Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Botmaster Attribution in Large-Scale P2P Botnets

at the area of work ISS

Lars Leo Grätz

June 2019

Abstract

Botnets have been ever evolving threats for more than a decade. The possibility of infecting and using computers distributed over the internet enables attackers to carry out coordinated attacks and accumulate computing power. These networks of infected machines can be used for arbitrary malicious activities. The *botmaster* is able to release versions of malware that are executed by the computers of the botnet at will. This enables *DDoS*- and other potentially unpredictable attacks to take place.

With the newer and more resilient *P2P* botnet architectures, these threats have increased even further, since these botnets do not have a singular point of failure. Thus, to take down and find the root of such a botnet, the network specific communication protocols and mechanisms have to be exploited.

This thesis focuses on creating crawlers for the P2P botnet Sality. It analyzes Salitys communication protocols, as well as the general architecture of the botnet to find a way of exploiting botmaster behaviour. For this purpose a sophisticated simulation of the network has been implemented. This simulation is utilized to analyze different malware distribution methods the botmaster could potentially use and create *crawlers* to exploit these methods. Finally an implementation idea is given to test out the resulting crawlers in the real botnet.

Contents

1 Introduction

This chapter describes the general motivation of the thesis. First it is explained, why P2P botnets in general pose a big threat and introduces Sality as the subject for research. Then the brief outline of the following chapters is summarized.

1.1 Motivation

Botnets have been and will continue posing a threat to IT Systems. They are essentially a set of confiscated computers that execute commands of a botmaster. To do this the botmaster initially has to infect devices that are connected to the internet. Once he has control over a number of *bots*, commands can be propagated via a *C2* (Command & Control) channel [1]. This often was done centralized, where the botmaster would leave a new piece of code on a predefined server, that all infected machines downloaded. *Sinkholing* such a central server to get information about the botnet was relatively easy. Nowadays however, P2P propagation methods are far more common. In this distribution model, malware is propagated between infected machines, making it harder to track.

This controlled botnet can be exploited for various malicious attacks, such as DDoS (Distributed Denial of Service), distributed password cracking etc. This results in P2P botnet monitoring being an ever important task, especially to find out about propagation techniques as well as *entry points* of the botmaster.

This thesis states how one could possibly attribute botmasters in P2P networks, focusing on the botnet Sality, a P2P botnet that is used for various malicious attacks. In order to achieve this task, firstly different malware distribution techniques that are potentially used in Sality are evaluated. The found distribution technique that has the highest chance of being used in the real network is then used for further analysis. Building up on that technique, specialized crawlers are discussed and evaluated, that try to find a subset of *superpeers* possibly connected to the botmaster.

Regular monitoring techniques used to get an estimation of the botnet size as described in [2] can not be applied in this case, since the goal is not to find all members of the botnet, but rather: Given all members and connections, find the source of malware propagation. In order to narrow down the set of closely connected *peers*, the fact that new commands are propagated through the network in sequence, resulting in stepwise updates of individual bots, is used. Realizing this, the main lead for the crawlers is to identify bots that have a newer malware version than others and traverse the network accordingly.

1.2 Outline

In this first chapter, a brief overview of the topic, as well as motivation for the thesis was given. The following chapter displays the functional and nonfunctional technical requirements and states related work in the area. Additionaly an overview over relevant terms and functionality in P2P botnets is given. The third chapter describes the design of the simulation and its entities. The fourth chapter evaluates different malware propagation techniques and states which one is most likely being used in Sality. In the fifth chapter the resulting crawlers are explained and evaluated. The final chapter summarizes the thesis and provides further information on possible future work.

1.3 Requirements

The following functional and nonfunctional requirements summarize the work of the thesis:

Functional Requirements:

1. Identification of botmaster strategies: The thesis evaluates how the malware is possibly propagated throughout the Sality botnet.

2. Narrowing down botmaster entrypoints: Crawlers are created to find a certain set of superpeers that is likely to be connected to the botmaster.

Nonfunctional Requirements:

1. Genericity: The result of the thesis can be used to traverse different P2P botnets, by implementing the specific communication protocols.

2. Scalability: The speed of the crawlers scale with the size of a botnet by adding more processing power.

3. Efficiency: The crawlers avoid unnecessary overhead and only exchange messages that are needed.

4. Avoiding detection: The crawler works around popular botnet defense mechanism that detect crawlers.

2 Related work

This chapter displays the techniqual requirements of the crawlers. Additionaly related work on P2P botnets, especially Sality is provided and and a short introduction to crawler design is given.

2.1 Botnets

According to Grizzard et al. [3] the primary goal of a botnet is one of the following: information dispersion (sending out spam, DoS attacks etc.), information harvesting (obtaining data), information processing (password cracking etc.). Botnets can be difficult to detect for various reasons, such as low data traffic, few bots, or encrypted communication [4]. Generally botnets can be classified by their architecture, centralised or decentralised, as well as the communication topology of the C2 channel.

Centralised botnets Conventional botnets that require servers for information transmission. In this setting, the botmaster uploads new malware to these centralised servers. Bots then have to poll the endpoints regularly to gather the new commands. With this architecture, a botnet can be created without much effort. However the centralisation itself is a singular point of failure, making it easier to sinkhole and take down centralised servers. The controlled servers often use the IRC or HTTP protocol to expose endpoints for the bots [2].

