives generated by bots behind NAT and proxies for frequency-based detection mechanisms, i.e., traps within the class of Abuse?*

As explained in Section 5.2.2, a BT node can be deployed in most of existing (or known) botnets. However, the question remains on how susceptible are current generation of crawlers against such crawler detection mechanisms. Therefore, the following research question needs to be answered in the evaluation:

- *How susceptible are current crawlers against the BT detection mechanisms?*

Considering that not much work has been done in this aspect and the fact that current and previous botnets have only implemented simple crawler detection/prevention mechanisms, it is expected that most of current crawlers are not anticipating such countermeasures. As such, many of the crawlers are expected to be detected by the BT detection mechanisms. However, it is also acknowledged that there may be some crawlers that were left undetected by BT.

Implementation of a crawler can range anywhere from bare minimum to full functionality support of a botnet's protocol (cf. Section 3.3.2). Moreover, a crawler can also adopt various strategies to improve its efficiency in crawling the botnets, e.g., multi-threading or distributed crawling. However, very little is known about the characteristics or design choice of the crawlers currently out in the wild. Therefore, the following research question needs to be answered in the evaluation:

- *What are the common characteristics of existing crawlers in the wild?*

#### 5.3.3.4 *Results*

This section is outlined as following. Firstly, the results of a parameter study for obtaining the threshold values for the BT mechanisms is presented. Then, the evaluation results of the crawler detection mechanisms adapted for Sality and ZeroAccess are presented. Finally, the common characteristics exhibited by the detected crawlers are presented.

**PARAMETER STUDY OF SUITABLE THRESHOLD VALUES FOR  $t$  AND  $n_{\min}$**  First, the best threshold values for the frequency-based detection mechanisms is investigated. For that, a parameter study of the various combination of parameters of  $t$  and  $n_{\min}$  is conducted. For Sality, the sliding window interval was varied, i.e., 60, 120, ..., 2400 seconds, and the experiments were repeated with different number of minimum messages required to trigger a detection, i.e., 10, 20, ..., 100 requests. The study 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  $n_{\min} = 40$ . It is possible that the higher number of messages required for ZeroAccess in comparison to Sality to trigger a detection can be due to the short MM-cycle interval of this botnet, i.e., 256 seconds.

**PERFORMANCE OF THE BOOBYTRAP MECHANISMS** Next, the performance of all detection mechanisms deployed in both botnets within the measurement period is evaluated. The overall results of the detection mechanisms are presented in Table 9 according to the respective classes of misbehaviors (cf. Section 5.2.2).

	Defiance			Abuse		Avoidance
	SD1	SD2	ZD	SAB	ZAB	ZAV
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 9: Performance of our *BoobyTrap* mechanism

For the class of *Defiance*, two BTs were set up for Sality (*SD1* and *SD2*) and one for ZeroAccess (*ZD*). 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 bots in Sality would simply ignore entries that are already known and responsive, i.e., a 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 4.2.2), 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, the BT node recorded *Hello* messages from a different IP than the one received for the  $NL_{Req}$ , thus triggering the trap. These cases were identified and sanitized by correlating a Sality-specific identi-

fier. As a result, we 77% of the detected IPs were identified 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, i.e., 1 (cf. Section 4.2.2.1). Hence, they were classified as *Unknown*.

The *ZD-NonComplianceTrap* was triggered by 88 IPs. Seven of those IPs, which were detected on the first day, consistently responded with a *retL* message containing a fixed flag, i.e.,  $\text{flag} = 0$ . These IPs are particularly interesting because they responded with exactly 65 messages before they stopped to contact the BT node. It is suspected that these are crawlers that implement a blacklisting mechanism to avoid crawling or contacting other sensors. These IPs were also observed to keep communicating with another instance of sensor node after the 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 possibility for such behavior is not observed in the reverse-engineered malware variants (cf. Section 4.3), there is no other explanation other than them being potential crawlers.

The 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 the results of the previous parameter study: Sality ( $t = 120$ ,  $n_{\min} = 30$ ) and ZeroAccess ( $t = 60$ ,  $n_{\min} = 40$ ). Evaluation results indicated 11 flagged IPs by the *SAB-BurstTrap*. Out of the 11, nine IPs were classified as *Highly Possible*. A daily analysis of this particular BT as presented in Figure 9 indicated that an average of four crawlers is successfully identified every day. Meanwhile, the single false positive was identified to be caused by many bots behind a single shared IP coincidentally contacting our BT node around the same time.

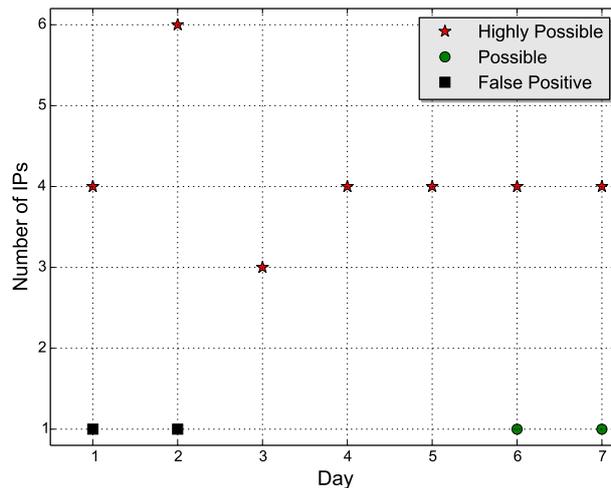


Figure 9: Daily analysis of *SAB-BurstTrap*

The *ZAB-BurstTrap* flagged a total of 188 IP addresses throughout the measurement period. After manual inspection, 116 IPs were classified as strongly exhibiting crawler-like behaviors. The remaining IPs that were classified as *Possible* exhibited similar behaviors to bots that lack responsive neighbors in their *NL* during their initial boot-

strapping phase. This speculation could especially be true considering that a large portion of the ZeroAccess botnet was sinkholed in 2013 [NG13]. As such, bots that experience lack of neighbors could also have requested *NLs* with a higher frequency.

The *ZAV-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 *ZD-NonComplianceTrap*. The evaluation of this BT indicates 108 IPs in the ZeroAccess dataset that never responded to any of the request messages. More precisely, 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 the crafted request messages. Seventy-three (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.

**CHARACTERIZATION OF DETECTED CRAWLERS** Finally, the various detected crawlers were analyzed to identify common characteristics exhibited by them. These characteristics can be categorized as the following:

- **Blacklisting :** Observations on the detected crawlers indicate that some of them are using blacklisting mechanisms to improve the quality of their crawl data, i.e., ignore sensor nodes. Such crawlers seemed to identify sensor nodes based on the type or content of the responses received from bots/sensors (explained next), e.g., empty or consistent duplicated *NL* replies.
- **Sanity Checking :** Some crawlers also seemed to perform sanity-checking on the results obtained from bots. The detection mechanisms within the class of *Defiance* detected a smaller fraction of crawlers than those within the class *Abuse*. For instance, the SD2 detected only three crawlers whereas SAB detected nine in total. Hence, it seems that there are crawlers that follow the implementation of the botnet protocols very closely, e.g, in deciding whether some entries need to be discarded (or not).
- **Aggressive and/or Persistent Crawling :** Upon analyzing crawlers detected via the frequency-based mechanism, it is noticed that only a minority of crawlers were observed to crawl the botnets continuously, i.e., 24x7. Nevertheless, some of these persistent crawlers communicated aggressively with the BTs with a high frequency, i.e., in average 15 requests per minute in the case of *Sality*.
- **Crawling Redundancies :** Some crawlers were observed to utilize identical botnet-specific identifiers and port numbers across

different instances at the same time in a given botnet. This observation may indicate identical crawler mechanisms deployed for redundancies or for obtaining additional crawling vantage points. Multiple vantage points for crawling may also help identify inconsistencies in crawl data that could be introduced by network failure or other network-specific issues in one or more crawlers. In such cases, results from a different crawler can be used instead.

- **Efficient Crawler Design :** Crawlers are also observed to use dedicated ports to process incoming NL replies. Unlike regular bots, each request message that was sent by the crawlers uses an identical source port number (cf. Section 4.3.2). This source port number corresponds to the port expected by the crawler to receive the response messages. Such a communication design can improve the crawling efficiency as it allows the processing thread to be independent of the thread that sends the requests. As a direct consequence, such crawlers do not require a thread to wait for a reply for each sent request message. Therefore, this separation of duty between the two threads would allow the crawler to crawl more bots in parallel.
- **Identity Hiding :** By performing *WHOIS* queries on the detected IP addresses, it is observed that only a few of the IP addresses disclose information about the organization or individual that is behind the crawling activities. In fact, quite some crawlers have been seen sharing residential IP addresses with bots, i.e., behind ISP NAT devices. In addition, IPs of some of these crawlers were also observed to constantly change due to dynamic IP address reallocation by ISPs, i.e., IP address aliasing. Such scenarios makes it more difficult to detect crawlers via any frequency-based detection mechanisms.
- **Neutral :** Finally, based on the crawlers identified in this work, it can be concluded that the detected crawlers are neutral. They do not seem to aid the botnets in any manner, e.g., dissemination of botnet commands. Even in cases where neighbors are being returned, these were either other sensors or invalid entries.

#### 5.4 CHAPTER SUMMARY

This chapter presented works on advanced botnet monitoring on the basis of crawling P2P botnets and outlined three major contributions as part of this dissertation. The first contribution presented a novel crawling algorithm called *ZEUSMILKER* (see Section 5.1.1) that deterministically spoofs keys to be used for requesting NL from bots to circumvent the GameOver Zeus' NL restriction mechanism. *ZEUSMILKER* is, to the best of knowledge, the first and the only known solution to provably retrieve all NL entries of a bot. Evaluation results of *ZEUSMILKER* as presented in Section 5.3.1.5 also indicated that state of the art meth-

ods are inferior to ZEUSMILKER. ZEUSMILKER is able to circumvent the NL restriction mechanism of GameOver Zeus by exploiting the deterministic neighbor selection mechanism (see Section 4.1.2.1) and the fact that keys included in the NL requests can be spoofed to manipulate the returned entries. Concluding, both above-mentioned factors have contributed jointly to the possibility of this anti-crawling mechanism to be circumvented by ZEUSMILKER.

To anticipate the retaliation of the botmasters against ZEUSMILKER, Section 5.2.1 proposed several enhancements, i.e., *Random Node Return*, *Bit-XOR+*, and *Bit-AND*, to the existing GameOver Zeus' NL restriction mechanism. These proposed countermeasures address the main drawback of the original mechanism: not allowing the requester to manipulate the returned entries completely. The evaluation of the new proposals indicated that *Bit-AND* performs best compared to the other two proposed mechanisms in impeding the performance of crawlers. However, this countermeasure adversely affects the resulting botnet overlay, so it is not likely to be used in future botnets. Therefore, the *Bit-XOR+* is most likely to be adopted by future botnets. Both above-mentioned mechanisms can affect the ability of a crawler to retrieve the (complete) NL of a bot, and therefore need urgent attention of the researchers as future botnets can easily adopt these mechanisms.

The second contribution in this chapter proposes a novel crawling algorithm called *LICA* (cf. Section 5.1.2) that attempts to enumerate as many bots as possible but by avoiding to crawl all bots exhaustively. *LICA* attempts to approximate a *minimum vertex cover* that represents the minimum set of bots that need to be crawled to discover all bots in the botnet. By prioritizing *popular* bots that are returned by other bots, this algorithm crawls the backbone of the botnet and can terminate as soon as the ratio of newly discovered bots falls under a certain threshold value. Evaluation results indicated that *LICA* outperforms other state of the art crawling algorithms, in particular BFS and DFS-based graph traversal techniques. Depending on the purpose of bot enumeration, this algorithm can be utilized in crawling P2P botnets in a more stealthy manner.

As the third contribution of this chapter, a lightweight crawler detection mechanism called *BoobyTrap (BT)* that exploits botnet-specific protocol and design constraints were proposed. *BT* aims at detecting crawlers in an autonomous manner by analyzing the communication of other bots with itself. Based on simple test-cases, a behavior of a crawler can be distinguished from bots. Evaluation results in Section 5.3.3 indicated that many crawlers in *Sality* and *ZeroAccess* can already be detected by *BT*.

The findings of the different work presented within this chapter imply that more advanced monitoring mechanisms are needed to tackle future P2P botnets. Such advanced mechanisms should focus on the following:

- **Larger pool of IP addresses:** Since most of the existing and proposed anti-crawling mechanisms are based on the notion of an

IP address representing a crawler, it is important to acquire a larger pool of IP addresses that can be used for future crawling activities. Most importantly, these IP addresses should not be of a single contagious block or range of IP addresses to avoid bots applying IP prefix-based blacklisting similar to that implemented by GameOver Zeus. Even if some of the IP addresses were blacklisted, additional IP addresses serve as redundancies to continue botnet monitoring activities. Such non-contagious block of IP addresses could also be obtained through the cooperation of several security organizations or institutions that may be interested in jointly monitoring the botnets.

- **Distributed crawling :** Moreover, future botnet monitoring activities should also consider using distributed crawlers in combination with a large pool of IP addresses to circumvent IP-based anti-crawling mechanisms, e.g., BT or NL restriction mechanisms, to capture the characteristics of bots accurately. This way, future crawlers are able to circumvent any IP address or frequency-based anti-crawling mechanisms through distributed crawling.

In the next chapter (Chapter 6), advanced botnet monitoring using sensor nodes is discussed. It also includes various novel countermeasures that will be proposed to detect deployed sensor nodes in the botnet.



The deployment of sensor nodes in P2P botnets enables an additional vantage point in monitoring bots. In particular, sensors are able to enumerate bots that are otherwise not discoverable by crawlers due to their inability to contact bots behind network devices like NAT and stateful firewalls (see Section 3.3.2).

A sensor node is deployed in a botnet by *announcing* its presence to other superpeers leveraging the node announcement mechanism of the botnet (see Section 2.1.3). Sensor announcements are often carried out by crawlers that (in)directly announce the presence of the sensor during crawling. After being well-known amongst many superpeers, the information of the sensor node will be frequently handed out to non-superpeers that may request additional neighbors from existing superpeers. Thereafter, a sensor node receives increased communication requests from non-superpeers that have added the sensor node as a candidate within their NL. As a consequence, an effective sensor is usually very *popular* amongst all bots in a botnet, i.e., known by many bots. Please note that although the BT mechanism proposed in Section 5.2.2 can be generalized as a sensor, the mechanism is aimed at detecting crawlers in the botnet. Since a crawler aims to discover as many bots as possible, a BT node does not need to be very popular among other bots. In contrast, sensors aim to be as popular as possible to increase their visibility to as many nodes as possible.

By combining the monitoring data of both crawlers and sensors, a more accurate enumeration of a botnet's population can be obtained for further analysis, i.e., informing affected stakeholders. In addition, a sensor node is also often used as a *sinkhole* server in botnet takedown attempts. Such sinkhole servers would usually aim to remain responsive to probing messages of bots as outlined in Section 3.2. However, they would not disseminate any new botnet updates or commands in order to prevent the bots from being able to be contacted or instructed by the botmaster. Therefore, sensors do not only pose as a threat to botnets due to its monitoring capabilities but also as a tool or stepping stone to launch botnet takedown attacks. Nevertheless, not much work has been done in the area of preventing and/or detecting sensors being deployed in P2P botnets.

Unlike the content organization of Chapter 5, due to the lack of prior work, this chapter first starts from a perspective of a botmaster in Section 6.1 to introduce three mechanisms to detect sensor nodes deployed in botnets. Then, from the perspective of a defender, Section 6.2 proposes countermeasures to circumvent the detection mechanisms. Finally, Section 6.4 concludes this chapter. Take note that some passages in this chapter are quoted verbatim from the following publications [Böc+15; Kar+16b].

## 6.1 DETECTING SENSOR NODES IN BOTNETS

As indicated by the lack of prior work, there are many challenges in successfully detecting sensor nodes deployed in P2P botnets. In contrast, this section will show that it is indeed possible to detect sensor nodes by relying upon graph-theoretic metrics of the botnet overlay. The remainder part of this section is outlined as follows:

Section 6.1.1 presents a short introduction on the basics of a sensor node and a discussion on the associated challenges in detecting the sensor nodes. In addition, a set of assumptions for a sensor detection mechanism is also derived and presented. Based on the outlined assumptions and the possible design choices of sensor nodes in Section 6.1.1, Section 6.1.2 presents the first sensor detection mechanism called LCC. This mechanism attempts to identify sensors based on the mutual inter-connectivity of neighbors that are being returned by a particular node.

Section 6.1.3 introduces the second detection mechanism called *SensorRanker*. This mechanism attempts to distinguish sensor nodes from popular backbone bots using a variation of the *PageRank* algorithm. Finally, Section 6.1.4 introduces the third detection mechanism called *SensorBuster* that discerns sensor nodes from regular bots based on the availability of connected paths from a node into the tightly connected backbone of the botnet overlay and back to the node itself.

### 6.1.1 Introduction

This subsection provides some basic introduction on sensor nodes as well as the related issues in detecting them from the perspective of a botmaster. Particularly, Section 6.1.1.1 introduces and discusses the most important feature of a sensor node: handling of botnet request messages. The handling of the botnet messages is important to enable the sensor node to communicate with other bots and enumerate them at the same time. Section 6.1.1.2 briefly describes the process of deploying the sensor node in a P2P botnet. Section 6.1.1.3 presents a set of assumptions for a sensor node detection mechanism that were derived from the discussions of Section 6.1.1.1 and 6.1.1.2 and own observations on sensor nodes deployed in existing botnets. These assumptions are important and are the basis for the remaining work presented in this chapter. Finally, Section 6.1.1.4 discusses the challenges often faced in detecting sensor nodes.

#### 6.1.1.1 Message Handling by a Sensor Node

The main functionality of a sensor node is the handling of botnet-specific communication messages, i.e., responding request messages with valid replies. Without handling the messages, a sensor will not be able to enumerate bots. However, a sensor node is only required to handle some of the important and relevant messages for the purpose

of botnet monitoring. Some of the most common types of messages and how they are often handled are detailed in the following:

1. **(Responsiveness) Probe Messages :** This type of message, i.e., *probeMsg*, is usually sent by bots to assert the responsiveness of a particular bot (see Section 3.2). In the case of a sensor node, bots send this message to check if the sensor node is still responsive. If the sensor node fails to respond to these messages, the bots will eventually flush the entry of the sensor node from their NL. Therefore, a sensor node needs to always be able to handle and respond to this type of message to ensure it remains in the NL of the bots. Furthermore, by remaining in the NL of many bots, the information about the sensor can be quickly propagated by existing bots to newer bots that may require additional neighbors.
2. **NL Request Messages :** This type of message, i.e., *requestL*, is usually sent by crawlers and bots that require additional neighbors (see Section 3.2). Although this message handling is often not important for a sensor node, it does help the sensor to blend in among other bots, i.e., remain under radar. For instance, a sensor node that deliberately ignores handling or replying all such messages may easily raise suspicions to an attentive botmaster. Based on researcher's observations, three approaches were seen being adopted by sensor nodes in existing botnets. There are some sensors that either do not respond at all to such request messages, i.e., ignoring the messages, or respond with just an empty but valid reply. The second approach of returning empty replies can be more suspicious although such replies are also valid in some botnet protocols, e.g., GameOver Zeus and Sality.

Some sensors are also observed to return information of other sensor nodes in the replies. This approach is not only a more stealthier technique, i.e., exhibiting similar behavior with regular bots, it also ensures that the sensor do not participate or help the maintenance of the botnet overlay by returning valid bots (Functional Requirement 4). Besides that, this approach can also be leveraged to help popularize another sensor node by frequently handing out information of other sensor nodes. Another variation of this approach that was observed is to return invalid or non-existing entries as neighbors in the replies. However, such an approach will introduce a significant amount of noise that would not only taint monitoring data of others, but could also cause harm to the normal maintenance activities of the botnet overlay (Functional Requirement 4).