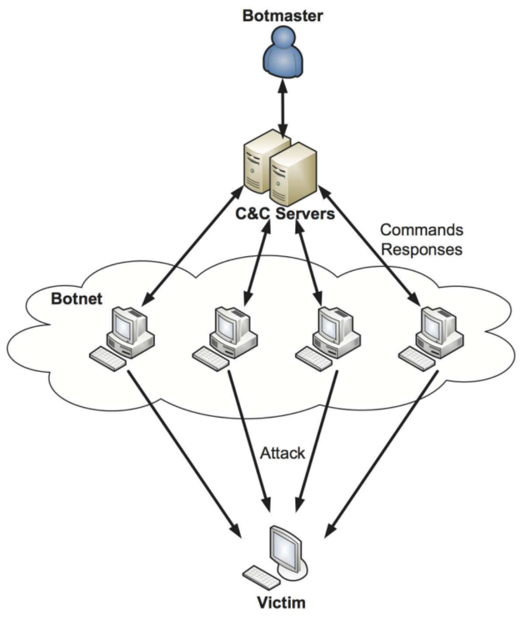


Figure 1: Centralised botnet architecture [4]

Decentralised botnets This newer architecture of botnets was created to circumvent the singular point of failure, a centralised botnet has. One such variant uses a *DGA* (Domain Generation Algorithm) to generate new domain names given certain environment variables such as the date etc. This allows the botmaster to use different servers to distribute his malware. Sinkholing one such server simply delays the process of malware distribution, but does not kill the botnet [2]. Another architecture lies in the P2P connected botnets, the focus of this thesis.

P2P botnets distribute malware between peers, instead of having them poll the data from centralised servers. This is done by differentiating bots between superpeers, which are routable servers that can directly be contacted, and peers, which are not routable and thus have to poll information from superpeers [2].

In general each superpeer in a P2P botnet holds a list of neighbours (also superpeers), that can directly be contacted. This *neighbourlist* differs between bots, since it is dynamic and thus changes over time, depending on the accessibility of the neighbours. Each bot runs periodic *MM* (membership maintenance) cycles in order to identify non responsible bots within its neighbourlist. Non responsive bots are often discarded at a given point. This also means that a superpeer will try to gain new neigbhbours, once its neighbourlist reaches a low threshold of entries. This is done by contacting reliable neighbours and polling from their neighbourlist [2].

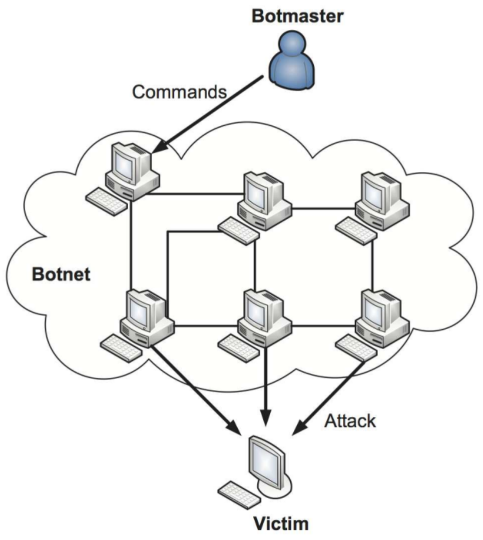


Figure 2: P2P botnet architecture [4]

2.2 Crawlers

The following subsections provide a brief overview over common crawler characteristics and implementation techniques as well as anti-crawling techniques of botnets

Crawlers are used to retrieve information about a botnets size as well as communication behaviour. A crawler uses the botnet specific communication protocols to contact and communicate with peers. In order to do this, the crawler itself disguises itself as a peer and participates in the botnet. Often, a botnet needs to be reverse engineered to fully understand the protocols needed to insert a sophisticated crawler [5].

One fluke of crawlers is the inability to contact peers, that are hidden behind *NAT* (Network Adress Translation), which are not publicly reachable over the internet. These peers often represent the biggest part of a botnets population (up to 90%) but can only be estimated [2].

Anti-crawling techniques

Most P2P botnets apply anti-crawling techniques that identify and block crawlers from botnet participation. These can be classified into prevention, detection and response:

- **Prevention** Botnets can try to prevent crawlers by design. They either stop the crawler from any communication or try to slow it down drastically, often focusing on neighbourlist return mechanism. One example for this is to only return a small portion of a neighbourlist, when a peer receives a neighbourlist request. Some botnets have peers solve time intensive algorithms before receiving a neighbour response [2].

- **Detection** In order to detect unusual behaviour, botnets might blacklist IPs that send many requests in a given time. If the protocol is not implemented in the proper way, a botnet might also detect crawlers by observing communication anomalies, or using botnet intern *sensors* [6].

- **Response** Often botnets contain static blacklists of IP adresses, that are known to monitor botnets. Alternatively some botnets just start a DDoS attack on the IP monitoring node [2].

Sality uses prevention by only letting a peer return one random neighbour, whenever it receives a neighbourlist request. In order to circumvent this restriction, a crawler for Sality is able to send neighbourlist requests continously to a peer until it converges towards the set of neighbours [2].