3. **Botmaster Command Request Messages :** This type of message is sent by bots to query if there are any newer updates from the botmaster that can be downloaded by the bots, e.g., *Hello* message for Sality or *VersionRequest* for GameOver Zeus. However, all of the three analyzed botnets integrate the functionality

of this message along with the *probeMsg* described earlier. Bots use sequence numbers to indicate the most current botmaster command known to them. Whenever a bot shares the information in its replies that it knows of a command with a higher sequence number, other bots will try to pull the latest update, i.e., command dissemination via bot-to-bot. As such, the probe for the responsiveness of a bot also checks at the same time if its neighbor has any new updates from the botmaster.

Most sensors observed in the wild for the three botnets (see Section 4) try to avoid returning any new or valid update by reporting that they have an older update or lower sequence number when being requested. This way, the sensors do not help out in disseminating the command of the botmaster to other bots. Similarly, although there were some sensor nodes that reported having the newest update, they were observed to deliberately not sharing the newer update when being requested.

#### 6.1.1.2 *Popularizing the Sensor Node*

After the development of a sensor node, the next step is to deploy it within the botnet (see Section 3.3.3). This is often done by leveraging the node announcement mechanism of the botnet to *popularize* the sensor node. Although it is usually sufficient for a sensor node to be popularized only once in the botnet, some botnets may require constant or more frequent popularization efforts to stay popular in the botnet. One example of such a botnet is ZeroAccess (see Section 4.4) due to the very short MM-interval that quickly flushes out entries in the NL.

Popularization of a sensor node is usually performed in tandem by a crawler. Therefore, aggressive popularization strategies such as *Popularity Boosting* by Yan et al. [Yan+14b] can be easily detected by crawler detection mechanisms such as the BT mechanism (see Section 5.2.2). However, a slow and non-aggressive sensor node popularization technique is sufficient to deploy the sensor node in most botnets. The only drawback with a slow popularization technique is the fact it takes longer before most bots get to know of the sensor node.

#### 6.1.1.3 *Assumptions for a Sensor Node Detection Mechanism*

In the following, a set of assumptions for a sensor node detection mechanism is presented based on the discussions in Section 6.1.1.1 and 6.1.1.2. Please note that these assumptions were derived from own observations of the sensor nodes deployed in the wild as well as from the necessity to adhere to legal requirements.

1. **A sensor node is already deployed in the botnet.** As explained in Section 6.1.1.2, aggressive popularization strategies of a sensor node can be easily detected by BT-like detection mechanisms. However, the sensor node can still be popularized using a much slower and less aggressive strategy to evade detection of such

mechanisms. Therefore, this assumption forces a sensor detection mechanism to focus on advanced sensors that are already deployed in botnets; those that have evaded BT-like detection mechanisms.

2. **The sensor node does not return any valid bots as neighbors.** A sensor node should adhere to legal requirements of many countries whereupon any botnet monitoring node should not actively or knowingly aid the bots in the regular maintenance of the botnet overlay or malicious activities of the botnet. Therefore, a sensor node should not return any valid bots when being requested for additional neighbors. For that, the node can either ignore replying the NL request message, or return non-bot entries, i.e., information of another sensor node or non-existing entries.
3. **The sensor node does not disseminate or exchange any valid update or command from the botmaster with other bots.** Similar to reasoning of the previous assumption, this too is relevant in the context of adhering to legal requirements by not actively participating in a botnet related maintenance or its malicious activities. For that, a sensor node should avoid returning any botmaster command or update that may benefit the botnet in one way or another. This can be easily achieved by bluffing on the latest command that is known to the sensor node or by ignoring the request for such updates.
4. **The total number of sensor nodes under the control of any attacker is lesser than the total number of bots.** For simplicity, an attacker, e.g., a researcher, is assumed to be able to deploy only lesser number of (colluding) sensor nodes than the total number of bots in the botnet, i.e., 50%. This assumption holds because it usually does not require many sensor nodes to be deployed to monitor a botnet. Furthermore, any party that might have the control of more sensors than the total number of bots would have already tainted most of the botnet monitoring data, e.g., churn measurements or crawling results, rendering any collected monitoring data almost useless.

#### 6.1.1.4 *Challenges in Detecting Sensor Nodes*

The main challenge in detecting a sensor node is the difficulty in distinguishing it from regular bots in the botnet [ARB15]. For instance, sensor nodes are usually very popular among bots in the botnet overlay, i.e., high indegree. However, this observation is also true with superpeers that have been responsive for a long period of time and became a part of the backbone of the botnet.

In addition, a sensor node generates only minimal network traffic as it only responds to incoming requests in comparison to crawlers that actively generate requests at high frequencies [ARB15]. Moreover, the passive nature of sensor nodes also makes them even more difficult to be distinguished from regular bots.

In contrast, the work presented in Section 6.1.2–6.1.4 will show that graph-theoretic metrics can be used to distinguish sensor nodes from regular bots.

### 6.1.2 Local Clustering Coefficient (LCC)

This first sensor detection mechanism attempts to detect sensor nodes based on the inter-connectivity relationship amongst neighbors being returned by a particular node. This detection mechanism exploits the observation that in unstructured P2P botnets, nodes with high uptime tend to establish neighborhood relationships among themselves and thus form a backbone. As these backbone nodes are also popularly contained in the NLs of most bots in the botnet, there is a high probability that an ordinary bot has multiple backbone bots in its NL.

The degree of inter-connectivity of the neighbors of a bot can be represented by the *clustering coefficient* ( $cc$ ) metric. The clustering coefficient is often used to express the density of networks. In this work, the *Local Clustering Coefficient* ( $LCC$ ) introduced by Watts and Strogatz [WS98] is used to express the connectivity of a node's neighbors by computing their degree of inter-connectivity. Extreme values of 0.0 and 1.0 indicate that the neighbors are either *not* connected amongst each other at all or that they completely *mesh*.

To detect sensors, snapshots of the botnet overlay, i.e., network topology, is required to be captured using a crawler (cf. Section 3.3.2). The *directed* variant of the LCC is calculated using these snapshots for each bot  $x$ ,  $lcc^+(x)$ , to analyze the inter-connectivity of its neighbors by using Eq. (3).  $E$  is the set of all edges in the network and  $NL_x$  represents the NL of a bot  $x$ . The mechanism sets  $lcc^+(x) = 0.0$  if  $|NL_x| = 0$  or 1 (the numerator will fast-evaluate to 0)

$$lcc^+(x) = \frac{| \{ (u, v) \in E : u, v \in NL_x, u \neq v \} |}{|NL_x| \times (|NL_x| - 1)} \quad (3)$$

Figure 10 presents the analysis results of LCC on Sality V3 for a given snapshot on the LCC values of nodes in dependence to their popularity, i.e., indegree. From the results, it can be seen that bots exhibit a similar degree of inter-connectivity in their neighborhood due to the presence of common backbone nodes in their NLs, i.e., a majority of the bots having  $0.6 = lcc^+(x) \leq 0.8$ . However, according to the assumptions laid out above, sensor nodes do not share or give away information of legitimate bots when they receive a NL request. Thus, their  $lcc^+$  will differ from that of bots.

As depicted in Figure 11, a sensor has three possible behaviors upon receiving a NL-request as explained in Section 6.1 if it is to follow strictly the ethical standards of not contributing to the botnet overlay maintenance :

1. Return *no* neighbors or ignore the request. This behavior will lead to  $lcc^+(x) = 0.0$ , e.g., Sensor A in Figure 11.

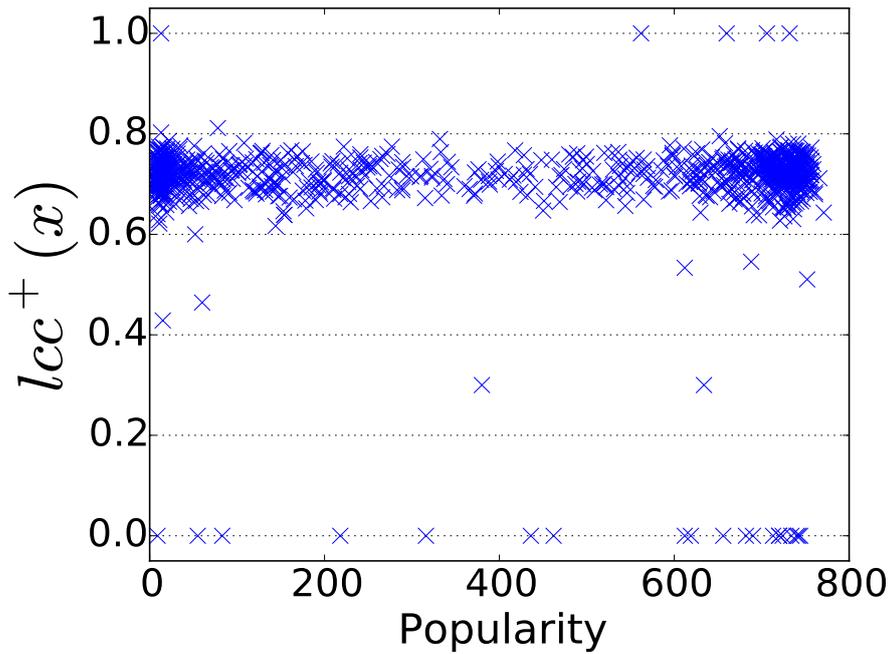
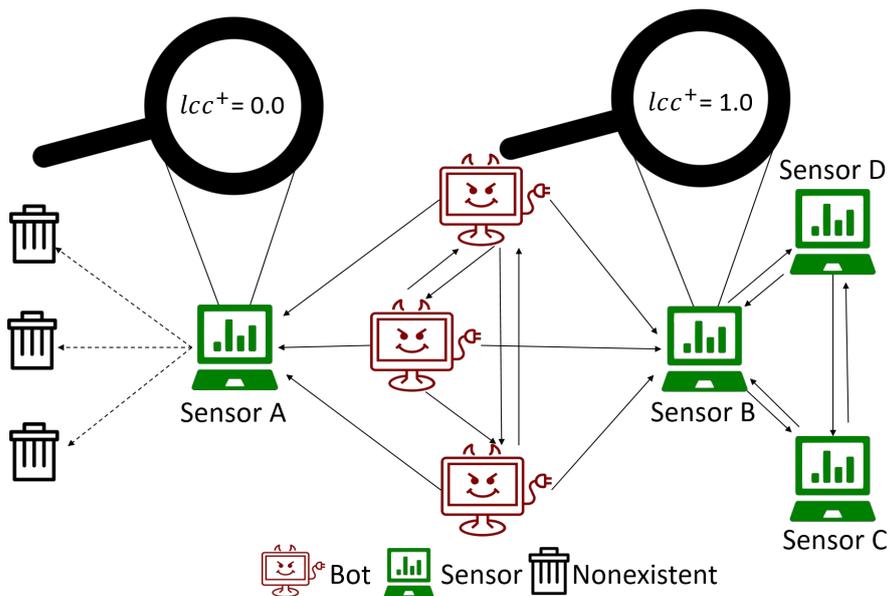


Figure 10: LCC values of bots in Salty V3.

Figure 11: Extreme values of LCC can be used to identify sensors deployed within a botnet overlay, i.e.,  $lcc^+(x) = 0.0$  or  $1.0$ .

2. Return only *invalid* neighbors. As invalid neighbors are not interconnected or even contactable in the first place, again  $lcc^+(x) = 0.0$  holds, e.g., Sensor A in Figure 11.
3. Return only *responsive sensors*. If each of the returned sensors return each other as their neighbors,  $lcc^+(x) = 1.0$  will hold since the sensors are in a full mesh, e.g., Sensor B, C, or D in Figure 11. However, if the connectivity between the returned sensors

is as in a directed cycle or not connected at all, it will lead to  $lcc^+(x)=0.0$ .

As stated earlier and depicted in Figure 10, due to the tendency of having mutual backbone nodes in the NL, regular bots will not have these extreme  $lcc^+$  values, i.e., 0.0 or 1.0. Therefore, nodes exhibiting extreme values should be flagged as potential sensors by LCC.

### 6.1.3 *SensorRanker*

The second sensor detection mechanism is called *SensorRanker*. It primarily uses the *PageRank* algorithm [Pag+99] to distinguish sensors from bots. The PageRank algorithm was initially designed to determine the importance, i.e., popularity, of websites based on the number of pages referring to them via hyperlinks. Towards this a relation of websites and hyperlinks is modeled as directed graphs with the websites as *nodes* or *vertices* and the hyperlinks as *edges*. The similarity to the formal model of botnet presented in Section 3.2 is obvious. This model is extended in the following to describe PageRank and SensorRanker in more detail.

**EXTENDED BOTNET FORMAL MODEL** The neighborhood relationship, i.e.,  $NL$ , of a peer  $v \in V$  can be defined as the successors of  $v$ ,  $succ_v = NL_v = \{u | \forall u \in V : (v, u) \in E, v \neq u\}$  that contains the set of all peers to which  $v$  has an outgoing connection. The NL can also be more specifically expressed as  $NL_v^t$  to reflect the exact view, i.e., neighbor entries, of the NL of peer  $v$  at time  $t$ . Consequently, the set of bots that have bot  $v$  as their neighbors or incoming connections to  $v$  can be expressed by the set of predecessors  $pred_v = \{u | \forall u \in V : (u, v) \in E, v \neq u\}$ .

The PageRank algorithm assigns values between 0.0 and 1.0, where a higher value denotes higher *rank* or popularity of a node  $v$ , e.g.,  $PR_v = 1.0$ . The values are calculated based on a node's predecessors  $pred_v$  and their respective ranks. In each iteration of the algorithm, the rank of a node is distributed equally among all its outgoing edges, i.e.,  $succ_v$ . The rank-value distributed over all edges of a node  $v$  is expressed as  $edgeweight_v = \frac{PR_v}{|succ_v|}$ . The PageRank value of a node, in turn, is the *sum* of the edge-weights of all of its predecessors.

The concept of ranks in PageRank is also directly comparable to the popularity of bots in a P2P botnet, i.e., the rank of a bot increases proportionally with the number of bots having it in their NL. Bots become more widely known and popular in the botnet when they have been available and responsive for a prolonged period. However, sensors are also observed to be equally popular when they are widely known amongst many bots. As such, popularity alone is not sufficient nor effective to distinguish sensor nodes from popular bots [ARB15]. However, when taking PageRank into consideration, the edge-weights on outgoing edges for sensor nodes differ greatly than popular bots because they have, either none or very few outgoing edges compared to the popular bots, i.e., sensor nodes have significantly higher edge-weights due to very few (if any) outgoing connections. This discrep-

ancy can be exploited to distinguish sensors from bots using the edge-weight as a reliable metric.

Nevertheless, due to the churn dynamics in P2P botnets, using the original PageRank algorithm as it is may indicate unpopular bots, i.e., bots known only by a small fraction of superpeers, that have some predecessors that are coincidentally with very high PageRank values, to appear as having a very high edge-weight. The edge-weight of a node  $v$  is normalized to address this drawback by multiplying it by its *popularity ratio*, i.e., ratio of predecessors over the size of the botnet population. This adapted PageRank algorithm is the proposed *SensorRank* value and is defined as:

$$SensorRank_v = edgweight_v \times \frac{|pred_v|}{|V|} \quad (4)$$

Although the *SensorRank* values for sensors would be significantly higher than those of bots, a means to automate the detection of sensors is still needed from the perspective of a botmaster. For this, clustering algorithms from the domain of machine learning are utilized to assist in further distinguishing sensor nodes from bots. The details of the clustering algorithms are elaborated later in Section 6.3.2.

#### 6.1.4 *SensorBuster*

The third proposed mechanism is called *SensorBuster*. It utilizes the SCC connectivity metric introduced by Robert Tarjan [Tar72] to identify sensor nodes. An SCC of a directed graph  $G$  is defined as a maximum set of vertices  $C \subseteq V$  with a directed path between each pair of nodes  $(u, v) \in C$ , i.e.,  $u \rightarrow v$  and  $v \rightarrow u$ .

Considering that P2P botnets rely heavily on the inter-connectivity between bots to prevent segmentation or partitioning of the overlay, bots  $B \subseteq V$  often form a *single* SCC where there is a path to and from one bot to another. From here onward, such an SCC is referred to as the *main* SCC. Please note that from the fourth assumption presented in Section 6.1.1.3, the largest SCC can be safely assumed as the main SCC, i.e., the SCC formed by bots are larger than those formed by sensor nodes. Without the main SCC, any new command disseminated in a botnet would require a much longer period to reach all bots. Due to the assumptions about sensor nodes presented in Section 6.1.1.3, a sensor would not be part of this main SCC, since a sensor node will not have any bot as its successor, i.e., no paths from the sensor node into the main SCC.

Therefore, sensors will either form their SCC consisting of either only a single sensor or multiple colluding sensors. As such, all nodes that are not included in the main SCC are most likely sensors that are deployed in the botnet.

## 6.2 CIRCUMVENTING SENSOR DETECTION MECHANISMS

The previous section proposed three detection mechanisms to detect sensor nodes deployed in botnets. In this section, from the per-

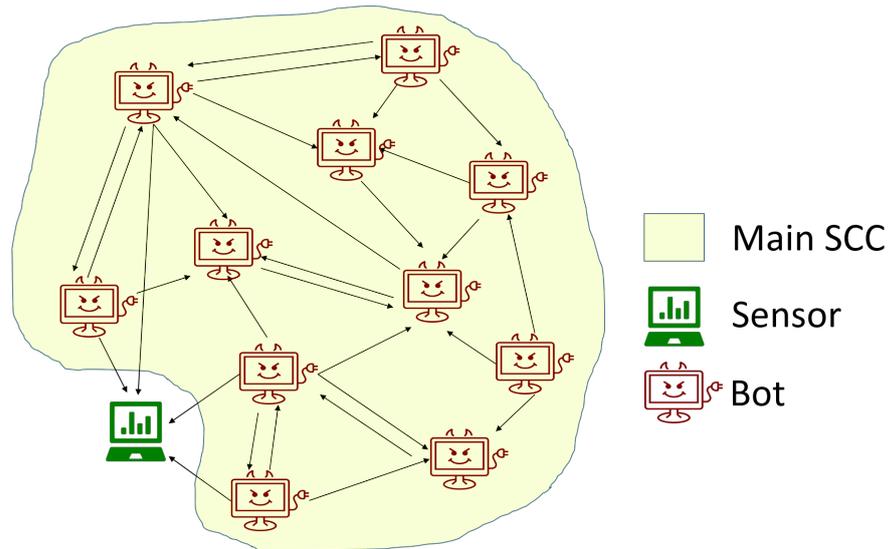


Figure 12: Sensor nodes that do not return valid bots as neighbors would not be part of the *main* SCC.

spective of a defender, e.g., security researcher, methods to circumvent the sensor detection mechanisms proposed in Section 6.1 are proposed. All proposed methods in this section require a set of colluding sensors to circumvent the detection mechanisms. Specifically, sensors utilize a distributed sensor deployment strategy called *Distributed Sensor Injection (DSI)*. In this strategy, sensors controlled by a user, e.g., a researcher, are used to manipulate the observed connectivity metrics of the sensors by intelligently distributing loads among multiple colluding sensors.

The DSI strategy assumes the following:

1. At least four colluding sensors are available for the usage of the user, i.e.,  $|S| \geq 4$ .
2. Colluding sensors communicate with the user using out-of-band communication channels.
3. The user can instruct the sensors to ignore communications from selected bots or attempt to *inject* or *announce* themselves into the NL of other bots.

The remainder of this section is outlined as follows: Section 6.2.1 introduces a method to manipulate LCC. Meanwhile, Section 6.2.2 presents a method to circumvent SensorRank. Finally, Section 6.2.3 discusses methods to evade SensorBuster.

### 6.2.1 Circumventing LCC

LCC calculates the clustering coefficient of each bot  $v$  and identifies sensor nodes that have extreme values of  $lcc^+(v) = 0.0$  or  $lcc^+(v) = 1.0$ . Such values indicate that neighbors of a bot  $v$  are not connected at all or connected in a full mesh respectively. As the majority of bots have common neighbors in the form of reliable backbone nodes [Böc+15],

it is unlikely that a bot could have neighbors that are not inter-connected at all, i.e.,  $lcc^+(v) = 0.0$ . Therefore, it could only be of sensors that refuse to share any neighbors or those that shared non-existent neighbors.

Moreover, due to large  $NL$  sizes in botnets such as Sality and ZeroAccess, i.e.,  $|NL| \geq 256$ , it is also unlikely that all neighbors seen in  $NL$  have exactly each other in their  $NL$ . Nodes having each other in their  $NL$  is more likely the case of a group of sensors attempting to popularize themselves and to increase vantage points for monitoring the botnet.

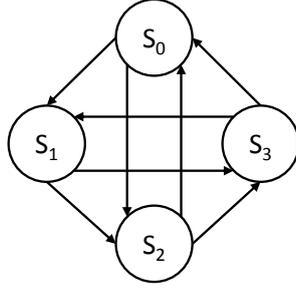


Figure 13: A simple example of colluding sensors evading the LCC mechanism

However, since this detection mechanism only detects extreme values, LCC can be easily circumvented by having sensors not exhibiting these extreme values. In more detail, by using a minimum of four sensors that are connected among themselves as depicted in Figure 13, each sensor can yield a clustering coefficient of  $lcc^+(S_i) = 0.5$ . The main idea is to avoid having a full mesh connectivity between the sensors but at the same time having some common connectivity with other sensors, i.e.,  $lcc^+(v) \neq 0.0$  and  $lcc^+(v) \neq 1.0$ . Therefore, for a group of  $N$  sensors where  $N \geq 3$ , a user needs to connect each sensor  $S_i, i \in [0, N]$  to all other sensors except  $S_{i-1 \bmod N}$  and to the sensor itself, i.e., to avoid self-loop.

### 6.2.2 Evading SensorRanker

While it was sufficient just to introduce a few interconnections among colluding sensors to circumvent the LCC mechanism, the SensorRanker mechanism requires a more sophisticated mechanism to evade detection. SensorRank focuses on the popularity of nodes; that is defined by their indegree. As such, the more popular a bot is, the higher the SensorRank value is. Therefore, to circumvent this detection mechanism, it is important that sensors avoid being *abnormally* popular. Moreover, having sufficient outgoing edges, i.e., neighbors, helps to reduce the SensorRank value of a particular node.

By connecting colluding sensors in the DSI strategy as mentioned above, a user can distribute the popularity of his sensors among the set of sensors in an optimal manner, i.e., distributed evenly with no redundant information. For instance, to obtain the full enumeration of the botnet's population  $|V|$ , one could distribute  $N$  sensors to have  $\frac{|V|}{N}$  predecessors. Please note that it is also important that the set of

predecessors of a sensor  $S_i$  be distributed evenly among all sensors based on the *edge-weight* of the bots that would also influence the SensorRanker detection mechanism (cf. Section 6.1.3). Such a distribution of sensors could not only reduce the popularity or the *SensorRank* value of each sensor significantly but the increased number of interconnections with other sensors also further reduces the *SensorRank* value of a sensor.

For the purpose of circumventing the SensorRank mechanism, one additional assumption is adopted: SensorRanker can be evaded if the SensorRank value of a sensor  $S_i$  is lower than or equal to the average value of all bots, i.e.,  $SR(S_i) \leq \text{Avg}(SR(V - S))$ . As such, the *minimum* amount of sensors needed to circumvent SensorRank, i.e.,  $N_{min} = |S|$ , can be calculated by satisfying Equation 5.

$$SR(S_i) = \frac{PR(S_i)}{N_{min} - 2} \times \frac{1}{N_{min}} \leq \text{Avg}(SR(V - S)) \quad (5)$$

In more details, this equation calculates the *SensorRank* value of a sensor  $S_i$  depending on the number of outgoing edges, i.e., other sensors, to satisfy the condition of being lesser or equal to the average value of all bots. However, depending on a botnet's *NL* size, this method could be resource-intensive or costly regarding the numbers of sensors needed to evade this mechanism. Detailed analysis through a simulation study on the feasibility of circumventing the SensorRanker mechanism is presented in Section 6.3.4.

### 6.2.3 Evading SensorBuster

The DSI strategies proposed above will remain detected by SensorBuster as the colluding sensors create their own SCC with no connection back to any bots within the botnet overlay. Evading SensorBuster requires, at least, one connection from any of the sensors to (and back from) the main SCC. However, to the best of knowledge, there are no possible strategies to evade this mechanism, unless an assumption that is presented in Section 6.1.1.3 is ignored, i.e., handing out bots as neighbors when being requested.

Therefore, for the sake of completeness, this assumption is ignored in the context of evading SensorBuster although the decision to return bots is debatable regarding this action's legality. By ignoring the assumption, the proposed DSI strategies not only can circumvent SensorBuster but also all other detection mechanisms, i.e., LCC and SensorRanker. For that, that particular assumption is relieved so that handing out legitimate neighbors would be permitted for sensors.

The following method circumvents the detection mechanisms by not only returning bots but also ensuring that the botnet receives only *minimal* benefits from the neighbors returned by the sensors. Stutzbach and Rejaie reported that nodes in P2P networks which already exhibit a long *uptime* would most likely continue to remain available [SR06]. In contrast, nodes that are newly seen have a higher probability of leaving the network immediately or soon. Since their

observation can also be generalized to P2P botnets due to similar nature of both networks, the observation is leveraged to only return neighbors that newly joined as such bots have higher tendency to be less useful, i.e., most likely to go offline soon. One could also argue that well-known bots should be returned instead of newbies. However, well-known or reliable bots that usually form the backbone of a botnet overlay are very important to the maintenance of the botnet overlay. In contrast, returning newbies allows the SensorBuster mechanism to be circumvented with only minimum assistance offered to the botnet.

For this, each sensor would be required to keep track of timestamps of the first and last point of contact of each superpeer, i.e., *firstSeen* and *lastSeen* respectively. Then, sensors may choose to alternate between returning other sensors and sometimes return legitimate neighbors by picking bots with the most recent *firstSeen*. By returning bots, sensor(s) would no longer form an isolated strongly connected component but merge with that of the whole botnet itself since there is a path to and back from the botnet's main component. Therefore, this method can effectively circumvent the SensorBuster detection mechanism while minimizing the benefits offered to the botnet's overlay or activities.

## 6.3 EVALUATION

This section presents the evaluation results and analysis of the mechanisms proposed in Section 6.1 and 6.2 as outlined in the following. Section 6.3.1 describes the datasets utilized for evaluating the sensor detection mechanisms. Then, Section 6.3.2 elaborates the setup for the experiments. Section 6.3.3 discusses the investigated research questions as well as the expected outcomes. Finally, Section 6.3.4 presents the results of the experiments. Please note that some of the results presented in this section have been published in the following publication [Böc+15].

### 6.3.1 Datasets

The datasets were obtained by continuously crawling the Sality (Version 3) and ZeroAccess botnets for a duration of one week, respectively. Sality was crawled from 08/10/2015 00:00:00 UTC to 14/10/2015 23:59:59 UTC. Due to the neighborlist return mechanism in Sality that returns only one entry when requested (cf. Section 4.2.2), a multi-session crawling that sends 30 simultaneous requests to each bot for every crawl session was conducted. Please note that for Sality, a *Hello* request message is additionally sent to each bot before crawling the bot for every crawl session. A response to the sent message allows the crawler to assert the responsiveness of a bot before crawling it. Meanwhile, the ZeroAccess botnet was crawled in a similar manner from 07/11/2015 00:00:00 UTC to 13/10/2015 23:59:59 UTC with a single request message that returns 16 neighbors for every crawl session.

From the initial 32,693 bots discovered in the Sality botnet, 4,131 bots have been pruned because they have never responded to any *Hello* messages, i.e., suspected artifacts resulting from bots that went offline. This resulted in only 9,306,998 out of 9,416,427 edges retained in the resulting dataset. Similarly, out of 95,668 bots discovered in the ZeroAccess botnet, 93,632 bots have been removed, because they never responded to requests, i.e., suspected artifacts resulting from bots that went offline and an ongoing active pollution attack. Consequently, only 343,314 out of 2,413,223 edges were retained in that dataset. The summarized details of both sanitized datasets are presented in Table 10.

Table 10: Summary of the sanitized datasets

	Sality (Version 3)	ZeroAccess
Total Bots	28,562	2,306
Hourly Avg. (Bots)	1,479	105
Max. Neighbors	656	134
Min. Neighbors	0	0
Avg. Neighbors	318	86
Median Neighbors	369	93

### 6.3.2 Experimental Setup

The performance evaluation experiments utilized *Python* scripts that were built upon the *NetworkX* [HSC08] and *scikit-learn* [Ped+11] modules to implement all three sensor detection mechanisms, i.e., LCC, SensorRanker, and SensorBuster. The detection mechanisms require the crawl data, i.e., snapshots, of the botnet’s overlay topology as an input to detect sensors on the basis of connectivity characteristics that are distinguishable from bots. However, several potential issues will surface in both using the snapshots and to accurately detecting sensor nodes. The following subsection addresses these concerns:

As it is often difficult to capture the ‘complete’ state of a bot’s NL at a given point in time, Section 6.3.2.1 suggests a method to split crawl sessions into smaller chunks and use them as a representation of a bot’s NL instead. Section 6.3.2.2 discusses how auxiliary data obtained during crawling can be used to help to assert a detection of a sensor node or to identify false positives. Meanwhile, Section 6.3.2.3 presents a mechanism to identify and remove artifacts within the obtained snapshots that could otherwise adversely affect the performance of the detection mechanism.

#### 6.3.2.1 Splitting crawl data into snapshots

Since the NL-reply mechanism that is adopted by both Sality and ZeroAccess prevents a crawler from capturing the complete  $NL_v^t$  of a bot  $v$  at time  $t$ , an approximation or a near-complete representation

is required as a replacement. Therefore, the results from the multiple crawl sessions are aggregated into hourly snapshots to represent as the *near-complete* NL of bots at any given hour  $t \in [1, 24]$ .

Each snapshot in the Sality dataset has an average of 81 crawl sessions, which corresponds up to about 2,430 requests sent to each bot within an hour. As for ZeroAccess, each snapshot in the dataset has an average of 299.7 crawl sessions. For simplicity, each of these snapshots is considered as a "complete" botnet topology at the given point of time, i.e., hours. Please note that any distortion in the captured snapshots does not adversely affect the ability to detect true sensor nodes.

In addition, only one snapshot per day is chosen and selected according to the one with the lowest number of bots seen in a day. Since sensors always attempt to be responsive (cf. Section 6.1), it is inherently assumed that a sensor would be responsive throughout every hour of a given day. Therefore, it is sufficient to execute the detection mechanisms on the snapshot with the least nodes. Furthermore, such a snapshot with a lower number of bots also reduces the probability of increased false positives (if applicable). From the analysis of the hourly snapshots within the datasets, the 5th snapshot of any day, i.e., 04:00:00 - 04:59:59, is the lowest for Sality. In comparison, it was the 7th snapshot of any day for the ZeroAccess botnet. Details of the seven selected snapshots for each botnet is presented in Table 11.

Table 11: Summary of the selected snapshots

	Sality (Version 3)	ZeroAccess
Total Bots	3,975	325
Hourly Avg. (Bots)	1,061	91
Max. Neighbors	564	111
Min. Neighbors	0	0
Avg. Neighbors	340	72
Median Neighbors	434	77

These snapshots are then utilized as inputs for each of the detection mechanisms. For simplicity, these selected snapshots are hereafter referred to as the respective botnet's dataset itself. After running the detection mechanisms on the input datasets, each mechanism will generate a list of IPs that are flagged as potential sensors.

### 6.3.2.2 Using auxiliary data to help in making decisions

Although the sensor detection mechanisms can flag potential sensors, it would be important for a botmaster to inspect the flagged nodes further before deciding if they are indeed true positives or false positives. For that, all metadata and payload contents of each received response should be logged by the crawler.

**STRENGTHENING CONFIDENCE** As described in Section 4.2.2, bots in Sality that receive a *Hello* message with an older *URLPack* would

respond by attaching the latest *URLPack* known to them. Hence, by transmitting an older sequence number of the *URLPack* within the sent *Hello* messages before crawling a bot, the corresponding response should consist of an attached *URLPack* from the bots. Similarly, all *getL* messages that are sent to a *ZeroAccess* bot should also be responded with a *retL* message that would consist of all *plugins* that are available for download from the responding bot. The details of which *URLPack* or *plugin* is missing within the responses are logged as part of the auxiliary data. In addition, the corresponding neighbors returned by bots in *Salicy* for each received  $NL_{Rep}$  is logged. Similarly, the neighbors within the received *retL* messages for bots in *ZeroAccess* is also logged.

Based on auxiliary data, it is possible to strengthen the confidence of accurately flagging a sensor node by inspecting if there were any logged misbehavior, e.g., missing *URLPack* or *plugins*. Please note that although the usage of the auxiliary data itself can be used as a technique to identify sensors, there are an arbitrary number of reasons that can skew the results of such as detection technique, i.e., network-specific anomalies. For instance, Andriess et al. reported that they were not able to observe any node that refused to exchange the *URLPack* in *Salicy* [ARB15]. In contrast, a more current analysis conducted within the scope of this work indicated that there were indeed nodes that are refusing to exchange their *URLPack* when requested. Hence, the collected data is only used as a reference to further *strengthen* the confidence of any detection flagged by the proposed detection mechanisms.

**IDENTIFYING FALSE POSITIVES** It is also possible to identify false positives that could have been triggered by detection mechanisms due to temporal network issues experienced by bots based on historical data. For instance, all neighbors of a bot could coincidentally be offline at the same time and forced the bot to exhibit the behavior of not having any valid neighbors to share. However, by looking at past or future historical records of the particular bot, it is possible to identify such a scenario and mark the detection as a false positive correspondingly.

**CLASSIFYING FLAGGED NODES** By relying on the collected auxiliary data, nodes flagged by the detection mechanisms can be classified into the following categories:

1. **Sensor:** Nodes exhibiting characteristics that correspond to the assumptions in Section 6.1
2. **False Positive:** Nodes exhibiting characteristics that conform to the botnet protocol
3. **Unknown:** Nodes that are not able to be classified as either *Sensor* or *Bot* due to lack of information

Therefore, a node is classified as a sensor node if there are any misbehavior or anomalies observed within the auxiliary data, e.g., refusal

to exchange neighborlists or botnet-specific commands. Meanwhile, a node is classified as a false positive if there are no misbehavior or anomalies observed and all behavior of the node conforms to the botnet protocol. Finally, a node is classified as unknown if it is not able to be classified due to lack of information.

### 6.3.2.3 Handling of Churn Artifacts

Since the detection mechanisms heavily depend on connectivity-specific metrics, artifacts introduced during crawling may skew the accuracy of the detection mechanisms. Two different strategies can be adopted to remove such artifacts from the datasets. The first strategy is to remove bots that have no neighbors at all, i.e., these bots have been returned as neighbors by other bots, but never responded to any NL-request messages. However, this strategy is not recommended as it would also remove sensors that are designed not to respond to such messages.

The second strategy is to measure and utilize the *responsiveness* of a bot to identify and remove artifacts. The *responsiveness* of a bot  $v$  at time  $t$  can be expressed as the ratio of the number of received replies to the number of sent requests between  $t - \delta t$  and  $t$ :

$$R_v^t = \sum_{\tau=t-\delta}^t \text{rep}_v^\tau \times \frac{1}{\sum_{\tau=t-\delta}^t \text{req}_v^\tau} \quad (6)$$

The responsiveness for bots in Sality is measured based on the number of received *Hello* replies while ZeroAccess simply uses the number of received *retL* messages. By specifying the *minimum* ratio of responsiveness that is required for any bot within a given snapshot, poorly responsive bots can be identified as artifacts or churn affected nodes. For instance, a value of  $R = 0.4$  represents all bots that have been responsive at least 40% of the whole monitoring period, i.e., the total number of crawl sessions within that particular snapshot. Take note that a value of  $R = 0.0$  is used to represent bots that have responded at least once to the probe requests. Sensors usually do not get flagged as artifacts as they would aim to be responsive to the probing messages as much as possible unless they are experiencing poor network connectivity. Please note that these identified artifacts are not directly removed from the dataset, but only being flagged as such. Removing artifacts from the dataset may lead to cascading changes and/or alter connectivity metrics of other bots within the snapshot that could adversely affect the performance of the detection mechanisms. Therefore, all detection mechanism would simply ignore nodes without a minimum responsiveness threshold value from their final classification.

### 6.3.3 Research Questions and Expectations

In the following, research questions that focus on the evaluation of the sensor detection mechanisms are presented along with investigated parameters.

An important aspect of any detection mechanism is the need to specify the baseline or ground truth. Hence, the following research question needs to be answered in the evaluation:

- *How to establish ground truth on the total number of sensors present in the datasets?*

Since it is difficult to establish such ground truth in botnets, the SensorBuster mechanism is used to provide a baseline information on the maximum number of sensors present in the datasets. The simplistic design of SensorBuster allows it to detect sensor nodes that adhere to the assumptions presented in Section 6.1.1.3. Since sensor nodes are assumed to not return a bot as a neighbor, sensors would establish isolated *strongly connected component(s)* which will not have any bots inside them. Hence, all IPs flagged by this mechanism needs to be inspected, and the total number of sensors detected by this mechanism is to be assumed as being the maximum number of sensors (or *True Positives*) present in the datasets. Therefore, *False Positives* of any other detection mechanism corresponds to an action of accidentally flagging an IP as a sensor that does not belong to the set of valid sensors found by the SensorBuster mechanism. Similarly, *False Negatives* are the inability of any other detection mechanism to detect a sensor flagged by the SensorBuster within the dataset. In the investigation to answer this research question, each snapshot within the datasets is evaluated using SensorBuster with varied *responsiveness* threshold  $R$  from 0.0 to 0.9.

Next, since the SensorRanker mechanism relies upon clustering algorithms to help distinguishing sensors from bots, the following research question needs to be answered in the evaluation:

- *Which clustering algorithm is suitable in distinguishing sensors from bots when applied to the results of the SensorRank mechanism?*

For that, the effectiveness of five clustering algorithms, namely *K-Means*, *DBSCAN*, *Gaussian Mixture Models*, *SpectralClustering*, and *Agglomerative Clustering* from the *scikit-learn* module is investigated to be used in SensorRanker to classify sensors accurately. These algorithms were chosen to be investigated due to their simplicity (in operation) as they easily create two clusters from a given set of data, i.e., distinguishing between bots and sensors. Unlike other clustering algorithms, these algorithms only require an input parameter of how many clusters needed, i.e., two, besides the SensorRanker data. The effectiveness of a clustering algorithm is evaluated by the highest number of classified sensors, i.e., *True Positives*, in combination with the lowest number of generated *False Positives*. This experiment is conducted to the *responsiveness* threshold of  $R = 0.0$  for each snapshot

within both datasets. Based on the results of the experiment, the best algorithm is then chosen and used for subsequent analysis.

As artifacts present in datasets can highly influence the accuracy of the detection mechanisms, the following research question needs to be answered in the evaluation:

- *How influential are the artifacts which are present in datasets towards the accuracy of the detection mechanisms?*

All three detection mechanisms are evaluated for each snapshot in both datasets with varying *responsiveness* threshold  $R$  from 0.0 to 0.9 to answer this question. It is expected that with a higher value of  $R$ , the number of artifacts or false positives decreases accordingly (if applicable).

It would be of interest to identify and evaluate the strengths of each of the proposed detection mechanisms. Therefore, the following research question needs to be answered in the evaluation:

- *Which mechanism performs best amongst the three sensor detection mechanisms?*

For this, the performance of all three mechanisms is evaluated and compared on both datasets. This evaluation is performed by selecting the appropriate clustering algorithm for SensorRanker and best values of  $R$  to eliminate the influence of artifacts within the datasets by answering the previous research questions.

Finally, the feasibility of the methods proposed to circumvent the detection mechanism in Section 6.2 need to be analyzed. Hence, the following research question needs to be answered in the evaluation:

- *How feasible are the proposed DSI strategies in circumventing the three detection mechanism?*

An analysis is conducted on the Salty dataset as a case study to demonstrate the feasibility or applicability of the proposed method to circumvent SensorRanker. In addition, a short analysis on the feasibility of the DSI strategies proposed for circumventing LCC and SensorBuster is conducted and presented.

#### 6.3.4 Results

In answer to the research questions presented in Section 6.3.3, the evaluation results are presented.