2.3 Sensors

Sensors are passive components injected into the botnet to receive messages without actively having to poll. Sensors are deployed to the network and announce themselves. The goal is to get into as many neighbourlists as possible to be spread even further. Thus in contrast to crawlers they can also reach peers instead of just superpeers, since their information will be propagated by superpeers, that give out reliable neighbours. This leads to regular peers contacting the sensor. Since most botnets, such as Sality evict unresponsive neighbours from neighbourlists, sensors must deploy the botnet specific communication protocols and furthermore react to incoming messages. Thus they are harder to maintain and setup than regular crawlers, but provide deeper insight into the botnet topology while being able to discover all bots not only superpeers.

2.4 Sality

This thesis investigates the P2P version of the botnet Sality, that spreads via a polymorphic file infector for windows executables. Sality originally was developed as a centralised botnet, which was first observed in 2003. In 2008, the first P2P version (V3) was found, followed later on by the newest, most resiliant version: V4 [7]. Both V3 and V4 are still active today.

Overview

Sality infects machines by concatenating malicious payload to valid windows binaries. Then the entry point of the binary is obscured, such that it executes the malicious code, and afterwards jumps back to the original binary [8]. This way, new malware can easily be deployed at any time by letting the malicious binary download new instructions that will be executed. The new included malware can then be used to carry out a number of tasks, such as shutting down services, deleting/encrypting files, sending spam, using the host for computational tasks etc. The propagation of new malware through the Sality botnet is as follows: The malicious code is deployed to certain servers by the botmaster. He then proceeds to distribute a *URL pack* throughout the network, a message of links to the servers that host the new version of the malicious code. This means, that the botmaster can use different IP Adresses each time, since they will be part of the propagated URL pack. Each bot that receives one such pack downloads and executes the new malware [8]. This leads to the necessity of URL packs having a unique *sequence number*, since a bot should not download outdated malware. When a new URL pack is released, this number is simply incremented, such that different versions can be discriminated. This necessarily leads to the situation of different URL pack versions being present in a snapshot of the botnet at times, when a new pack has just been released by the botmaster.

Protocols

Salitys superpeers typically hold a neighbourlist of up to 1000 entries, that additionally contains the *LastOnline* timestamp, a *GoodCount*, IP adress, Port and UID for each neighbour.

The main message types according to [2] are:

- **Hello message** Probes a neighbour for responsiveness. On a successful response, the LastOnline timestamp is set to the current time and the

GoodCount is incremented. If a timeout occurs or the bot is unresponsive, the GoodCount is decremented. With the probe message, the current sequence number is also delivered.

- **Neighbourlist request message** Probes a neighbour for additional superpeers. Bots use this to build their corresponding neighbourlist. The receiving peer will respond with one randomly chosen entry from a list of potential candidates.

The MM cycle is invoked every 40 minutes, which starts the following processes for each neighbour sequentially:

1. Probe the responsiveness of the neighbour using a hello message. Depending on the sequence number received, either ask for the whole URL pack, if its sequence number is lower or send back its own URL pack, if it is higher.

2. The own superpeer status is tested. This is necessary for a bot to know if it is a superpeer and can propagate messages. When a bot is initialized it starts off with UID $= 0$, meaning its superpeer capabilities are unknown. If it has any other UID it is a superpeer. Thus, a bot with a UID of 0 will test his own status.

3. If the size of the own neighbourlist is < 980 and the neighbour has a high GoodCount, it is also probed for a neighbour entry. The answering superpeer returns an entry that is chosen from a list of potential candidates that have a high GoodCount.

After the cycle, a cleanup process takes place. Bots that have a GoodCount less than are dropped from the list, if the size of the neighbourlist is at least 500. [2].

2.5 Strobo crawler

Haas et al. [9] created a crawler, that estimates the size and connections of the Sality botnet. The main focus of this crawler lies on accurately tracking neighbourlist changes and node churn of the network via high frequent periodic crawling. Figure 3 dsiplays the basic architecture deployed. It consists of three modules and a list of enumerated nodes that is maintained by the Prober Module. This list contains all known peers, that are periodically probed by the Prober Module at a given frequency f_c. The list is further distinguished into all nodes V^M and online nodes $V^O \subseteq V^M$. Incoming messages, such as responses, are handled via the Receiver Module. If the Receiver Module receives a message, it reports it to the Prober Module to update the lists of nodes. If the received message is a neighbourlist message, new unknown nodes are potentially discovered. A Session Module contains online information of nodes in form of sessions. The sessions consist of points of first and last contact. Sessions are kept open, if the bot does not time out. In this case a timeout is defined as not answering a predefined number of probes. If the bot does time out, it is removed from the lists of known nodes.

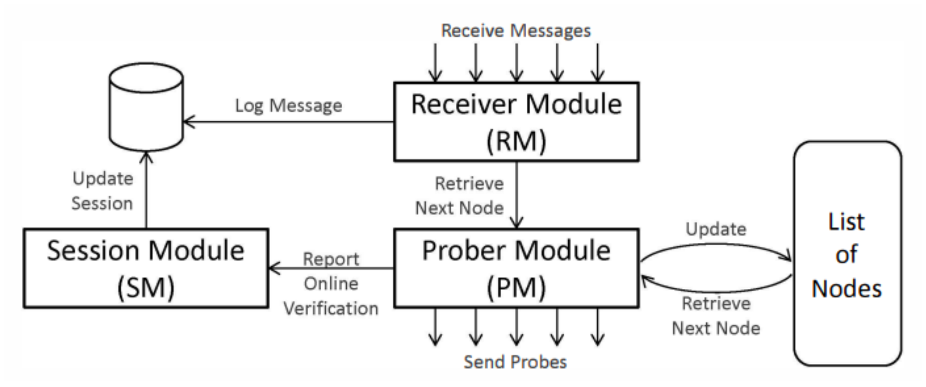


Figure 3: Strobo Crawler introduced in [9]

Snapshots of the discovered network architecture can be reconstructed for specific timestamps. A general reconstruction that is valid over an undefined period of time is not possible, because of node churn and MM resulting in a changing network topology. To reconstruct such a topology, Haas et al. [9] explain a time constrainted method. First off, a timestamp t has to be chosen. Then, for each peer the times of the last and next MM cycle around timestep t are determined. This is done, since neighbourlists are not consistent during MM cycles and thus the topology should be reconstructed inbetween such cycles. Since these MM times differ for each bot, the topology is not reconstructed for the specific timestamp t but rather for a $\delta(t)$, where δ is a function that determines the offset of the time at topology information is collected to the time t.

The Sality specific implementation of the Strobo Crawler and graph reconstruction uses hello messages for session initiation. Once the online status of a bot has been confirmed, neighbourlist requests are send out. If neighbourlist replies are received by the Receiver Module, hello messages are no longer send [9]. To deduct the MM cycle information, a seperate sensor node is deployed, that receives messages of MM cycle starts. Since Salitys superpeers only share information about specific candidate bots, that have a high GoodCount as explained in section 2.4, only these peers can be retrieved.

3 Simulation design

This chapter describes the system design of the simulation environment. Firstly a broad overview of the simulation environment is given. Afterwards, the individual steps needed to run a simulation are explained in detail.

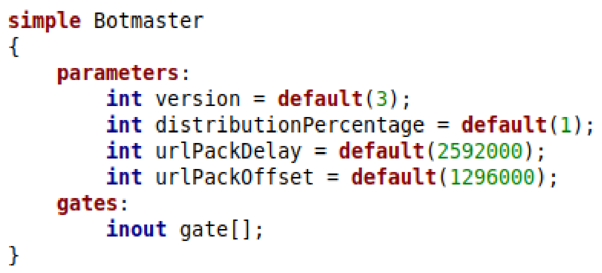
3.1 Overview

To test malware propagation strategies, as well as crawlers, a simulation environment in *OMNeT++* (Objective Modular Network Testbed in C++) has been created. OMNeT++ is a discrete event simulation framework. In a discrete event system, state changes happen at specified time instances without delay. The time between events is skipped, since no actions are specified. Events itself such as sending a message are retrieved from an event queue and executed sequentially [10]. The simulation time is given in seconds. These properties can be used to simulate large periods of real network behaviour within a short period of time, depending on computing power. Thus an implementation via the OMNeT++ framework is very scalable and well suited for simulating potentially big P2P botnets.

The simulation environment this thesis describes, features an implementation of Salitys protocols, message formats and superpeer behaviour. Regular peers are of no interest in the simulation, since they cannot propagate URL packs and thus do not supply information about the botmaster. Additionally the behaviour of different crawlers is also implemented. The main entities of the simulation are:

- **Botmaster** The botmaster propagating the malware. Three different versions of the botmaster can be selected, that propagate the URL packs in different ways, further explained in section 4.2.1.
- **Superpeer** The public routable peers of the botnet, that have been crawled in the existing network and parsed into the simulation environment as visualized in section 2.5. These superpeers behave conform to the Sality protocol explained in section 2.4.
- **Crawler** The crawler to traverse the botnet towards the botmaster. Multiple crawler versions can be selected, each using different algorithms to explore the network that are further reviewed in section 4.3.

All these entities are defined as modules. OMNeT++ modules declare the individual nodes of the simulated network. These are able to communicate with each other and execute arbitrary behaviour, since they are simply written in C++. This means that the botmasters, superpeers and crawlers each are defined in C++ classes with their own individual behaviour. To declare modules, a module description, as well as behaviour have to be implemented. The description is in form of a NED file, which defines parameters, as well as communication gates. Figure 4 displays an example setup of a botmaster module. The parameters can be accessed in the C++ classes, that implement the behaviour. Gates determine ways for other modules to communicate with a specific module.



```
simple Botmaster
{
    parameters:
        int version = default(3);
        int distributionPercentage = default(1);
        int urlPackDelay = default(2592000);
        int urlPackOffset = default(1296000);
    gates:
        inout gate[];
}
```

Figure 4: Example NED file of a botmaster

Modules are able to communicate via messages. In OMNeT++ messages come in the form of the cMessage class, that can be transmitted between modules. These messages hold different attributes as well as network statistics such as a message name, creation time, sender, receiver, transmission channel etc.. The cMessage class can be extended to create individual messages. In the case of this thesis, it was extended to create URL pack messages. As described in section 2.4 messages between superpeers are exchanged in different states. Firstly, with each MM cycle, a peer probes all of its neighbours. This is implemented via a URL probe message. Secondly, if any peer receives a probe message with a lower sequence number than its own, it sends back its sequence number via a URL pack message. When it receives a higher sequence number, it asks for the whole URL pack via its own probe message. Since regular peers are of no interest in this thesis, the superpeer probe message is not implemented. This however does not change superpeer behaviour and thus can be omitted without altering simulation output. Figure 5 displays the basic superpeer behaviour on receiving a message.