**ESTABLISHING BASELINE INFORMATION** The SensorBuster mechanism is evaluated using both datasets with varying values of *responsiveness* threshold  $R$  between 0.0 and 0.9 (see Section 6.3.2.3) to establish the baseline information on the maximum number of sensors present in the datasets.

Analysis of the Salty dataset indicated a combined total of 61 nodes were flagged by the SensorBuster mechanism with the minimum value

	Minimum Responsiveness Threshold, $R \geq$									
Day	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	10	10	10	9	9	9	8	8	7	7
2	11	11	11	11	10	10	9	9	8	8
3	11	11	11	11	11	11	11	11	10	9
4	11	11	11	11	11	11	11	11	11	10
5	10	10	10	10	10	10	10	10	10	9
6	10	10	10	10	10	10	10	10	10	10
7	10	10	10	10	10	10	10	10	10	10

Table 12: Maximum sensors present on a particular day dependent on R in the Sality dataset

of  $R = 0.0$ , i.e., all bots that have responded to at least one probing request. Out of these 61 nodes, 11 nodes were verified to be *Sensor* and seven of them were classified as *Unknown* based on manual analysis using the auxiliary data (see Section 6.3.2.2). However, all nodes initially classified as *Unknown* were later identified and flagged as artifacts for all values of R 0.0. The remaining 43 nodes were false positives in the form of artifacts present within the snapshots, i.e., churn affected nodes that were present for only a short period. Detailed results of the total number of sensors present in the different snapshots are provided in Table 12 in dependence on the various values of R. The column in the table represents the minimum responsiveness threshold values that are required for a particular sensor to be considered. Whereas, the rows represents a particular day or snapshot within the dataset. As such, a value within the table should be interpreted as the total number of sensors that were present on a particular day or snapshot with a responsiveness ratio of at least R.

	Minimum Responsiveness Threshold, $R \geq$									
Day	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	3	3	3	3	3	3	3	3	3	3
2	3	3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3	3	3
4	3	3	3	3	3	3	3	3	3	3
5	3	3	3	3	3	3	3	3	3	3
6	3	3	3	3	3	3	3	3	3	3
7	3	3	3	3	3	3	3	3	3	3

Table 13: Maximum sensors present on a particular day dependent on R in the ZeroAccess dataset

Meanwhile, the analysis on the ZeroAccess dataset indicated only a total of four nodes flagged by the SensorBuster mechanism with the minimum value of  $R = 0.0$ . Out of these nodes, three nodes were verified to be *Sensor* and one was a false positive. The analysis of total

number of sensors present in the different snapshots is presented in Table 13 in dependence to the various values of  $R$ .

The information from Table 12 and Table 13 are then assumed as the ground truth for the total number of sensors present in the dataset for a particular responsiveness threshold value of  $R$ . This data is then used throughout the remaining evaluations in this paper. Please note that the nature of the SensorBuster mechanism itself ensures that all sensors that follow the assumptions presented in Section 6.1 are already detected and represented in Table 12 and Table 13. As such, SensorBuster can be leveraged to establish the ground truth for all sensors that are present in both datasets or at least for the scope of evaluating the proposed mechanisms.

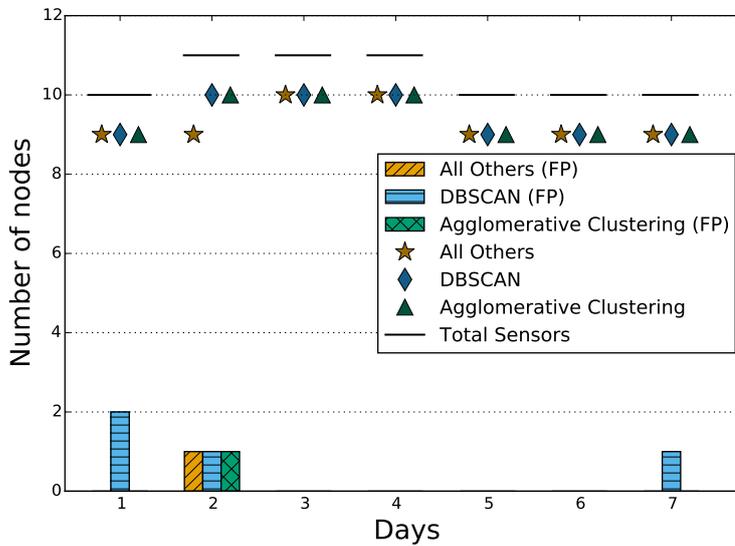


Figure 14: Effectiveness of different clustering algorithms with  $R = 0.0$  on accurately classifying sensors in the Salty dataset

**SUITABLE CLUSTERING ALGORITHM FOR SENSORRANKER** To investigate the effectiveness of different clustering algorithms in classifying sensors based on the *SensorRank* values that were generated by the SensorRanker mechanism, the evaluation was repeated on both datasets with the *responsiveness* threshold set to  $R \geq 0.0$ , i.e., bots that responded at least once during the monitoring period. In the evaluation, the performance of *K-Means*, *Gaussian Mixture Models*, and *SpectralClustering* were identical regarding the number of detected sensors and false positives. Therefore, these algorithms are grouped and referred to as *All Others* for simplicity. Figure 14 presents the performance of *DBSCAN*, *Agglomerative Clustering*, and *All Others* depending on the day of the measurement within the Salty dataset. The individual markers represent the number of accurately detected sensors, and the bar plots indicate the number of false positives generated by the algorithm for a given day. Evaluation results indicate that all algorithms except *DBSCAN* incurred exactly one false negative. *DBSCAN* incurred two false positives throughout the whole week.

Also, there were no false positives generated by all detection mechanisms between day three and six.

Upon further investigation, it is discovered that all algorithms missed a particular node that had a very low popularity. This node which had only one incoming connection, was a BT node that was deployed within Sality (see. Section 5.2.2). However, since the characteristics and nature of this BT node satisfy all assumptions of a sensor node (see Section 6.1.1.3), this node should have also been detected as a sensor. *DBSCAN* is observed to perform inferior to the other algorithms as it generated far more false positives compared to the other algorithms.

This evaluation is repeated on the ZeroAccess dataset and all algorithms were able to successfully detect three sensors throughout the week with no false positives or false negatives. After considering the analysis results of both datasets, the *Agglomerative Clustering* algorithm was decided to be used as the choice of clustering algorithm in SensorRanker for the rest of the analysis, including on the ZeroAccess dataset, due to its simplicity with regards to tuning parameters and improved performance particularly on Day 2 within the Sality dataset.

**INFLUENCE OF ARTIFACTS PRESENT WITHIN DATASETS** As artifacts could adversely affect the performance of the detection mechanisms, their influence was investigated with varying values of  $R$ , i.e., between 0.0 and 0.9, on all three detection mechanisms using both datasets. From the results that were aggregated over the whole week for the Sality dataset, very high number of false positives were observed for both SensorBuster and LCC when  $R = 0.0$ . Both mechanisms managed to detect all existing sensors, i.e., 11, at the expense of 43 and 70 false positives respectively. Since these extreme values of false positives distort the overall representation of the results, the analysis of the Sality dataset is presented in Figure 15 only for values of  $R$  between 0.1 and 0.9. This figure represents the number of nodes classified as *Sensor* by the respective detection mechanisms along with the corresponding false positives dependent on varying values of  $R$ .

The results of this analysis met the initial expectation presented in Section 6.3.3 that with increasing  $R$ , false positives can be reduced considerably. The observation of reduced false positives is particularly true for LCC that was able to reduce 90% of its false positives from the initial 70 to only seven false positives when  $R$  is set from 0.0 to 0.5. Although, similar observations were seen for SensorBuster, this was only true for values of  $R \leq 0.5$ . Interestingly, SensorRanker was found to be least affected by the different values of  $R$ . Although there was only one false positive for all values of  $R$  0.9 for SensorRanker, it also incurred one false negative. Nevertheless, take note that with a high threshold value of  $R \geq 0.8$ , some of the sensors are flagged as artifacts and ignored by the detection mechanisms in some of the days, e.g., Day 3.

Similar observations were also seen in the evaluation using the ZeroAccess dataset. In this evaluation, all detection mechanism were

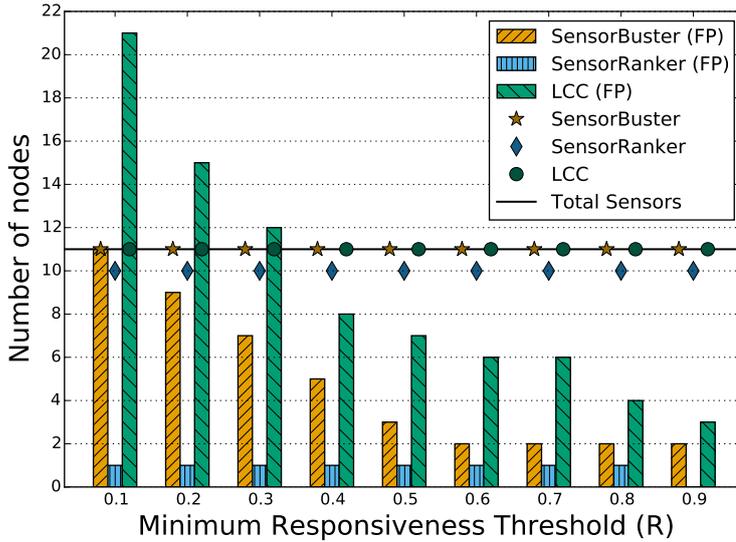


Figure 15: Analysis of the influence of artifacts to the detection mechanisms with varying values of  $R$  between 0.1 to 0.9 on the Sality dataset

able to detect all present sensors, i.e., three sensors. However, SensorRanker did not generate any false positives regardless of different values of  $R$ . Meanwhile, both SensorBuster and LCC eliminated their only false positive with threshold values of  $R \geq 0.1$  and  $R \geq 0.2$  respectively.

In conclusion, as presented in Table 14, only SensorRanker was found to be minimally affected (if any) by the different values of  $R$ . Hence, they would perform the same regardless of the presence of artifacts in the used dataset. Furthermore, it is also observed that LCC is heavily influenced by the presence of artifacts compared to the SensorBuster mechanism. As such, a minimum responsiveness threshold of  $R \geq 0.6$  is found to be a conservative value for all detection mechanisms without accidentally ignoring sensors with poor responsiveness within the Sality dataset. Similarly, a threshold of  $R \geq 0.2$  is observed to be appropriate for all detection mechanisms on the ZeroAccess dataset. The disparity between the threshold values of the Sality and ZeroAccess dataset can also be argued as directly influenced by the MM interval of the respective botnets. Sality, which has a longer MM interval compared to ZeroAccess, has a higher probability of introducing artifacts from churn affected nodes in their  $NLs$ . In contrast, the shorter MM interval of ZeroAccess reduces the probability for its bots to have unresponsive bots or artifacts present in their  $NLs$ .

#### PERFORMANCE COMPARISON OF ALL DETECTION MECHANISM

After obtaining the appropriate threshold parameters, a comparison analysis of the performance of all three detection mechanisms is performed on both datasets with responsiveness threshold of  $R = 0.6$  and  $R = 0.2$  respectively. Figure 16 presents the results on the Sality dataset whereby the accuracy of each detection mechanism in dependence on the day of the snapshot is plotted. Meanwhile, Table 14 provides a

brief summary of the performance comparison among the three detection mechanisms.

Table 14: Performance comparison of all three sensor detection mechanisms

	LCC	SensorRanker	SensorBuster
<b>True Positives</b>	High	Medium	High
<b>False Positives</b>	High	Low	Low
<b>Sensitivity to Artifacts</b>	High	Low	Medium

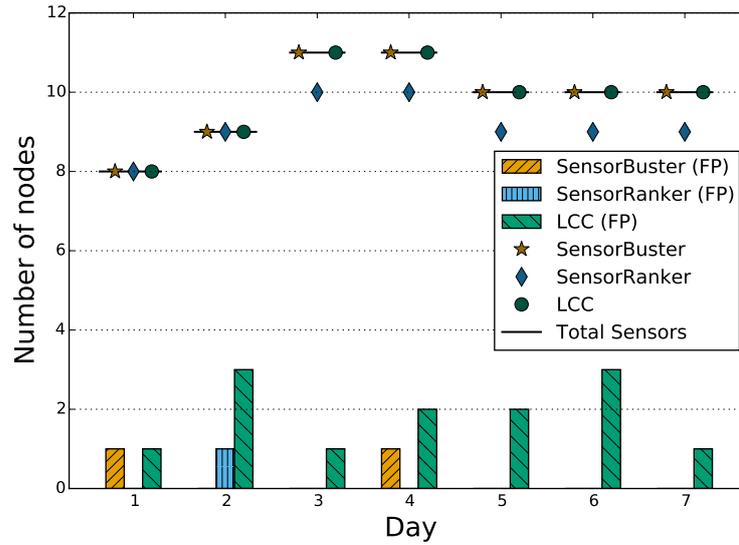


Figure 16: Performance comparison of all detection mechanism with R = 0.6 on the Sality dataset

Analysis results in Figure 16 indicate that both SensorRanker and LCC were able to detect all available sensors on each day, i.e., up to 11 sensors. SensorRanker, although had one false negative compared to the other mechanisms, has the least false positive throughout the whole week, i.e., one false positive. In comparison, LCC performs worst compared to all other mechanisms in terms of the number of produced false positives. Meanwhile, the analysis of the ZeroAccess dataset indicated that all algorithms were able to detect exactly three sensors with no false positives or false negatives.

**FEASIBILITY OF CIRCUMVENTING LCC** As discussed in Section 6.2.1, LCC can be easily circumvented using the DSI strategy with a minimum of four colluding sensor nodes. In fact, as long as the DSI strategy is able to obtain any non-extreme value, i.e.,  $lcc^+ \in \{0.0, 1.0\}$ , LCC can be circumvented. However, if the LCC-values of the sensors are significantly different than most of the bots in the botnet, clustering algorithms can be used to identify such anomalies in the values as used in the SensorRanker detection mechanism. Therefore, additional effort may be required to first identify the average value of the botnet, and the sensor nodes should try to approximate this value using the DSI strategy in order to avoid being detected through the us-

age of clustering algorithms in a more advanced LCC detection mechanism.

**FEASIBILITY OF EVADING SENSORRANKER** To investigate the feasibility of evading SensorRanker, an analysis was conducted to attempt to provide a lower bound estimation of the number of sensors needed to comfortably evade detection based on the Salty dataset. For this, the sensors are assumed to evade detection if their *SensorRank* value is lower than or equal to the average value of all bots. As such, the minimum number of sensors  $N_{min} = |S|$  that are required to satisfy Equation 5 needed to be calculated to evade the mechanism.

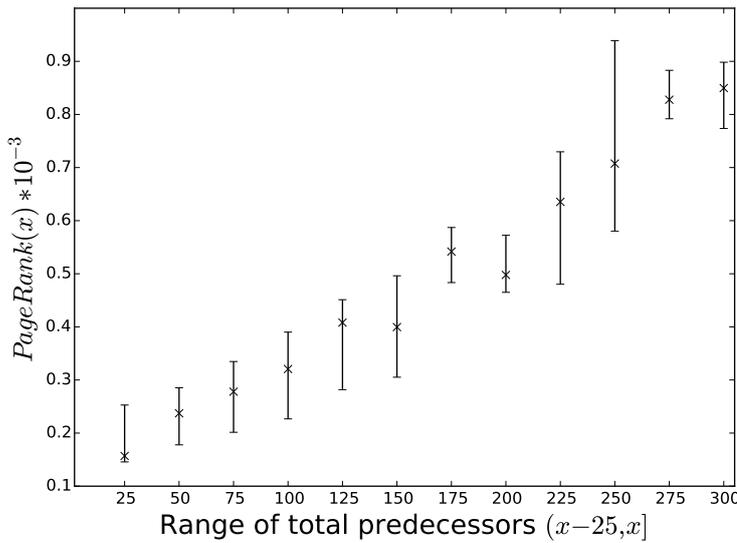


Figure 17: Classification of PR values by range of total predecessors in Day 1 of the Salty dataset

However, it is not easy to calculate the PR values of a sensor  $S_i$ , as it is not only influenced by its predecessors but also by the rank or popularity of all of its predecessors. As such, the distribution of existing PR values for bots on Day 1 in the Salty Dataset is referred. Figure 17 represents the *maximum*, *minimum* and *average* PR values with respect to the range of total predecessors in the investigated snapshot. As expected, the classification results indicate a linear increase in PR-values relative to increasing range of total predecessors, i.e., higher popularity yields higher ranks. Moreover, some ranges are also observed to have extreme PR values compared to their next ranges, e.g., comparison between the range of (225, 250] and (250, 275]. This behavior is due to one or more predecessors of some bots within the range (225, 250] having abnormally high rank, hence increasing the max PR of the bot(s) within this range compared to those in range (250, 275].

Based on the distribution of PR values in Figure 17, all possible SensorRank values is calculated by considering scenarios of deploying up to 20 colluding sensors within the Salty botnet as presented in Figure 18 with respect to the number of colluding sensors. The values plotted in this figure are the corresponding *maximum*, *minimum* and *average* SR values considering the possible range of total predecessors

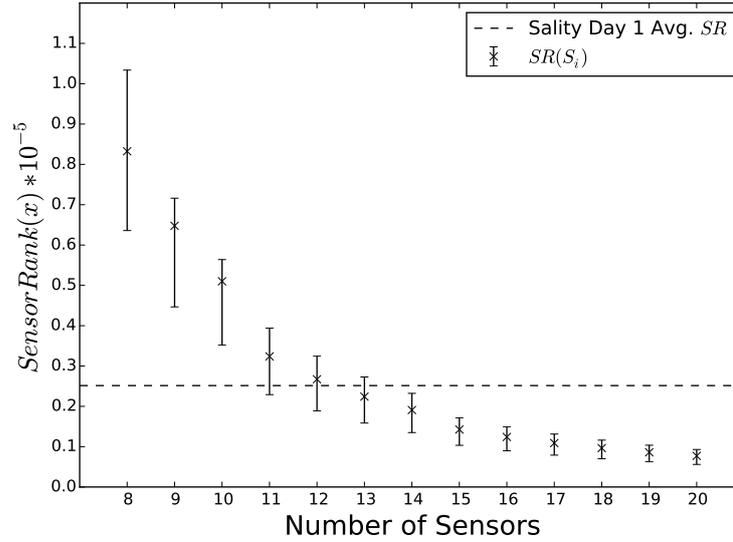


Figure 18: Estimation of *SensorRank* values for  $S_i$  in dependence on number of colluding sensors

each of the sensors would have. As the average SR value for the whole dataset is  $2.5145 \times 10^{-6}$ , the maximum SR value of each colluding sensor needed to be lesser than it. Therefore, at least 14 colluding sensors are needed to evade the *SensorRank* detection mechanism based on the utilized snapshot.

However, the estimation shown in Figure 18 does not resemble the worst case scenario, where one of the sensors has the  $\frac{|V|}{N}$  highest ranked nodes as predecessors. Hence, the estimation provides only a best-case scenario, but in reality requires additional efforts to distribute the PR values intelligently or more sensors to reliably evade this detection mechanism. In conclusion, *SensorRanker* can be circumvented provided sufficient resources be available at disposal to deploy additional sensors.

**FEASIBILITY OF CIRCUMVENTING SENSORBUSTER** As discussed in Section 6.2.3, to the best of knowledge, *SensorBuster* can only be circumvented if sensor nodes return valid bots when being requested for additional neighbors. Instead of returning any bot in the NL replies, a *less useful* bot is suggested to be returned instead, i.e., newly joined bots. From the perspective of a sensor node, this strategy incurs only minimal overhead to keep track of the timestamps of bots recently joined and known to the sensor. However, considering the fact that newly joined bots have the tendency to leave the botnet overlay immediately, the returned bot may not be responsive, i.e., offline, when a crawler captures the snapshot for *SensorBuster* analysis. As a consequence, the sensor nodes can once again be detected by *SensorBuster* due to the missing path into the main SCC of the botnet. Therefore, it is advisable to return more than one of such newly joined bots, i.e., redundancies, to ensure there is at least one path into the main SCC always.

## 6.4 CHAPTER SUMMARY

This chapter focused on advanced monitoring on the basis of using sensor nodes in P2P botnets and outlined another major contribution as part of this dissertation. In particular, contrary to the reports of other researchers, works presented in Section 6.1 clearly indicated that it is indeed possible to identify deployed sensor nodes in P2P botnets and distinguish them from popular backbone nodes. Evaluation results of the LCC, SensorRanker, and SensorBuster suggest that many existing sensor nodes are currently susceptible to the proposed detection mechanisms.

**DETECTING SENSOR NODES** In particular, LCC and SensorBuster are able to detect sensor nodes deployed in Sality and ZeroAccess. However, LCC is more prone to false positives resulting from artifacts present in the datasets than SensorBuster. Meanwhile, SensorRanker is able to perform reasonably well in detecting sensors and is only minimally influenced by artifacts, i.e., low number of false positives, within the evaluated datasets. Therefore, a future botmaster would potentially deploy the SensorBuster mechanism if he is concerned on detecting all sensors that are present in the botnet and is tolerant for some false positives. However, this mechanism requires additional *responsiveness* information of the bots in addition to the connectivity information. In the absence of such information or if accurately identifying sensors is the only concern, the botmaster should utilize SensorRank instead.

**CIRCUMVENTING THE DETECTION MECHANISMS** This chapter also introduced methods to circumvent the proposed detection mechanisms using a set of colluding sensor nodes, i.e., DSI. While it is relatively easy to circumvent LCC and SensorRanker, it is more complicated to circumvent SensorBuster. As discussed in Section 6.2.3, it requires a sensor node to return bots when being requested for neighbors to evade the detection of SensorBuster. As a result, such actions may have some legal implications as it would contradict with cyber-laws of many countries. In view of this, more work needs to be done to investigate the extent of which an organization or individual should be allowed to go in future botnet monitoring. This is especially important in anticipating future anti-monitoring countermeasures that could enforce strategies that require all bots, including sensor nodes or crawlers, to participate in regular botnet maintenance activities before they can retrieve any information, e.g., additional neighbors.

Next, the churn dynamics of Sality and ZeroAccess is investigated and a thorough churn analysis is presented in Chapter 7 as the final contribution of this dissertation. In order to obtain higher quality churn measurements, the SensorBuster mechanism introduced in this chapter is applied in the work to detect and remove sensor nodes from the final measurements.



## UNDERSTANDING THE CHURN DYNAMICS IN P2P BOTNETS

---

Churn dynamics in P2P botnets play an important role when it comes to understanding the botnet itself and sometimes also when preparing for a botnet takedown. Without understanding them, data gained by measurements may be misinterpreted and lead to false conclusions or assumptions. For instance, a botnet takedown attempt may require to target newly joining bots first. For that, the information on the rate of new bots joining the overlay will be helpful in allocating or predicting the required resources to launch such an attack.

Moreover, the understanding of the churn dynamics is also helpful in modelling churn using mathematical distributions [SR06] to be used in botnet simulations. Such churn models enable more realistic botnet simulations which in turn lead to a better understanding of the botnet phenomenon. However, capturing or characterizing *churn* is a difficult task due to the highly dynamic nature of P2P botnets as explained in Section 3.4.1. Existing works in characterizing churn in P2P botnets have only presented measurements with coarse granularity that may not be accurate enough for churn modelling purposes. Moreover, the presence of unknown third party monitoring activities may also introduce bias in the resulting monitoring data.

This chapter presents an analysis of the churn dynamics observed in Sality and ZeroAccess using a self-developed high-frequency crawler called *Strobo-Crawler*. Besides obtaining fine-grained measurements for the superpeers of both botnets, this crawler is also able to characterize churn of non-superpeers in ZeroAccess by leveraging the design of this botnet to perform UDP hole-punching (see Section 4.3.2). Unfortunately, it was not possible to characterize churn in GameOver Zeus since the bots have already been sinkholed in 2013 [Ros+13]. To ensure the churn measurements are not tainted by presence of sensor nodes in the botnets, sensor detection techniques as presented in Section 6.1 are also applied to identify and remove sensors from the resulting data.

The remainder of this chapter is organized as follows: First, Section 7.1 proposes the design of a high-frequency crawler called *Strobo-Crawler*. This section also includes a short description on how the crawler can be adapted to crawl Sality and ZeroAccess. Second, Section 7.2 presents the measurements and the derived churn model using *Strobo-Crawler*. Finally, Section 7.3 summarizes this chapter. Please take note that some passages in this chapter are quoted verbatim from the following publication [Haa+16].

## 7.1 ACCURATELY CAPTURING CHURN DYNAMICS IN P2P BOTNETS

Crawlers are commonly used as an important tool in characterizing churn in P2P botnets. Snapshots generated by crawlers (cf. Section 3.3.2) are used to discover bots and their interconnectivity. Furthermore, these snapshots help to assert the responsiveness of bots during the crawl period. However, a botnet overlay changes frequently with new bots joining and leaving the overlay throughout a measurement duration. Therefore, depending on the crawler configuration, e.g., crawling frequency or algorithm, resulting snapshots could be distorted due to a bias introduced by slow or inefficient crawlers, i.e., causing undiscovered or introducing non-existing nodes and edges in the snapshot. Correlating such distorted snapshots is not well suited to obtain fine-grained churn measurement. The impact of such distorted snapshots becomes more evident with presence of anti-monitoring mechanisms deployed by the bots. This requires an efficient monitoring solution that is fast enough for obtaining fine-grained churn measurements within large botnets.

In the following, Section 7.1.1 introduces a crawling framework called *Strobo-Crawler* that is specifically designed to capture snapshots of P2P botnets at a high frequency. These snapshots are then utilized to characterize churn in botnets. Section 7.1.2 and 7.1.3 presents adaptation of this crawler for *Salicy* and *ZeroAccess* respectively.

### 7.1.1 *Strobo-Crawler*

In this subsection, the botnet formal model proposed in Section 3.2 is first extended to describe *Strobo-Crawler*. After that, the design of *Strobo-Crawler* is elaborated using the extended botnet formal model.

**EXTENDED BOTNET FORMAL MODEL** The two major requirements for an accurate data collection for churn modeling are (1) enumerating all nodes and (2) reliably determining their online status. However, the exact set of all online bots  $V$  in a botnet  $G$  is often difficult to discover by crawling. Therefore, the set  $V$  is *approximated* by the set of *monitored* nodes  $V^M$ , that is the set of bots discovered by the crawler. This set consists of online bots as well as artifacts of bots that already went offline. For instance, if a neighbor  $v \in V$  of node  $u \in V$  goes offline,  $v$  is removed from  $u$ 's NL at least after the next  $MM$ -cycle of the bot. Until the information on the departure of  $v$  has propagated to all nodes,  $v$  is still known to other bots and thus this offline node is part of the real-world graph  $G$ , i.e.,  $v \in V$ . As  $V$  and thus  $V^M$  might contain offline nodes or artifacts, the set  $V^O \subseteq V^M$  is introduced, which contains all nodes that have been *verified* to be *online*.

**DESIGN OF STROBO-CRAWLER** *Strobo-Crawler* checks all nodes regularly for their online status. Using the extended formal model as presented above, the design of the *Strobo-Crawler* as depicted in Figure 19 is described in the following. The *Prober Module (PM)* main-

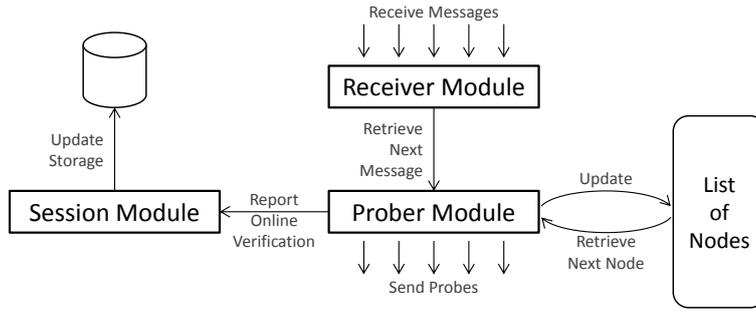


Figure 19: Strobe-Crawler Design Overview

tains the set of known nodes  $V^M$  (initially a bootstrap list) and probes each of them periodically at a fixed frequency  $f_c$  to determine their online status and to discover more nodes. Any incoming message  $m$ , including the response to a probe, is processed by the *Receiver Module (RM)* that validates the received message. It reports the sender  $v \in V$  of the message  $m$  and the shared nodes  $D_m \subseteq V$  included in  $m$  (if any) to the *PM*. After that,  $v$  and all entries within  $D_m$  can be merged with  $V^O$  and  $V^M$  respectively.  $v$  being the sender of the received message  $m$ , is merged into  $V^O$  as receiving the message also verifies the online status of  $v$ . The entries within  $D_m$  are merged with  $V^M$  because they are now discovered, but their online status has not been verified. Therefore, the entries that are not already in  $V^O$  are considered in the next probing iterations. Meanwhile, the *Session Module (SM)* summarizes the online duration of a node as a session that includes the first and last point of contact. The start time is set to the time when the presence of the node was verified, i.e., the time when a response message was received from the node. Similarly, the end time is set to the time of the last message received, i.e., the last communication before a node goes offline.

#### *Transitioning of Different Phases Within Strobe-Crawler*

The sequence diagram in Figure 20 depicts the three different phases within the Strobe-Crawler framework of a node  $v \in V$ . The initial phase starts with a previously unknown node, i.e.,  $v \notin V^M$ , that was newly discovered from a message  $m$ , i.e., the NL reply message. Now that  $v$  is known  $v \in V^M$ , the *PM* executes the *Session Initiation* phase and periodically probes  $v$ . As soon as one out of the first  $\eta$  requests sent to  $v$  is answered, a session is created and  $v$  is marked online, i.e.,  $v \in V^O$ . To compensate temporary network or host failures, each known node  $v \in V^M$  has a counter  $C_v \in \mathbb{N}_0$  that is increased with each probe message sent to  $v$ . The counter is reset upon every received reply from  $v$  and indicates a timeout if it reaches  $\eta$ .

A valid response received for a new session also creates a transition for the node to the *Session Looping* phase. In this phase,  $v$  is periodically probed and any subsequent valid response keeps its session active. When  $\eta$  consecutive requests are unanswered, the node  $v$  transits into the *Session Closure* phase. The node is considered to be

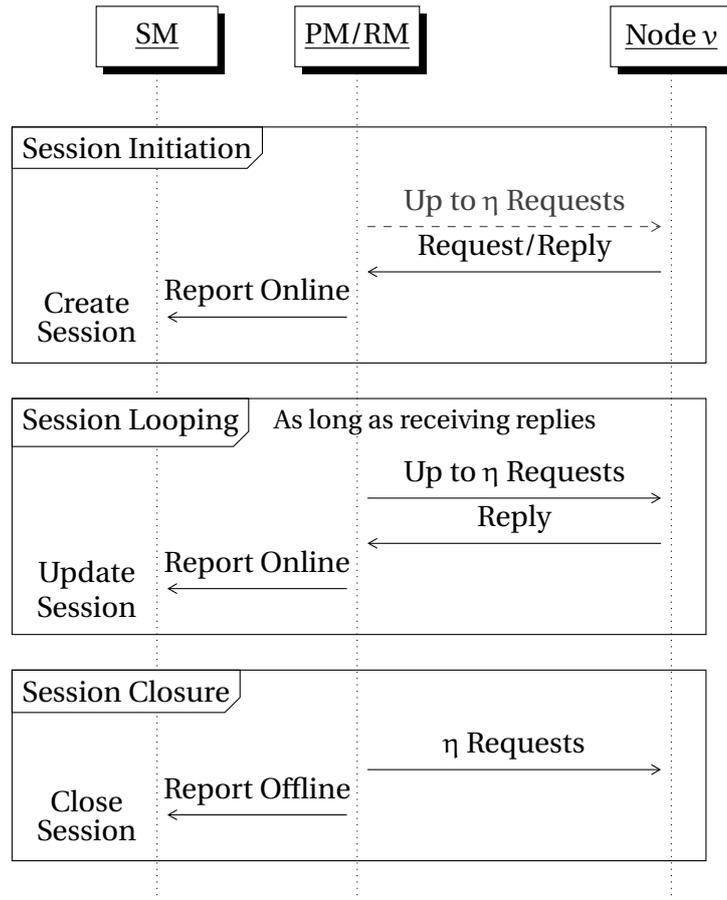


Figure 20: Sequence Diagram for the Strobo-Crawler Framework

offline, its session is closed, and  $v$  is removed from the set of known nodes  $V^M$  to end probing.

#### Steps in Handling Sessions

To ensure proper session handling, the events representing either an *incoming message*, *shared node*, or *outgoing probe* have to be scheduled for processing at the time of reception, sharing, or sending respectively. Step by step, each scheduled event  $e$  is processed by Algorithm 7 appropriately. This algorithm requires three input parameters depending on the event that is taking place: the node  $v$ , event type ID  $e$ , and the event's time  $t$ . In addition, two additional measurement-specific parameters can also be used to fine tune the crawler: crawling frequency  $f_c$  and maximum retries value  $\eta$ . These parameters can be used either to evade botnet rate-limiting countermeasures (see Section 3.4.3) or to throttle the probing rate of the bots. The latter is usually done to select the suitable probing frequency for a particular botnet, and to reduce the generated communication overhead. The crawling frequency  $f_c$  influences the accuracy of the measured start and end time of a session. Meanwhile,  $\eta$  is used to tolerate temporary network problems like jitter and packet loss in the process of verifying a node's online status. The counter  $C_v$  for each node  $v \in V^M$  provides

a mechanism to deploy a session timeout that is effectively defined by  $\delta = \frac{\eta}{f_c}$ . Algorithm 7 is detailed in the following.

---

**Algorithm 7 :** processEvent( $v, e, t, f_c, \eta$ )

---

```

1 // node  $v$ , event  $e$ , time  $t$ , freq.  $f_c$ , & max. retries  $\eta$ 
2 if  $v \in V^M$  then
3    $V^M = V^M + v$ 
4   probeNode( $v$ )
5    $C_v = 1$ 
6   scheduleEvent( $v, outgoing\ probe, t + 1/f_c$ )
7 switch  $e$  do
8   case outgoing probe do
9     if  $C_v = \eta$  then
10       $V^O = V^O - v$ 
11       $V^M = V^M - v$ 
12     else
13      probeNode( $v$ )
14       $C_v = C_v + 1$ 
15      scheduleEvent( $v, outgoing\ probe, t + 1/f_c$ )
16   case incoming message do
17       $C_v = 0$ 
18       $V^O = V^O + v$ 
19   case shared node do
20     if  $v \in V^O$  then
21        $C_v = 0$ 

```

---

First, if the corresponding node  $v$  was previously unknown (Line 2), it is immediately probed and the *Session Initiation* phase is initiated with a counter value of 1. The next probing message is also scheduled to be sent according to the crawling frequency  $f_c$ . In contrast, if  $v$  is already known, further processing of  $e$  depends on the type of the event (Line 7).

For scheduled probing events (Line 8), a probe is sent to  $v$  (Line 12) and the next probe is scheduled, or a timeout is reached if  $\eta$  requests have been sent without receiving any valid reply (Line 9). In case the event represents an incoming response message (Line 16), the node's counter is reset and the node is marked online. Meanwhile, shared nodes are continued to be probed as long as they are shared by other bots, even if they have not been verified online. However, the counter for shared nodes (Line 19) is only reset, if the node is currently not marked online to avoid manipulation of the session timeout mechanism during the *Session Looping*.

### 7.1.2 Adaptation for Salinity

The Strobe-Crawler framework can be adapted to measure *only* the arrival and departure of superpeers in the Salinity botnet. Following the paradigm of a crawler (cf. Section 3.3.2), this framework allows to actively contact and discover nodes. The non-superpeers are omitted

in the measurements since they are not directly contactable by the Strobe-Crawler (see Section 2.2), i.e., devices behind NAT such as notebooks, residential machines, and smartphones. Moreover, using sensor nodes for monitoring only provides coarse-grained measurements. Sensor nodes are only able to obtain measurements that corresponds to the MM interval of the bots, i.e., interval of approximately 40 minutes (cf. Section 4.2). Hence, only superpeers are considered in the fine-grained churn measurements for Sality.

The *PM* is implemented to send NL requests  $NL_{Req}$  (cf. Section 4.2) to verify a superpeer's online status and to retrieve shared nodes at the same time. Although bots in Sality send a *Hello* message before sending a  $NL_{Req}$  message, this can be omitted for doing churn measurements and does not have any negative side-effects. In fact, omitting the *Hello* messages minimizes the overhead of generated messages by the Strobe-Crawler.

### 7.1.3 Adaptation for ZeroAccess

In contrast to Sality, Strobe-Crawler can be adapted for ZeroAccess to characterize both superpeer and non-superpeers. Since ZeroAccess uses dedicated sockets for its communication, i.e., server and client sockets, and processes messages received on both sockets in the same way (cf. Section 4.3.2), UDP hole-punching [FSK05] can be performed to crawl bots behind NAT as reported by Rossow et al. [Ros+13]. In ZeroAccess, UDP hole-punching can be performed by injecting a sensor into the botnet (cf. Section 3.3.3) to allow all nodes (including non-superpeers) to contact it. Once a bot behind NAT contacts the sensor, the responsiveness of this bot can be verified as long as the NAT tunnel remains open. For this, messages need to be sent periodically to ensure the established tunnel does not time out.

The design flaw of the malware is exploited by Strobe-Crawler to crawl all ZeroAccess bots. The *RM* is extended with a sensor functionality to additionally process incoming requests from non-superpeers. This corresponds to the *Session Initiation* in Figure 20 without the optional request message. The *RM* reports the sender  $v \in V$  of a request message  $m$  to the *PM*, so that the node transits into the *Session Looping* phase. Having both crawler and sensor functionality, this new design allows the Strobe-Crawler to actively discover new superpeers by requesting nodes to share neighbors and to passively discover new non-superpeers by processing incoming requests. However, the measurement technique of Strobe-Crawler for ZeroAccess could also initiate two distinct measurement sessions for superpeers in the botnet, i.e., one session for messages received from the server socket and another for the client socket. To avoid such duplicated sessions, whenever such a scenario occurs, Strobe-Crawler prioritizes the server sockets, i.e., identified by the botnet-specific source ports, and merges the sessions originating from the client socket. From that point onward, only the session associated with the server socket is maintained.

To ensure that non-superpeers in ZeroAccess are discovered and probed, Strobe-Crawler leverages the node announcement mechanism of ZeroAccess by sending *getL* messages to superpeers and respond to the subsequently received *getL+* messages. By responding to the *getL+* message, Strobe-Crawler gets inserted or shifted to the top of the bots' NLs (see Section 4.3.2.1) and will be frequently included in the response messages to other requesting non-superpeers. As a consequence, Strobe-Crawler is able to discover new bots quickly. Regarding non-superpeers, Strobe-Crawler only sends *getL+* messages to probe the responsiveness of non-superpeers. This is done to avoid generating additional overhead during the measurements. Moreover, it is not important to stay at the top of the primary NL of a non-superpeer as the NL is not shared to other bots, i.e., only superpeers play a role in propagating information of a node to other bots.

## 7.2 EVALUATION

This section presents the measurement results and the derived churn model from the analysis conducted in Section 7.1. Section 7.2.1 describes the obtained datasets. Section 7.2.2 elaborates the setup for the measurements and analysis. Then, Section 7.2.3 discusses the investigated research questions as well as the expected outcomes. Finally, Section 7.2.4 presents the measurement results and the derived churn model of the analyzed botnets.

### 7.2.1 Datasets

Table 15: Dataset Statistics

Botnet	Start (CET)	Unique Addrs	Sessions	Superpeers
ZA_16464	10.12.2015, 6pm	5,343 / h	611,134	5.9%
ZA_16465	11.12.2015, 5pm	1,454 / h	349,401	2.3%
ZA_16470	11.12.2015, 9pm	2,172 / h	560,700	1.9%
ZA_16471	13.12.2015, 7pm	2,535 / h	350,963	6.4%
Sality_V3	09.10.2015, 12am	1,581 / h	53,147	100%
Sality_V4	22.12.2015, 7am	189 / h	17,952	100%

The datasets used for deriving the churn models have been obtained using the Strobe-Crawler framework as described in Section 7.1.1. Both Sality and ZeroAccess were crawled for a duration of two weeks. The summary of the dataset is presented in Table 15.