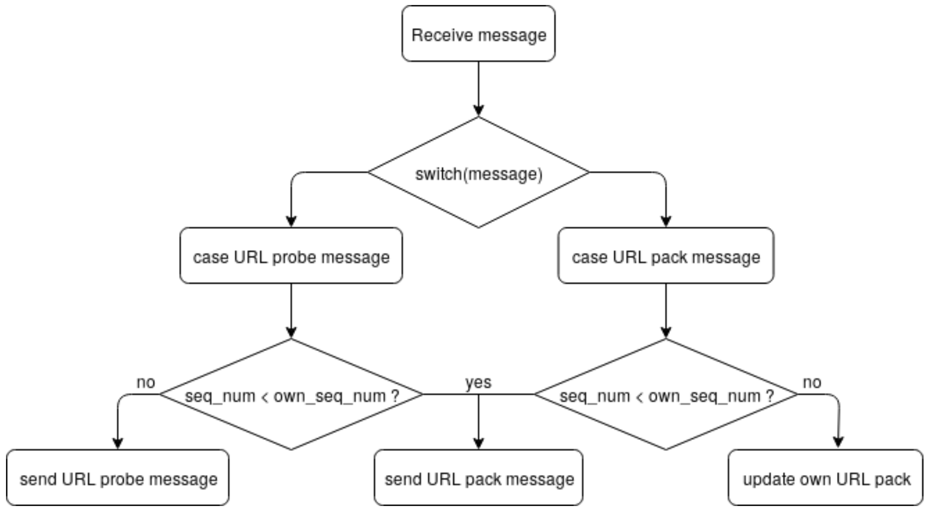
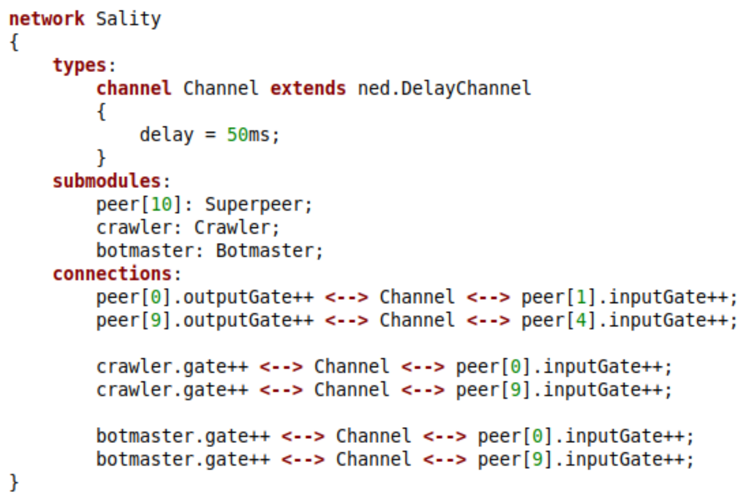


Figure 5: Superpeer behaviour on receiving a message

All communication happens via channels. OMNeT++ channels represent connections between modules. Channels can define behaviour such as network delay etc.. In order to simulate global communication behaviour, a minimum delay of 50ms is assumed. Furthermore delay is added for each individual message transmission based on a geometric distribution. This distribution adds on average between 50-150 additional ms of delay.

To condense network setup information, OMNeT++ also uses NED files. These network specific files contain modules, channels, connections and further information on the network architecture. Figure 6 displays an example NED file of the botnet Sality. For brevity information like imports etc. is skipped. Also only a few superpeers and connections are included. The network sality contains a channel definition, submodules and connections. The channel is used for the communication between the individual modules and holds the described base message delay of 50ms. The submodules contain the set of superpeers, a crawler and a botmaster module. The crawler is optional and only used for the crawler evaluation in section 4.3. The set of peers and the botmaster however are needed for the simulation of the Sality network. Each module is further connected to other modules in the connection section. In this case bidirectional channels are used, since superpeers need to be able to respond to URL pack messages.