Sality\_V*N* denotes the specific version *N* of Sality and ZA\_*N* identifies the specific ZeroAccess network with the fixed port number *N*. The table presents the average number of unique bots seen per hour, i.e. combination of IP address and port number, as well as the total number of observed sessions during the measurements. In addition, the percentage of unique superpeers that were observed within the measurements are also depicted in the last column. Please note

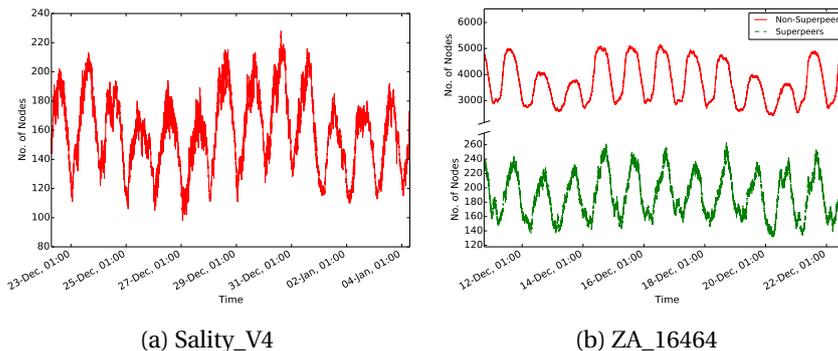


Figure 21: Distribution of online bots discovered during the measurement

that only superpeers were available for characterizing churn in Sality, hence the 100% for the right-most column in the table. Figure 21a and Figure 21b depict the distribution of online nodes discovered and probed by St robo-Crawler in Sality\_V4 and ZA\_16464 respectively in dependence on the time of discovery.

### 7.2.2 Experimental Setup

The experiments were conducted by an implementation of St robo-Crawler in *Python*. All communication to and from St robo-Crawler was logged using a *MySQL* database. To ensure heavy network traffic does not influence the ongoing measurements, each botnet was assigned a dedicated St robo-Crawler, i.e., a machine, that was each located in the network of different ISPs.

A crawling frequency of 30s was chosen as a conservative value to provide fine-grained churn measurements for both botnets. Although the frequency can be further increased, it will incur significantly higher communication overhead for the St robo-Crawler. The author argues that a frequency of 30s is already sufficient for the purpose of fine-grained churn measurements. However, due to the restricted NL return size in Sality (cf. Section 4.2.2) five requests were sent every 30s to ensure new bots are quickly discovered by the crawler. Moreover, to take network jitter and packet loss into consideration, a timeout was assumed for a session if no valid response received within  $\mu = 120$ s. This timeout also corresponds to effectively  $\eta = 24$  unresponsive sent requests for Sality and  $\eta = 4$  for ZeroAccess. Then, a two-week long measurement was obtained based on the specified parameters using St robo-Crawler.

### 7.2.3 Research Questions and Expectations

In the following, research questions that were focused on the characterization of churn in P2P botnets as well as the influence of unknown third party monitoring is presented along with investigated parameters.

Considering there have been quite a number of churn studies conducted on regular P2P networks, e.g., file sharing networks, it is interesting to see if the observed characteristics in those networks are identical to that of P2P botnets. Hence, the following research question needs to be answered in the evaluation:

- *What are the similarities in churn characteristics of P2P botnets compared to conventional P2P file sharing networks?*

For that reason, this work uses the churn-related metrics presented by Stutzbach and Rejaie [SR06]. Although the authors listed metrics for the two types of classifications mentioned above: *group-level* and *peer-level*, only group-level characterizations are observed in this work. This is due to the reasons explained before, more precisely to the absence of reboot-persistent botnet UIDs that could otherwise be used to reliably characterize the peer-level characteristics. It is expected that the churn behavior of P2P botnets and regular file sharing networks are similar due to the presence of diurnal patterns, the fact that both run on end-user machines, and the usage of common network infrastructures that result to similar issues like address aliasing.

It is not always possible to conduct experiments or analysis in real-world botnets. As alternative, simulations can be used to provide a controlled environment to investigate any aspect of botnets. For that, realistic simulations with accurate churn models are required. Stutzbach and Rejaie have reported that churn behavior in P2P networks can be modeled using a Weibull distribution [SR06]. However, it is unknown if the observation is also valid for P2P botnets. Therefore, the following research question needs to be answered in the evaluation:

- *Is Weibull distribution applicable to model churn in P2P botnets? If yes, what are the parameters to model them?*

To answer this question, a parametric distribution-fitting analysis of the churn measurements needs to be done to fit the observed measurements with a Weibull distribution.

Next, the presence of highly responsive sensors in P2P botnets could influence the final churn measurements in P2P botnets. Hence, the following research question needs to be answered in the evaluation:

- *Does the presence of highly responsive sensors affect the churn characteristics of P2P botnets?*

To answer this question, sensor detection mechanisms as presented in Section 6.1 needs to be applied on the obtained snapshots of Strobe-Crawler. Then, an analysis is conducted to identify if removing sessions of the detected sensors changes the measurement results significantly. It is expected that highly responsive sensors bias the measurements, in terms of average session length especially.

#### 7.2.4 Results

First, the churn measurements of Sality and ZeroAccess are presented in Section 7.2.4.1. Second, a churn model is derived and presented in

Section 7.2.4.2 based on a Weibull distribution fitting using the obtained measurements.

#### 7.2.4.1 Churn Measurements

In the following, churn metrics presented in Section 2.3 are used to characterize churn in the two botnets.

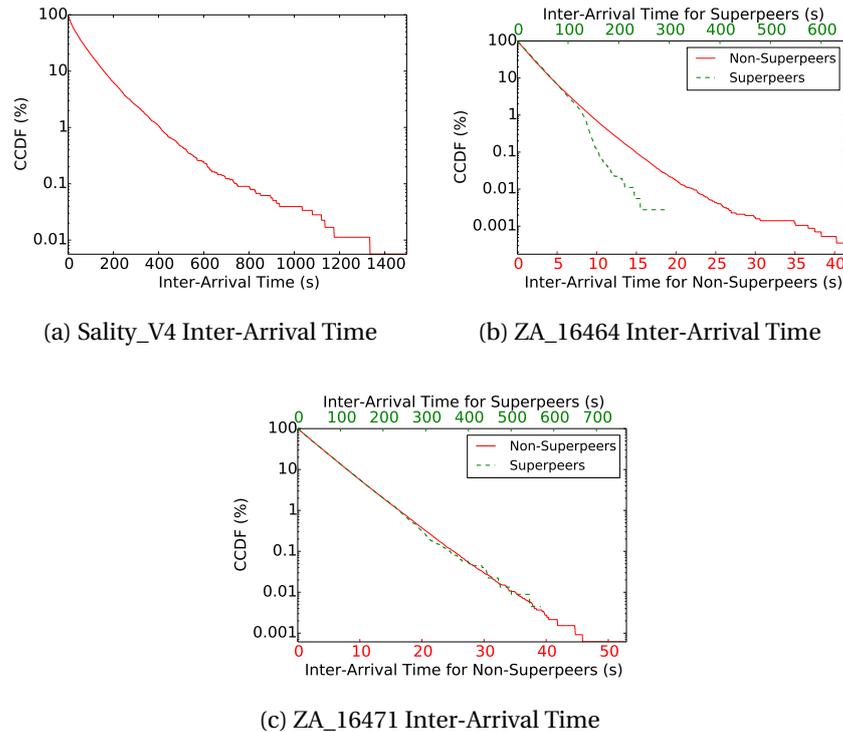


Figure 22: Churn distributions for Sality and ZeroAccess

**INTER-ARRIVAL TIME** To obtain the inter-arrival time distribution, the interval between two nodes joining the network is calculated. The distributions of these inter-arrival times in Sality\_V4, ZA\_16464 and ZA\_16471 are plotted in Figure 22a–22c as a Complementary Cumulative Distribution Function (CCDF) in log-linear scale. Please note that in the ZeroAccess botnets, both superpeers and non-superpeers are distinguished using two different x-axes, i.e., in green (top) and red (bottom), due to the differences in the scale/magnitude of the results. There is a common pattern in all botnets when looking at the distribution of inter-arrival times. However, the differences in the results of superpeers and non-superpeers in ZeroAccess are particularly interesting. The overall results for ZeroAccess indicate that non-superpeers have significantly shorter inter-arrival times than superpeers in ZeroAccess.

There are two possible explanations for these results. First, the set of superpeers is much smaller than the set of non-superpeers. Second, non-superpeers experience heavier node churn than superpeers. However, as depicted in Figure 22c, both types of nodes in ZA\_16471 share a similar shape of the graph and differ mainly on the x-scaling.

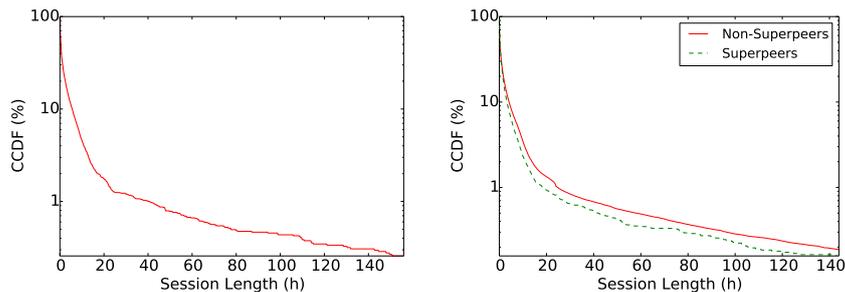
This further strengthens the argumentation that the differences between the observed distributions comes from different number of bots in between the two different classes of bots, i.e., superpeers and non-superpeers.

However, other ZeroAccess botnets exhibit different distribution shapes in comparison to ZA\_16471, e.g., ZA\_16464 in Figure 22b. In particular, several extreme long inter-arrival times in the ZA\_16470 botnet were observed for the non-superpeers but not for superpeers. As the initial argument on the non-proportional size in between the two different bot classes cannot explain the differences between these distributions, the joining behavior of bots in ZA\_16470 is concluded to be non-comparable to the other botnets for unknown reasons. However, the author acknowledges that this observation could also be possible due to the different node types in the botnets, i.e., router/server vs Smartphone/Desktops. Infected machines that reside in production networks may have shorter inter-arrival times due to the reliability of the network operators. In contrast, machines that are switching connection between residential and public networks may have significantly longer inter-arrival times.

In summary, there seems to be no perfect correlation of the inter-arrival behavior for superpeers and non-superpeers within the same botnet. However, the biggest differences in the distributions between both types of nodes (in log-linear transformation of the CCDF) are usually only caused by the longest 1% of recorded inter-arrival times. Although this percentage may seem to be insignificant, the longest records are important for extrapolating data such as sub-exponential distribution. Furthermore, no correlation is observed between superpeers in Sality and ZeroAccess that would allow a generalization for superpeers across different botnets. However, all distributions seem to have a higher tendency for smaller inter-arrival times.

**SESSION LENGTH** Any long-lived session in the measurements could have been artificially limited or prematurely terminated by the end of the measurement, i.e., after two weeks ( $\tau$ ). This problem can have high impact on the session length distribution because longer sessions are rare as reported by Stutzbach et al. [SR06]. Their approach to this problem divides the whole time window into two halves and to evaluate the sessions of the first half only. This approach is adopted in this work and for the remainder of the evaluation, the first half is referred to as the *analysis window*. All sessions that have a length exceeding  $\frac{\tau}{2}$  are pruned and set to a maximum length of  $\frac{\tau}{2}$ , i.e., one week, for all analysis (unless mentioned otherwise) to use a fraction of the time interval as an analysis window. This allows for a non-distorted distribution fitting as suggested by Stutzbach et al..

The session length of bots can be analyzed from several perspectives. First, the *joint session length* estimates the session length for the next node joining the network. Second, the *current session length* refers to only active sessions at any point in time and describes how the session length is distributed among online nodes. There is also a third perspective on the session length where peer-level measure-



(a) Sality\_V4 Joint Session Length

(b) ZA\_16464 Joint Session Length

Figure 23: Joint session length distributions for Sality and ZeroAccess

ments (see Section 2.3) can be considered using reboot-persistent identifiers (if available). In such a peer-level measurement, it would be possible to calculate the session length distribution for multiple sessions of the same peer. Yan et al. [Yan+14b], who conducted a churn measurement on Sality and ZeroAccess for more than 6 months by doing such peer-level measurements. However, since there are no reliable botnet UIDs, it is not clear how the authors could have conducted this measurement except by associating each unique IP address to be a unique bot. The peer-level measurement is omitted in this work as such IP-based association can lead to high bias due to IP address aliasing. In addition, results reported by Yan et al. indicate that no sessions have been observed with a length longer than a day, i.e., 24 hours. In contrast, measurements conducted in the context of this work have observed some sessions that are longer than seven days.

In the following, the measurement results for *joint session length* and *current session length* from the perspective of the group-level characterization is presented.

#### *Joint Session Length*

Figure 23 presents the distribution of the joint session length as CCDF in log-linear plot for Sality\_V4 and ZA\_16464 respectively. Plots for the other networks are omitted as the results are comparable to those presented in Figure 23. Specifically, the comparison of distributions in between the two bot classes in all ZeroAccess botnets indicated that they are very much similar. They vary from superpeers with session length slightly shifted towards smaller ones (ZA\_16464 in Figure 23b) to superpeers with longer session length (ZA\_16465 and ZA\_16470). Only in one network (ZA\_16471), both node types have a similar distribution.

Especially the graph for ZeroAccess non-superpeers (but also for the Sality superpeers), shows that there is a high number of sessions that are closed after around 24 hours. It can only be speculated that this is caused by the daily re-establishing of Internet connections that ISPs enforce in their networks.

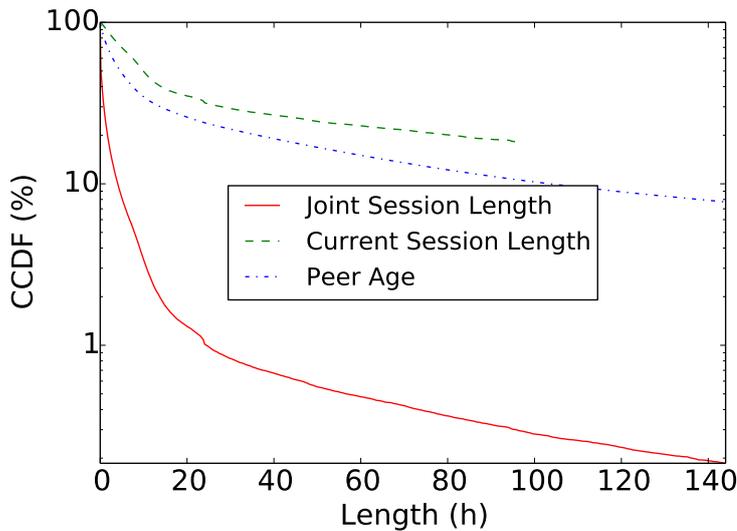
*Current Session Length*

Figure 24: Session Length Distributions of ZA\_16464

*Current Session Length*

The distribution of the current session length is obtained by summarizing the length of all active sessions in a given snapshot. However, it is necessary to identify the actual start time of a session as a session might have started earlier than the start of the measurements. To overcome this problem, the time window is divided into three sub-windows and the second window is chosen as the analysis window. This allows to identify sessions that are already active in the first  $\frac{\tau}{3}$  window or remain active after the second window. Hence, the maximum current session length is restricted to  $\frac{\tau}{3}$ .

Yao et al. [Yao+06] report that the influence of long sessions on the *current session length* is much more significant than the influence on the *joint session length*. In other words, it is more likely to observe a long session when selecting any *active* session than when selecting a *recorded* session. This is also true for the analysis presented in Figure 24 that illustrates the differences between the distributions of the joint and the current session length using ZA\_16464 as an exemplary network. Here, the probability that the current session length is four days or more is 19%. The probability for a joint session length of at least the same duration is only 0.3%. Please note that the longest current session length observed is limited to  $\frac{\tau}{3}$  and therefore shorter than the longest observed joint session length. Further when looking at the distributions for superpeers and non-superpeers, it reveals that if the joint session length distribution tends to observe superpeers exhibiting longer sessions, then the current session length distribution also shows the same trend.

**PEER AGE / REMAINING UPTIME** As for the calculation of the peer age and remaining uptime, different parts of the analysis window have to be analyzed. The overlapping sessions in the first half of the time window need to be analyzed to obtain the remaining uptime and overlapping sessions within the second half of the time window is needed to obtain the peer age. For both metrics, the age, and the remaining uptime respectively, is limited to  $\frac{\tau}{2}$ . The comparison of both metrics indicates that their distributions are similar within the same network, except for small differences that could be attributed to measurement in precision. However, Figure 24 presents the current peer age distribution that is also representative for the remaining uptime that seems to be close to the current session length time distribution. Please take note of the logarithmic CCDF scale (y-axis) that scales the values accordingly.

The shape of the distributions (group-level parameters) for the analyzed botnets are found to be similar to that of regular P2P file-sharing networks as reported by Stutzbach and Rejaie. Therefore, churn characteristics of both P2P botnets and regular P2P file-sharing networks can be concluded to be comparable to each other and meet the initial expectation. The similarities for both networks can be further speculated to result from similar set of users, i.e., regular home and office PCs, and their distribution throughout the globe, i.e., in regards with influence of diurnal effects.

#### 7.2.4.2 Modelling Churn With Weibull Distribution

On the basis of the obtained measurement results, the churn metrics for each botnet were found to be able to be modeled using a Weibull distribution (see Section 2.3) as proposed by Stutzbach and Rejaie for P2P file-sharing networks. The distribution fitting results of the measurements for the respective churn metrics are presented in Table 16 and Figure 25. The results of distribution fitting was done using the nonlinear least-squares estimation for the Weibull parameters *shape*  $k$  and *scale*  $\frac{1}{\alpha}$  in a log-linear scale. The time unit for all metrics is represented in seconds. Similarly, the CCDF plots for the metrics in Figure 25 is presented in a log-linear scale for clarity.

For the *inter-arrival time* metric, the shape parameter  $k$  of the Weibull distribution indicates that in some botnets, e.g., ZA\_16465 and ZA\_16471, the distributions for superpeers and non-superpeers can be transformed by a scaling factor for the inter-arrival times. In these cases,  $k$  is close to 1, which results in an almost straight line in the log-linear plot and allows the inter-arrival time to be approximated by an exponential distribution. The Weibull distributions for inter-arrival time in different botnets can have different shape parameters. The fitting results partially show some similarities with those presented by Stutzbach and Rejaie. The shape parameter of  $k = 0.62$  for their measurement in the *FlatOut* network is similar to the results for Sality superpeers and for all peers in ZA\_16464 and ZA\_16471. The Weibull distribution models the inter-arrival time very tightly for both Sality networks, and for the ZeroAccess networks ZA\_16465 and ZA\_16471. Al-

Table 16: Weibull Parameters; Tuples as (shape, 1/scale)

Node Type:	All	Non-Superpeers	Superpeers
ZeroAccess 16464			
Inter-Arrival	(0.65, 1.35e+00)	(0.66, 1.23e+00)	(1.01, 3.88e-02)
Joint Length	(0.22, 8.41e-03)	(0.22, 8.10e-03)	(0.21, 1.81e-02)
Current Length	(0.37, 1.31e-05)	(0.38, 1.30e-05)	(0.32, 1.53e-05)
Peer Age	(0.35, 2.72e-05)	(0.35, 2.70e-05)	(0.35, 3.00e-05)
ZeroAccess 16465			
Inter-Arrival	(1.04, 2.63e-01)	(0.97, 2.90e-01)	(1.02, 6.24e-03)
Joint Length	(0.18, 8.19e-02)	(0.19, 7.51e-02)	(0.17, 5.91e-02)
Current Length	(0.34, 1.39e-05)	(0.34, 1.52e-05)	(0.33, 2.78e-06)
Peer Age	(0.32, 3.03e-05)	(0.34, 3.20e-05)	(0.26, 5.46e-06)
ZeroAccess 16470			
Inter-Arrival	(0.63, 1.47e+00)	(0.64, 1.40e+00)	(0.84, 1.08e-02)
Joint Length	(0.19, 7.53e-02)	(0.19, 7.29e-02)	(0.20, 1.49e-02)
Current Length	(0.36, 1.40e-05)	(0.37, 1.47e-05)	(0.33, 5.91e-06)
Peer Age	(0.35, 3.38e-05)	(0.36, 3.43e-05)	(0.27, 1.65e-05)
ZeroAccess 16471			
Inter-Arrival	(0.95, 3.25e-01)	(0.87, 3.63e-01)	(0.91, 2.28e-02)
Joint Length	(0.21, 1.30e-02)	(0.21, 1.29e-02)	(0.20, 1.58e-02)
Current Length	(0.39, 7.16e-06)	(0.40, 7.05e-06)	(0.34, 8.96e-06)
Peer Age	(0.37, 1.66e-05)	(0.37, 1.61e-05)	(0.26, 2.70e-05)
Sality Version 3			
Inter-Arrival	-	-	(0.66, 1.72e-01)
Joint Length	-	-	(0.28, 8.38e-04)
Current Length	-	-	(0.47, 8.94e-06)
Peer Age	-	-	(0.32, 2.20e-05)
Sality Version 4			
Inter-Arrival	-	-	(0.61, 2.97e-02)
Joint Length	-	-	(0.22, 5.58e-03)
Current Length	-	-	(0.39, 9.85e-06)
Peer Age	-	-	(0.37, 1.88e-05)

though ZA\_16464 can be modeled with Weibull, modeling for ZA\_16470 is not successful due to several extreme long inter-arrival time measurements for the non-superpeers.

Most of the botnets also share a common shape parameter of  $k \approx 0.2$  for the *joint session length* distribution. This means that they differ only in their length scale. However, the botnet fitting results differ from the results of regular P2P file-sharing networks reported in

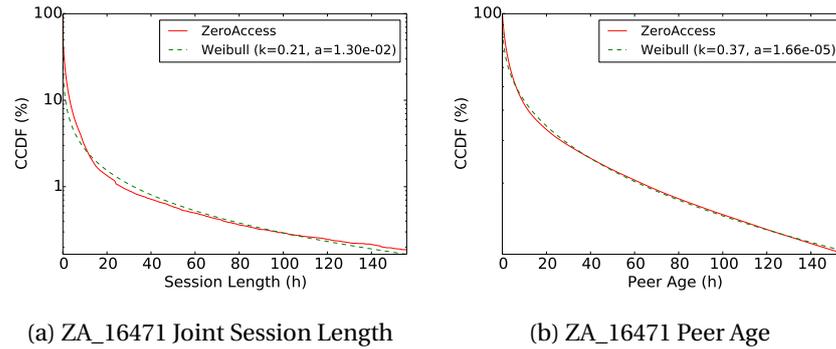


Figure 25: Distribution Fitting in ZA\_16471

[SR06], where the scale factor is between 0.34 and 0.59. Figure 25a presents a comparison of the empirical data of joint session length for all peers in ZA\_16471 with the estimated Weibull distribution. The probability distribution underestimates the majority of all lengths (here in 93% of the cases), which is consistently seen for all networks within Sality and ZeroAccess. This observation also applies to the *current session length* but the model still provides a decent fit for both metrics. The Weibull distributions for the peer age and remaining uptime are able to model the empirical data very tightly as depicted in Figure 25b, which is an example for all peers in ZA\_16471 but representative for all networks of the two botnets.

The combination of the different Weibull parameters presented in Table 16 can be used to apply accurate churn models in simulations. A user can not only choose to apply a generic churn model for both types of nodes, i.e., superpeers and non-superpeers, but also apply specific churn models for each type of nodes (if applicable). Accurate churn models enables a more realistic scenario for analysis and investigations of existing and future botnets. Moreover, since it is often difficult to conduct live botnet investigations, a simulation-based study using realistic churn models could be an alternative to understand and tackle the botnet phenomenon.

#### 7.2.4.3 Influence of Third Party Monitoring Activities

Sensor nodes that may have been present in the datasets could have influenced the resulting measurements or the derived churn models. Therefore the SensorBuster mechanism, which was presented in Section 6.1.4, is applied to detect and remove the sensors from the datasets. After removing the sensors, the analysis performed in Section 7.2.4.1 and 7.2.4.2 were repeated to compare the results of with and without the sensors in the datasets. The analysis results indicated that although there were a small number of detected sensors, i.e., less than ten sensors, in each botnet, the removal of the sensors were not influential enough to distort the overall churn measurements of the botnets. The results in the form of table and plots were omitted from being presented in this dissertation due to their insignificance.

The results from this analysis did not meet the initial expectation on the influence of third party monitoring as discussed in Section 7.2.3. This can be attributed by the low number of sensor nodes in comparison to the total number of available bots. Therefore, it can be concluded that churn measurements obtained in current Sality and ZeroAccess botnets, can ignore sensor nodes present within botnets.

### 7.3 CHAPTER SUMMARY

This chapter presented churn measurements and a representative churn model for two real-world botnets: Sality and ZeroAccess. In contrast to churn studies conducted by others, the presented work uses high-frequency crawling to obtain fine-grained measurements of the botnets. Such high-frequency crawling can reduce the amount of network bias or noise introduced in the final measurements in comparison to slower crawlers (see Section 3.4.1). In addition, this work also takes into consideration the influence of third party monitoring activities, i.e., sensors, on the resulting measurements through the application of the SensorBuster mechanism as proposed in Section 6.1.4. However, sensor nodes detected in the existing botnets are found to have no influence towards the overall botnet churn characteristics due to the small number of total sensor nodes currently deployed in the botnets.

From the evaluation results presented in Section 7.2.4, the research questions posed in Section 7.2.3 can be answered.

- *What are the similarities in churn characteristics of P2P botnets compared to conventional P2P file sharing networks?* Most of the churn measurements of Sality and ZeroAccess are observed to be similar in pattern across the duration of the measurement. The shape of their distributions also corresponds to those reported by Stutzbach and Rejaie [SR06] for regular P2P file-sharing networks. The similarities across these two different P2P domains is not surprising, considering that users of both domains usually experience similar IP address aliasing issues and diurnal behavior.

However, if a botnet consists of an even more diverse or mobility-enabled set of users, e.g., smartphone devices or Internet-enabled personal gadgets, the similarities with regular P2P file-sharing networks may need to be analyzed again. This is mainly because the botnets analyzed in this work (see Section 4) are more static when deployed in residential and production networks. However, with an increasing number of mobile and wearable devices in botnets, node churn may even be heavier. This in turn may cause the churn characteristics of such botnets not to be similar anymore with regular P2P file-sharing networks.

- *Is the Weibull distribution applicable to model churn in P2P botnets? If yes, what are the parameters to model them?* To answer this research question, the churn measurements of the two ana-

lyzed botnets were fitted with a Weibull distribution. The premise for this decision is the fact that the shape of the distribution of the botnet churn measurements is similar to that of regular P2P file-sharing networks [SR06]. Moreover, the Stutzbach and Rejaie reported that churn measurements of P2P networks are more accurately modeled by a Weibull distribution than other distributions such as heavy-tailed or Pareto. Findings of this work indicate that the results of [SR06] are also applicable to P2P botnets. Therefore, it is indeed possible to model the churn in P2P botnets using a Weibull distribution.

The derived Weibull parameters presented in Table 16 can be used to study botnets in a simulation setting using realistic churn models. Moreover, the results could also be helpful for analyzing the success or effort of a botnet takedown plan, e.g., the time needed for a sinkholing attack.

- *Does the presence of highly responsive sensors affect the churn characteristics of P2P botnets?* In contrast to the initial expectations that sensors adversely influence the churn measurements due to their extremely high uptime, analysis on the two botnet families indicated that their influence was negligible. This was primarily due to the very few sensors deployed in this botnets. However, this observation is only applicable for the evaluated botnets within the measurement period. If these or other upcoming botnets gain popularity in the near future, chances are, more sensors may be deployed to monitor them. Therefore, sufficient attention needs to be given to ensure the resulting churn data is not affected by noise from the presence of increased number of sensor nodes.

## CONCLUSION AND OUTLOOK

---

Over the course of the previous chapters, issues relevant to advanced P2P botnet monitoring were visited. Particularly, existing botnet monitoring mechanisms are analyzed regarding the challenges often faced by them: the dynamic nature of P2P botnets and the various anti-monitoring mechanisms implemented to impede existing monitoring mechanisms. Based on that, this work proposed several countermeasures to circumvent existing anti-monitoring mechanisms. In addition, several new and advanced anti-monitoring mechanisms have been introduced to anticipate the next steps of the botmasters. This chapter summarizes the main contributions and findings of this dissertation and presents an outlook.

### 8.1 CONCLUSION

The adoption of a P2P-based architecture by recent botnets made their monitoring more difficult. Specifically, specialized monitoring mechanisms such as crawlers and sensors are required to monitor them. However, realizing the threats posed by such monitoring mechanisms, some botmasters have equipped their botnets with additional anti-monitoring mechanisms that impede botnet monitoring. Examples of such advanced botnets are GameOver Zeus, Sality, and ZeroAccess. In addition, the dynamic nature of P2P botnets represents as a hurdle to perform an efficient or effective botnet monitoring.

In this dissertation, requirements for an advanced botnet monitoring mechanism was derived to serve as a guideline for a discussion of the current state of the art in Chapter 3. The proposed requirements are not only aimed at producing high-quality monitoring data, but also to stay undetected from botmasters. This discussion revealed that many of the existing botnet monitoring mechanisms only partially fulfill the non-functional requirements that were outlined in Section 3.1.2. Such monitoring mechanisms may produce biased results, and introduce additional noise that affect the monitoring results of others. Moreover, anti-monitoring mechanisms of existing botnets also cause the botnet monitoring mechanisms fail in fulfilling the Stealthiness or Accuracy requirements. One example of such a mechanism is the NL restriction and automated blacklisting mechanism of GameOver Zeus that targets crawlers.

This dissertation analyzed and proposed a countermeasure called ZEUSMILKER to circumvent this NL restriction mechanism of GameOver Zeus that provably retrieves all neighbors of a bot. Then, to address the issue of evading the automated blacklisting mechanism of GameOver Zeus or to efficiently crawl any botnet, a novel crawling algorithm called Less Invasive Crawling Algorithm (LICA) is proposed which min-

minimizes the number of bots that needs to be crawled to discover most of the bots in the botnet.

In addition, this dissertation also introduced the design of a generic high-speed crawler called *Strobo-Crawler*. It can be configured to crawl a botnet at high-speed, but also be used to rate-limit the crawling frequency accordingly to evade blacklisting mechanisms such as that of *GameOver Zeus*. Moreover, the high-speed crawling of *Strobo-Crawler* minimizes the noise introduced in the resulting snapshots. As it provides a snapshot in less time, it reduces the noise noise being introduced in the resulting snapshots. In addition, the design of this crawler, which differentiate responsive bots from newly discovered bots, helps to prevent noise from being introduced in the resulting monitoring data, i.e., artifacts in the form of non-existing or offline bots. All in all, *ZEUSMILKER*, *LICA*, and *Strobo-Crawler* help to improve the efficiency and accuracy of crawling while being harder to detect by botmasters.

However, it is just a matter of time before the botmasters come up with new mechanism to impede monitoring. For this reason, this dissertation also introduces several anti-monitoring countermeasures from the perspective of a botmaster to raise the stake in this arms race. In particular, some lightweight crawler detection mechanism called *BoobyTraps (BTs)* are proposed that leverages design constraints of existing botnets. Evaluation results of these mechanisms on *Sality* and *ZeroAccess* indicates that many crawlers are susceptible to them. This is indeed worrying, since the idea of *BT* is relatively simple and can be easily implemented in existing botnets .

Prior to the works done in this dissertation, some researchers have claimed that sensor nodes are a more stealthy monitoring mechanism as they are difficult to be distinguished from regular bots. However, as another major contribution of this dissertation, three sensor detection mechanisms were proposed by leveraging graph-theoretic metrics to discern sensor nodes from regular bots. Evaluation results of these mechanisms in *Sality* and *ZeroAccess* indicates that quite a number of sensor nodes are susceptible to the proposed mechanisms. Particularly, the *SensorBuster* mechanism is able to accurately detect independent and colluding sensor nodes deployed in a botnet. To give the upper hand back to the defenders, the dissertation also discussed strategies that should be adopted by future sensor nodes to remain undetected by the proposed sensor detection mechanisms.

One major problem that can be foreseen with the advancement of botnet monitoring mechanisms is that most of these mechanisms would eventually tend to be more stealthier than they are now. While being stealthy may help to perform monitoring without being detected, this may also introduce adverse effects to the monitoring data. For instance, the stealthy monitoring footprint of an individual or organization will be assumed to be those of the botnet itself. This in turn would affect the accuracy and quality of the monitoring data.

To investigate if such a problem already exists in current botnets, this dissertation also investigates if the presence of highly-responsive sensor nodes may affect churn measurements of botnets. Based on

fine-grained churn measurements conducted in Sality and ZeroAccess, the SensorBuster mechanism was applied to identify sensor nodes in the botnets. However, due to a small number of sensor nodes deployed in the botnets, the presence of sensor nodes within the botnet overlay was found do not influence the overall churn measurements of the bots. However, the author believes that with an increasing monitoring activity in a botnet, there will be a more significant influence of sensor nodes on the overall churn measurements.

Concluding, the works presented in this dissertation indicate that the existing anti-monitoring mechanisms implemented by botnets or proposed by researchers are still in their infancy. They can either be circumvented or tolerated with sufficient computing resources at disposal. However, botmasters are expected to improve and introduce more advanced anti-monitoring mechanisms. For that, the defenders need to be prepared to face such advancements. This can be done by preempting the possible advancements from botmasters, i.e., proposing advanced anti-monitoring mechanisms, and attempt to devise countermeasures against them.

## 8.2 OUTLOOK

This dissertation primarily focused on proposing countermeasures to circumvent or tolerate the anti-monitoring mechanisms of existing botnets. However, it would definitely be interesting to identify to which extent botnet monitoring can always be performed. To answer this question, an in-depth investigation and analysis is required starting with the set of assumptions that are applied on both the botnets and the monitoring mechanisms themselves. For instance, usage of a honeypot to monitor a botnet usually allows defenders to obtain monitoring data from at least the view of a single bot. However, this is only possible if the malware is not able to detect the virtualization environment of the honeypot itself. Furthermore, crawling is able to enumerate bots by leveraging the MM protocols of the botnets. If a future botnet can be designed in a manner that does not require exchange of neighbors between bots to maintain connectivity with the overlay, crawling mechanisms are not usable in such botnets. Therefore, understanding the possible extents of botnet monitoring will help defenders to prioritize and focus in developing new countermeasures and efficient monitoring mechanisms.

In addition, the monitoring data collected in the context of this work also requires additional analysis. For instance, while the presence of sensor nodes are found to be negligible in the context of influencing churn measurements, this may not be the case when the network properties of the botnet overlay are analyzed. A detailed analysis needs to be done to understand the impact of unknown third party monitoring activities on other aspects of botnet monitoring analysis, e.g., static network properties like average path length.

Another interesting research direction would be to detect the botmasters managing the botnet. While this is usually difficult, as botmasters may use multiple chains of proxies to hide their identity, con-

tinuous high-speed snapshots of a botnet overlay captured using *Strobo-Crawler* can be used to trace command dissemination/update from bot to bot. These snapshots, can be used to identify a subset of bots that could have been used to propagate a new update to the botnet.

In developing a countermeasure for the *SensorBuster* mechanism (see Section 6.2.3), regular bots are required to be returned by sensor nodes to remain undetected by the mechanism. However, the action of returning valid bots may be deemed illegal from the perspective of cyber laws of a country. Therefore, it is also important to revisit existing cyber laws to provide possibilities of conducting stealthy monitoring activities in presence of advance countermeasures similar to those presented in this dissertation.

In addition, the presence of existing IP-based anti-monitoring mechanisms like those of *GameOver Zeus* requires a defender to possess a large pool of IP addresses that can be used for botnet monitoring. However, this is not always possible due to the scarcity of IPv4 addresses and/or organizational restrictions, i.e., limited allocation of IP addresses. Therefore, future work should focus into developing a botnet monitoring ecosystem that can utilize a large pool of shared IP addresses for botnet monitoring. This ecosystem can also be envisioned as a community-based botnet monitoring platform. Users interested in contributing to the ecosystem can voluntarily install a client-side application that allows the IP address of their Internet-enabled computing devices, e.g., smartphones, PCs, or servers, to be shared. For instance, distributed crawling can be achieved by relaying request messages using different IP addresses of volunteering clients to contact the bots. From the perspective of a bot, the distributed crawling would appear as though the requests are originating from many different bots. This way, IP-based anti-monitoring mechanisms can be easily circumvented using this community-based botnet monitoring platform.

Moreover, such a monitoring platform also enables collaboration opportunities for organizations or defenders that are interested in monitoring the botnets. By contributing resources, e.g., IP addresses or servers, for the monitoring platform, the obtained monitoring data can be shared among the collaborators. This can not only ensure a higher quality of monitoring data, i.e., lesser noise due to fewer independent monitoring activities, but also helps to spur additional research collaborations between different third parties that are interested in botnet monitoring.

Finally, anti-monitoring mechanisms that were analyzed in this work can be either circumvented or tolerated with sufficient computing and network resources. Therefore, to anticipate further advancements from the botmasters, more work from the perspective of a botmaster needs to be carried out. Particularly, the author believes that future botnets may attempt to restrict further the botnet information that are shared among bots to impede botnet monitoring. However, this exchange of information among bots are important for the management of the botnet overlay itself. Therefore, it is also interesting to

investigate the impact of such restriction mechanisms to the robustness and resilience of the constructed botnet overlay.



## A.1 PROOF OF PROPOSITION 5.1.3

*Proof.* The keys  $id_j$  in  $L$  are enumerated such that  $id_j \leq id_{j+1}$  for  $j = 0 \dots n - 2$ . For  $1 \leq j \leq n - 1$ , denote the common prefix of  $id_{j-1}$  and  $id_j$  by  $cp_j = cp(id_j, id_{j+1})$ . Let  $s_0 = o(b)$  and  $s_j = cp_j \parallel 1 \parallel o(b - |cp_j| - 1)$  be the smallest key in  $I(id_j, id_{j+1 \bmod n})$  that is closer to  $id_j$  than to  $id_{j-1 \bmod n}$  with regard to the XOR distance (see Eq. 1 for more detailed explanation on why this is the case). In the following, the sets  $F_{j-} = I(id_{j-1 \bmod n}, s_j)$  and  $F_{j+} = I(s_j - 1, id_j)$  is considered for  $j = 0 \dots n - 1$  and an elaboration that at least one spoofed key has to be chosen in  $F_{j-}$  and  $F_{j+}$  each for all  $j$  is provided. First note that by construction, the  $2n$  sets  $F_{j-} \cup F_{j+}$  for  $j = 0 \dots n - 1$  are all disjoint. For non-empty  $F_{j*}$  with  $0 \leq j \leq n - 1$  and  $* \in \{+, -\}$ , consider an arbitrary key  $x \in F_{j*}$ . If  $x$  is the first element in a neighbor list  $L' = L \cup \{x\}$ , it is only returned if a spoofed key  $s$  with  $XOR(s, x) < XOR(id_i, x)$  for all  $0 \leq i \leq n - 1$  is used. However, by the construction of the sets  $F_{j*}$ , all such keys are contained in  $F_{j*}$ . Thus, at least one query is required for each non-empty set  $F_{j*}$ . Hence, for all neighbor lists  $L$  with only non-empty  $F_{j*}$ ,  $2n$  queries are required. Such lists exists: an example for such a neighbor list is given by  $id_j = j \cdot 16 + 1$ , i.e. the first  $n$  hexadecimal numbers ending in 1. Then  $F_{j+} = \{j \cdot 16\}$  and  $F_{j-} = \{j \cdot 16 + 2, \dots, id_{(j+1) \bmod n}\}$  for all  $j$ . So, a lower bound on the worst-case complexity of an algorithm for guaranteed retrieval of neighbor lists is indeed  $2n$ .  $\square$

## A.2 PROBABILITY OF NON-OPTIMAL PERFORMANCE

Figure 26 gives an upper bound on the probability that a neighbor list in a network of  $m$  bots is not retrieved by ZEUSMILKER at the optimal cost for  $m \leq 1,000,000$ . The probability is computed based on Eq. 2.