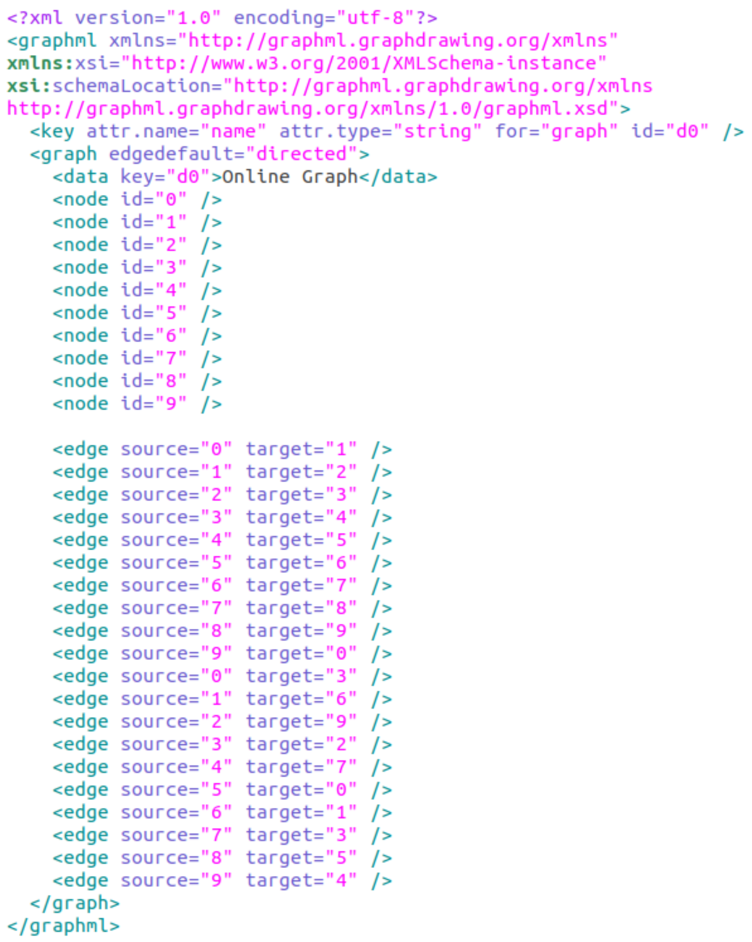
```
network Sality
{
    types:
        channel Channel extends ned.DelayChannel
        {
            delay = 50ms;
        }
    submodules:
        peer[10]: Superpeer;
        crawler: Crawler;
        botmaster: Botmaster;
    connections:
        peer[0].outputGate++ <--> Channel <--> peer[1].inputGate++;
        peer[9].outputGate++ <--> Channel <--> peer[4].inputGate++;

        crawler.gate++ <--> Channel <--> peer[0].inputGate++;
        crawler.gate++ <--> Channel <--> peer[9].inputGate++;

        botmaster.gate++ <--> Channel <--> peer[0].inputGate++;
        botmaster.gate++ <--> Channel <--> peer[9].inputGate++;
}
```

Figure 6: Simplified Sality NED file

Since snapshots of the Sality botnet are static, the simulation environment also holds a static set of superpeers. For the evaluation of botmaster strategies in section 4.2 the propagation is tested in this static environment. However, since crawlers in existing botnets need to overcome node churn, superpeers in the simulation also implement node churn behaviour. This is then used in section 4.3 to better simulate real crawling behaviour.

3.2 Simulation steps

To run a simulation of the Sality botnet, first the network topology has to be established. For this task, the output of the Strobo crawler 2.5 as seen in figure 3 is used to reconstruct the network structure, which is saved in a graphml file. This file declares nodes and edges. Each node represents one superpeer, each edge a connection between two superpeers of the existing Sality network.

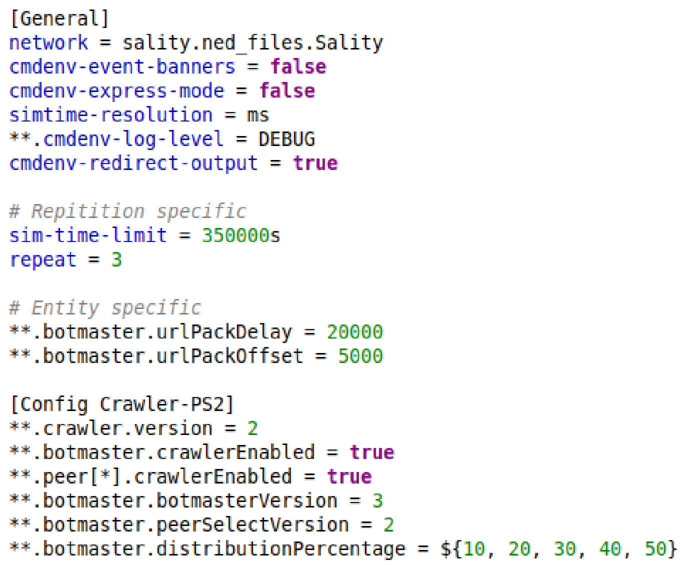


```xml
<?xml version="1.0" encoding="utf-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <key attr.name="name" attr.type="string" for="graph" id="d0" />
  <graph edgedefault="directed">
    <data key="d0">Online Graph</data>
    <node id="0" />
    <node id="1" />
    <node id="2" />
    <node id="3" />
    <node id="4" />
    <node id="5" />
    <node id="6" />
    <node id="7" />
    <node id="8" />
    <node id="9" />

    <edge source="0" target="1" />
    <edge source="1" target="2" />
    <edge source="2" target="3" />
    <edge source="3" target="4" />
    <edge source="4" target="5" />
    <edge source="5" target="6" />
    <edge source="6" target="7" />
    <edge source="7" target="8" />
    <edge source="8" target="9" />
    <edge source="9" target="0" />
    <edge source="0" target="3" />
    <edge source="1" target="6" />
    <edge source="2" target="9" />
    <edge source="3" target="2" />
    <edge source="4" target="7" />
    <edge source="5" target="0" />
    <edge source="6" target="1" />
    <edge source="7" target="3" />
    <edge source="8" target="5" />
    <edge source="9" target="4" />
  </graph>
</graphml>
```