## A.3 PROOF OF PROPOSITION 5.1.4

*Proof.* In this proof,  $P(A)$  is used to denote the probability of an event  $A$ ,  $P(A|B)$  for the probability of  $A$  conditioned on  $B$ , and  $\emptyset$  to denote the empty set. The proof of Proposition 5.1.3 gives a precise description of neighbor lists requiring  $2n$  spoofed keys, stating that the bound holds if all sets  $F_{j-}$  and  $F_{j+}$  are non-empty. Now an upper bound is given on the likelihood that an empty  $F_{j-}$  or  $F_{j+}$  exists. Because the distribution of keys in a neighbor list is unknown, an upper bound is obtained on the probability that for any pair  $(x, y)$  of keys in the network where  $y > x$ , the set  $I(x, y)$  of keys between them is empty.

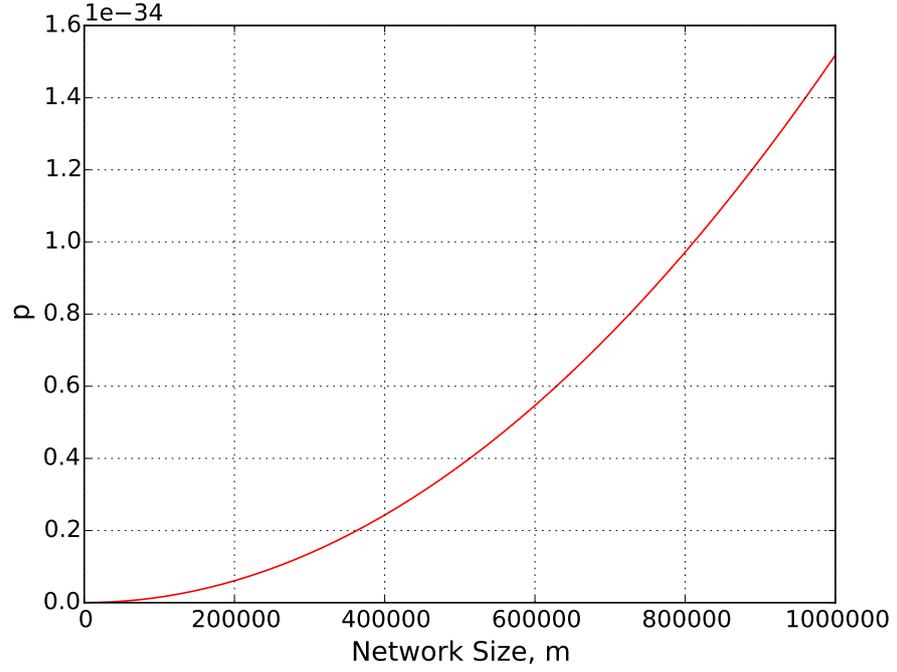


Figure 26: Quantity in Eq. 2 for  $b=160$  bits

Consider any two keys  $x$  and  $y$ : Recall that  $I(x,y)$  denotes the set of keys between  $x$  and  $y$ . Let  $F_-$  and  $F_+$  be the keys in  $I(x,y)$  closer to  $x$  and  $y$  with regard to the XOR, respectively. If  $|I(x,y)| \leq 1$ , either  $F_-$  or  $F_+$  is empty. Otherwise, there are  $|I(x,y)| + 1$  possibilities that the keys in  $I(x,y)$  could be divided between  $F_-$  and  $F_+$ , and only for two of them  $F_-$  or  $F_+$  is empty, namely if either  $x = \text{cp}(x,y) \parallel 0 \parallel \mathbf{1}(b - |\text{cp}(x,y)| - 1)$  or  $y = \text{cp}(x,y) \parallel 1 \parallel \mathbf{0}(b - |\text{cp}(x,y)| - 1)$ . Hence  $F_-$  or  $F_+$  is empty with probability  $\frac{2}{|I(x,y)|+1}$ . Let  $D$  denote the random variable giving  $|I(x,y)| + 1$  for two uniformly chosen keys. The probability that  $D$  attains the value  $d$  is

$$P(D = d) = \begin{cases} \frac{1}{2^b}, & d \in \{0, 2^{b-1}\} \\ \frac{2}{2^b} & 1 \leq d \leq 2^{b-1} - 1, \end{cases}$$

so that  $F_-$  or  $F_+$  is empty with probability

$$\begin{aligned} P(F_- = \emptyset \cup F_+ = \emptyset) &= \sum_{d=0}^{2^{b-1}} P(D = d) P(F_- = \emptyset \cup F_+ = \emptyset | D = d) \\ &= \frac{1}{2^b} \left( 3 + \sum_{d=2}^{2^{b-1}-1} \frac{4}{d} + \frac{2}{2^b} \right) \\ &\leq \frac{1}{2^b} (3 + 4(b-1) \ln 2). \end{aligned} \tag{7}$$

The last step follows because for the harmonic series  $\sum_{d=1}^m \frac{1}{d} \approx \ln m + \rho$  for the Euler-Mascheroni constant  $\rho = 0.577 \dots$ . Note that the keys in a neighbor list are not distributed uniformly, but are usually close to the node's own key. However, the keys of the  $m$  nodes in the network are selected uniformly at random. Therefore, the probability that none

of the set of keys closer to any of the  $m(m-1)/2$  pairs of keys is empty, is considered. An upper bound on the desired probability is obtained by a union bound using Eq. 7

$$P\left(\bigcup_j F_{j-} = \emptyset \cup F_{j+} = \emptyset\right) \leq \frac{m(m-1)}{2^{b+1}} (3 + 4(b-1) \ln 2).$$

□



## BIBLIOGRAPHY

---

- [Abu11] Abuse.ch. *Zeus Gets More Sophisticated Using P2P Techniques*. Tech. rep. 2011. URL: <http://www.abuse.ch/?p=3499>.
- [ARB15] Dennis Andriess, Christian Rossow, and Herbert Bos. “Reliable Recon in Adversarial Peer-to-Peer Botnets.” In: *ACM SIGCOMM Internet Measurement Conference (IMC)*. 2015. ISBN: 9781450338486.
- [And+13] Dennis Andriess, Christian Rossow, Brett Stone-Gross, Daniel Plohmann, and Herbert 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.
- [BYE85] R Bar-Yehuda and S Even. “A local-ratio theorem for approximating the weighted vertex cover problem.” In: *Annals of Discrete Mathematics* (1985).
- [BHK07] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. “OverSim: A Flexible Overlay Network Simulation Framework.” English. In: *Global Internet Symposium*. IEEE, 2007.
- [Böc+15] Leon Böck, Shankar Karuppayah, Tim Grube, Max Mühlhäuser, and Mathias Fischer. “Hide And Seek: Detecting Sensors In P2P Botnets.” In: *IEEE Conference on Communications and Network Security*. 2015, pp. 731–732. ISBN: 9781467378765. DOI: [10.1109/CNS.2015.7346908](https://doi.org/10.1109/CNS.2015.7346908).
- [CER13] CERT Polska. *Zeus-P2P monitoring and analysis*. Tech. rep. CERT Polska, 2013.
- [Dag+07] David Dagon, Guofei Gu, Christopher P. Lee, and Wenke Lee. “A Taxonomy of Botnet Structures.” In: *Computer Security Applications Conference (ACSAC)*. Ieee, 2007, pp. 325–339. ISBN: 0-7695-3060-5. DOI: [10.1109/ACSAC.2007.44](https://doi.org/10.1109/ACSAC.2007.44).
- [Dav+08] Carlton R Davis, Stephen Neville, José M Fernandez, Jean-Marc Robert, and John McHugh. “Structured Peer-to-Peer Overlay Networks: Ideal Botnets Command and Control Infrastructures?” In: *Computer Security - ESORICS 2008*. Ed. by Sushil Jajodia and Javier Lopez. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. Chap. Structured, pp. 461–480. ISBN: 978-3-540-88313-5. DOI: [10.1007/978-3-540-88313-5\\_30](https://doi.org/10.1007/978-3-540-88313-5_30). URL: [http://dx.doi.org/10.1007/978-3-540-88313-5\\_{\\\_}30](http://dx.doi.org/10.1007/978-3-540-88313-5_{\_}30).
- [DD08] David Dittrich and Sven Dietrich. “Discovery techniques for P2P botnets.” In: *Stevens Institute of Technology CS Technical Report 4* (2008).

- [EF94] Kjeld Egevang and Paul Francis. *The IP network address translator (NAT)*. Tech. rep. RFC 1631, 1994.
- [Enr+08] Brandon Enright, Geoff Voelker, Stefan Savage, Chris Kanich, and Kirill Levchenko. “Storm: when researchers collide.” In: *USENIX ;login:* 33 (2008), pp. 6–13.
- [Fal11] N Falliere. *Sality: Story of a Peer-to-Peer Viral Network*. Tech. rep. 2011, pp. 1–21.
- [FSK05] B. Ford, P. Srisuresh, and D. Kegel. “Peer-to-peer Communication Across Network Address Translator.” In: *USENIX Annual Technical Conference*. 2005. arXiv: [0603074](https://arxiv.org/abs/0603074) [cs].
- [Haa15] Steffen Haas. “Reverse-Engineering Windows-based P2P Botnets.” PhD thesis. TU Darmstadt, 2015.
- [Haa+16] Steffen Haas, Shankar Karuppayah, Selvakumar Manickam, Max Mühlhäuser, and Mathias Fischer. “On the Resilience of P2P-based Botnet Graphs.” In: *IEEE Conference on Communications and Network Security (CNS)*. 2016, (In Submission).
- [HSC08] A Hagberg, P Swart, and DS Chult. “Exploring network structure, dynamics, and function using NetworkX.” In: *SciPy* (2008), pp. 11–15.
- [Hol+08] Thorsten Holz, Moritz Steiner, Frederic Dahl, Ernst Bier-sack, and FC Freiling. “Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm.” In: *LEET* (2008).
- [HHH08] Ralf Hund, Matthias Hamann, and Thorsten Holz. “Towards Next-Generation Botnets.” In: *European Conf. on Computer Network Defense*. IEEE, 2008. ISBN: 978-0-7695-3479-4. DOI: [10.1109/EC2ND.2008.11](https://doi.org/10.1109/EC2ND.2008.11).
- [KCTL09] BBH Kang, E Chan-Tin, and CP Lee. “Towards complete node enumeration in a peer-to-peer botnet.” In: *Proceedings of International Symposium on Information, Computer, and Communications Security (ASIACCS)* (2009).
- [KLE08] Chris Kanich, Kirill Levchenko, and Brandon Enright. “The Heisenbot Uncertainty Problem: Challenges in Separating Bots from Chaff.” In: *Proceedings of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*. 2008.
- [Kar+14] Shankar Karuppayah, Mathias Fischer, Christian Rossow, and Max Mühlhäuser. “On Advanced Monitoring in Resilient and Unstructured P2P Botnets.” In: *IEEE International Conference on Communications (ICC)*. 2014. ISBN: 9781479920037. DOI: [10.1109/ICC.2014.6883429](https://doi.org/10.1109/ICC.2014.6883429).
- [Kar+15] Shankar Karuppayah, Stefanie Roos, Christian Rossow, Max Mühlhäuser, and Mathias Fischer. “ZeusMilker: Circumventing the P2P Zeus Neighbor List Restriction Mechanism.” In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2015.

- [Kar+16a] Shankar Karuppayah, Emmanouil Vasilomanolakis, Steffen Haas, Max Mühlhäuser, and Mathias Fischer. “Booby-Trap: On Autonomously Detecting and Characterizing Crawlers in P2P Botnets.” In: *IEEE International Conference on Communications (ICC)*, (In Print). 2016.
- [Kar+16b] Shankar Karuppayah, Leon Böck, Tim Grube, Selvakumar Manickam, Max Mühlhäuser, and Mathias Fischer. “SensorBuster: On Identifying Sensor Nodes in P2P Botnets.” In: *ACM Conference on Computer and Communications Security (CCS)*. 2016, (In Submission).
- [Kle15] Peter Kleissner. “Sality.” In: *Botconf*. 2015.
- [MM02] Petar Maymounkov and David Mazieres. “Kademlia: A peer-to-peer information system based on the xor metric.” In: *Peer-to-Peer Systems*. Vol. 2429. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 53–65.
- [McC03] B McCarty. “Botnets: big and bigger.” In: *Security & Privacy, IEEE* 1.4 (2003), pp. 87–90. ISSN: 1540-7993. DOI: [10.1109/MSECP.2003.1219079](https://doi.org/10.1109/MSECP.2003.1219079).
- [Naz07] J Nazario. “Botnet tracking: Tools, techniques, and lessons learned.” In: *Black Hat* (2007).
- [NG13] Alan Neville and Ross Gibb. “ZeroAccess Indepth.” In: *Symantec Security Response* (2013).
- [Pag+99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web*. Tech. rep. 1999, pp. 1–17. DOI: [10.1.1.31.1768](https://doi.org/10.1.1.31.1768). arXiv: [1111.4503v1](https://arxiv.org/abs/1111.4503v1).
- [Ped+11] F Pedregosa et al. “Scikit-learn: Machine Learning in Python.” In: *Journal of Machine Learning Research* 12 (2011). arXiv: [1201.0490](https://arxiv.org/abs/1201.0490).
- [Ros14] Christian Rossow. “Amplification Hell: Revisiting Network Protocols for DDoS Abuse.” In: *Network and Distributed System Security Symposium*. 2014. ISBN: 1891562355.
- [Ros+13] Christian Rossow, Dennis Andriess, Tillmann Werner, Brett Stone-gross, Daniel Plohmann, Christian J Dietrich, Herbert Bos, and Dell Secureworks. “P2PWED: Modeling and Evaluating the Resilience of Peer-to-Peer Botnets.” In: *IEEE Symposium on Security & Privacy*. 2013.
- [SS13] Hani Salah and Thorsten Strufe. “Capturing Connectivity Graphs of a Large-Scale P2P Overlay Network.” In: *IEEE International Conference on Distributed Computing Systems Workshops*. 2013.
- [Spi03] L Spitzner. “The HoneyNet Project: Trapping the Hackers.” In: *Security & Privacy, IEEE* 1.2 (2003), pp. 15–23. ISSN: 15407993. DOI: [10.1109/MSECP.2003.1193207](https://doi.org/10.1109/MSECP.2003.1193207).

- [SKK08] Guenther Starnberger, Christopher Kruegel, and Engin Kirda. "Overbot: a botnet protocol based on Kademia." In: *4th International Conference on Security and Privacy in Communication Networks*. ACM, 2008. ISBN: 9781605582412.
- [Ste07] Joe Stewart. *Storm Worm DDoS Attack*. 2007. URL: <http://www.secureworks.com/cyber-threat-intelligence/threats/storm-worm/> (visited on 11/17/2015).
- [Sg+09] Brett Stone-gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. "Your Botnet is My Botnet : Analysis of a Botnet Takeover." In: *Conference on Computer and Communications Security*. ACM, 2009. ISBN: 9781605583525.
- [SR06] Daniel Stutzbach and Reza Rejaie. "Understanding Churn in Peer-to-Peer Networks." In: *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*. 2006. ISBN: 1595935614.
- [SRS05] Daniel Stutzbach, Reza Rejaie, and Subhabrata Sen. "Characterizing Unstructured Overlay Topologies in Modern P2P File-Sharing Systems." In: *ACM SIGCOMM Internet Measurement Conference (IMC) (2005)*. DOI: [10.1145/1330107.1330114](https://doi.org/10.1145/1330107.1330114).
- [Sym13] Symantec. *Grappling with the ZeroAccess Botnet*. 2013. URL: <http://www.symantec.com/connect/blogs/grappling-zeroaccess-botnet?SID=sh1k1i488abj{\&}cjid=6146953>.
- [Tar72] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms." In: *SIAM Journal on Computing* 1.2 (1972). ISSN: 0272-4847. DOI: [10.1109/SWAT.1971.10](https://doi.org/10.1109/SWAT.1971.10).
- [Wan+09] Binbin Wang, Zhitang Li, Hao Tu, Zhengbing Hu, and Jun Hu. "Actively Measuring Bots in Peer-to-Peer Networks." In: *International Conference on Networks Security, Wireless Communications and Trusted Computing*. Vol. 1. 2009. ISBN: 9780769536101. DOI: [10.1109/NSWCTC.2009.288](https://doi.org/10.1109/NSWCTC.2009.288).
- [WSZ10] Ping Wang, Sherri Sparks, and Cliff C Zou. "An Advanced Hybrid Peer-to-Peer Botnet." In: *IEEE Transactions on Dependable and Secure Computing* 7.2 (2010), pp. 113–127. ISSN: 1545-5971. DOI: [10.1109/TDSC.2008.35](https://doi.org/10.1109/TDSC.2008.35). URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4569852>.
- [WS98] Duncan J. Watts and Steven H. Strogatz. "Collective Dynamics of "Small-World" Networks." In: *Nature* 393. June (1998).
- [Wyk12] J Wyke. "The ZeroAccess Botnet—Mining and Fraud for Massive Financial Gain." In: *Sophos Technical Paper* (2012). URL: [http://www.sophos.com/en-us/medialibrary/PDFs/technicalpapers/Sophos{\\\_}ZeroAccess{\\\_}Botnet.pdf](http://www.sophos.com/en-us/medialibrary/PDFs/technicalpapers/Sophos{\_}ZeroAccess{\_}Botnet.pdf).

- [YCE11] G Yan, S Chen, and S Eidenbenz. “RatBot: Anti-enumeration Peer-to-Peer Botnets.” In: *Information Security*. Vol. 7001. LNCS. Springer Berlin Heidelberg, 2011. ISBN: 9550091007.
- [Yan+14a] Jia Yan, Lingyun Ying, Yi Yang, Purui Su, and Dengguo Feng. “Long Term Tracking and Characterization of P2P Botnet.” In: *International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2014, pp. 244–251. ISBN: 9781479965137. DOI: [10.1109/TrustCom.2014.24](https://doi.org/10.1109/TrustCom.2014.24).
- [Yan+14b] Jia Yan, Lingyun Ying, Yi Yang, Purui Su, Qi Li, Hui Kong, and Dengguo Feng. “Revisiting Node Injection of P2P Botnet.” In: *Network and System Security*. Vol. 8792. Lecture Notes in Computer Science. 2014, pp. 124–137.
- [Yao+06] Zhongmei Yao, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov. “Modeling Heterogeneous User Churn and Local Resilience of Unstructured P2P Networks.” In: *IEEE International Conference on Network Protocols*. 2006, pp. 32–41. ISBN: 1424405939.



## ERKLÄRUNG

---

Hiermit erkläre ich die vorgelegte Arbeit zur Erlangung des akademischen Grades *Dr. rer. nat.* mit dem Titel

Advanced Monitoring in P2P Botnets

selbstständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Hiermit erkläre ich weiterhin, dass von mir bisher noch kein Promotionsversuch unternommen wurde.

*Darmstadt, 18 May 2016*

---

Shankar Karuppayah



## WISSENSCHAFTLICHER WERDEGANG

---

- 06/2006–08/2009 • Studium (B.Sc.) Computer Science,  
Universiti Sains Malaysia, Malaysia.
- 08/2009–09/2011 • Studium (M.Sc.) Software Systems Engineering,  
The Sirindhorn International Thai-German  
Graduate School of Engineering (TGGS),  
King Mongkut's University of Technology North  
Bangkok, Thailand.  
  
Masterarbeitsthema: *Selective Forwarding Attack:  
Detecting Colluding Nodes in Wireless Mesh  
Networks.*
- 09/2011–08/2012 • Forscher,  
National Advanced IPv6 Centre (NAv6),  
Universiti Sains Malaysia, Malaysia.
- 09/2012–Now • Stipendiat,  
Fachgebiet Telekooperation,  
Fachbereich Informatik,  
Technische Universität Darmstadt, Deutschland.