Figure 7: Simplified graphml file with 10 nodes and 20 connections

These graphml files are used to create the simulation environment, which is a parsed version of the connection graph. Thus, each node of the graphml file represents one superpeer entity in the OMNeT++ simulation, each edge a connection between two superpeers. A script (graphml-to-ned.py) parses the GraphML file and returns a NED file that can be used in OMNeT++. This script additionaly introduces a botmaster to the topology and optionally a crawler. Both the botmaster and the crawler are able to contact a predefined set of superpeers depending on the botmaster strategy and crawler version used in the simulation. To achieve this for both entities edges to all peers are added in the NED file similar to figure 6.

Once a network structure has been established, different simulations can be run depending on the botmaster strategy and crawler version of interest. Each strategy limits the known peers of the botmaster and thus the edges used. Simulation configurations for OMNeT++ are declared in an initialization file. A configuration defines the global as well as local variables necessary for a simulation run. Figure 8 displays such an initialization file. The two configurations General and Crawler-PS2 each represent runnable simulations. However, the General configuration is also run before each other configuration, making it ideal for global variable setup that is needed in each individual configuration. The global attributes in this example are highlighted in blue, while the entity specific local attributes name the entity they reside in. These local attributes are conform to the parameters declared in the NED file of an entity as displayed in figure 4 and must match. Thus each parameter can be overwritten in the initialization file.



```
[General]
network = sality.ned_files.Sality
cmdenv-event-banners = false
cmdenv-express-mode = false
simtime-resolution = ms
**.cmdenv-log-level = DEBUG
cmdenv-redirect-output = true

# Repitition specific
sim-time-limit = 350000s
repeat = 3

# Entity specific
**.botmaster.urlPackDelay = 20000
**.botmaster.urlPackOffset = 5000

[Config Crawler-PS2]
**.crawler.version = 2
**.botmaster.crawlerEnabled = true
**.peer[*].crawlerEnabled = true
**.botmaster.botmasterVersion = 3
**.botmaster.peerSelectVersion = 2
**.botmaster.distributionPercentage = ${10, 20, 30, 40, 50}
```

Figure 8: OMNeT++ initialization file

Given a whole project setup including an initialization file, individual configurations can be run using an OMNeT++ executable. The script 1 runs a configuration identified by name on up to six processors in parallel. The parallel execution can be further scaled to support more cores. Since OMNeT++ in its core is a discrete event simulation framework, one individual simulation run can only be executed on one processor. However, the scaling happens by running repetitions or different parameter combinations on different processors.

In order to evaluate the resulting run data, it is written to log files during the simulation process. This provides the groundwork for further processing of the information in order to retrieve insights on propagation statistics and crawler results.

4 Evaluation

4.1 Data sets

This chapter analyzes the provided data. It firstly introduces the analytical evaluation parameters for botnet architecture. Then it evaluates the graphml file used for the simulation.

The data used for analysis of the botnet architecture is a reconstructed graphml snapshot from the Strobo Crawler as explained in 2.5. Furthermore individual message logs from the Strobo Crawler have been provided, which are mainly used for comparison of botmaster strategies to real botmaster behaviour in section 4.2. These logs are in the form of messages, that the Receiver Module collected from peers. Each message holds the following information:

- **Timestamp** The time at which the RM received the message in the format: JJJJ-MM-DD hh:mm:ss.
- **IP** The IP address of the peer.
- **NodeType** The type of the peer. A 'client' denotes a peer that initiated the request and thus can either be a regular peer or a superpeer. A 'server' denotes, that the received message is a response to a message of the Prober Module and thus comes from a superpeer.
- **CMD** The protocol used by the contacting peer, explained in section 2.4.
- **PackID** The sequence number of the URL pack the peer holds.

Botnet architecture statistics

To analyze botnet characteristics, a graph model is used:

$$G = (V, E)$$

G is the graph of the botnet that consists of superpeers (V) and directed edges between them (E). Each directed edge $e \in E$ can be displayed as (u, v), where u is the superpeer that holds v as an entry in its neighbourlist. It is noteworthy, that V is a set, since no superpeers appear twice. Also e is not reflexive and thus superpeers are not connected to themselves. Furthermore the following characteristics are considered:

- **Number of superpeers**
$$num_v := |V|$$
The amount of unique superpeers the graph holds.

- **Number of edges**
$$num_e := |E|$$
The total number of directed edges in the graph. This corresponds to the accumulated number of neighbourlist entries among all superpeers of a snapshot.

- **Neighbourlist**
$$NL_u := NL_u \subseteq E \wedge \forall e \in NL_u : e = (u, v), v \in E \wedge v \neq u$$
The set of neighbourlist entries of an individual superpeer u.

- **Neighbourlist size**
$$NLS_u := |NL_u|$$
The size of a superpeers neighbourlist and thus the total amount of superpeers he directly is able to contact.

- **Sorted set of neighbourlist sizes**
$$set_{nl} := \{NL_u \mid u \in V\}$$
This set is sorted in ascending order. Thus the first entry is the smallest size of a superpeers neighbourlist, the last entry the largest.

Building up on the graph representation of the botnet, the following statistics are evaluated for Salitys snapshots:

- **Mean connections per superpeer:**
$$\mu_e := \frac{num_e}{num_v}$$
Describes the mean size of a superpeers neighbourlist across all superpeers.

- **Median connections per superpeer:**
$$\hat{x}_e := set_{nl}\left[\left\lfloor \frac{num_v}{2} \right\rfloor\right]$$
Displays the mean amount of connections across all superpeers. The brackets denote an index operator.

- **Connectiveness:**
$$\lambda := \frac{\mu_e}{num_v - 1} = \frac{num_e}{num_v \cdot (num_v - 1)}$$
The connectiveness displays the amount of possible connections utilized. Hence it displays how connected the superpeers are. This is because a fully connected directed graph can hold at maximum $n \cdot (n-1)$ edges, where n is the number of nodes. It is in the range $[0, 1]$, where a value of 0 means that $\forall u \in V : NLS_u = 0$, a value of indicates a fully connected graph, since in this case $\mu_e = num_v - 1$ holds true.

- **Closely connected superpeers**

$$peers_c \subseteq V := \forall u \in peers_c : NLS_u \geq \frac{num_v}{2}$$

The set of superpeers that know at least 50% of all other superpeers.

- **Loosely connected superpeers**

$$peers_l \subseteq V := \forall u \in peers_l : NLS_u \leq \frac{num_v}{10}$$

Superpeers are seen as loosely connected, if they know at most 10% of the botnets superpeers.

- **Regularly connected superpeers**

$$peers_r \subseteq V := V \setminus (peers_c \cup peers_l)$$

Superpeers are seen as regularly connected, if they know more than 10% but less than 50% of the botnets superpeers.

- **Outliers**

$$peers_o \subseteq V := \forall u \in peers_o : \left| \sum_{v \in V, v \neq u} (v,u) \in E \right| \leq \frac{num_v}{100}$$

Outliers are superpeers that are known by at most 1% of the network.

- **Isolated superpeers**

$$peers_i \subseteq V := \forall u \in peers_i : u \in peers_o \land \forall v \in (V \setminus peers_i) : \nexists\ e(u,v)$$

$$\hookrightarrow peers_i \subseteq peers_o$$

Isolated superpeers are outliers that are only known by other outliers.

It is importand to note that the closely, regularly and loosely connected status considers only outgoing edges of a superpeer, whilst the outlier and isolated status considers ingoing edges.

Sality architecture evaluation

The script 2 parses a graphml file reconstruction of the Strobo Crawler output, as explained in section 3.2 and extracts the relevant statistics. Table 1 displays the resulting statistics.

num_v	num_e	μ_e	$\hat{x}_e$	λ	$peers_c$	$peers_l$	$peers_r$	$peers_o$	$peers_i$
677	231869	342	372	0.51	519	16	142	120	89

Table 1: Statistic of the provided graphml file

The fact that $\hat{x}_e > \mu_e$ shows, how unevenly the botnet is connected, dividing it into a large set of $peers_c$ and a relatively small set of $peers_l$. The set $peers_c$ makes up about 77% of the network, $peers_l$ about 3%, leaving $\approx 20\%$ for $peers_r$. The connectiveness is quite high, considering that the total of 231869 measured connections is more than what would be possible in a fully connected undirected graph, which can yield a maximum of 228826 connections for 677 nodes, as calculated by $n \cdot (n-1)/2$, where n is the number of nodes. This means that the botnet graph in general is very densely connected. Approximately 18% of all superpeers are in $peers_o$, circa 13% are in $peers_i$.

These statistics divide the superpeers into a core network $peers_{core} = (V \setminus peers_i) \approx 82\%$ and the outlier part $peers_o \approx 18\%$. Furthermore approximately 83% of the well connected peers ($peers_{conn} = peers_c \cup peers_r$) are also in $peers_{core}$. This shows, that a part of the network ($\approx 70\%$) is densely connected with each other, leaving the rest of the nodes more isolated in regards to ingoing and outgoing edges.

For the remainder of the thesis p_d describes the set of superpeers that are densely connected, p_i the set of isolated superpeers and $peers_c$ the set of closely connected peers for the imported snapshot of Sality.

4.2 Botmaster strategies

This section evaluates the different strategies the botmaster potentially uses to distribute URL packs. The botmaster strategy is estimated by comparing statistics from the simulation environment to ones from Sality. In order to achieve this, the simulation is run with different hyperparameters to account for various botmaster behaviours. Each simulation run logs relevant statistics. The Strobo Crawler provides log files from the Sality network. These files are then further analyzed and compared to find a propagation technique that fits the real behaviour.

4.2.1 Distribution methods

The following distribution methods of a botmaster behaviour are evaluated:

1. Active Botmaster 1 (AB1): This botmaster is not part of the network. Instead, it pushes the new sequence numbers to a set of superpeers, using the default communication protocol described in section 2.4.

2. Active Botmaster 2 (AB2): Also not part of the network. This variant pushes the new URL packs directly to a set of superpeers, avoiding the default communication protocol, resulting in faster propagation time compared to the Active Botmaster 1. This method could possibly be used in the existing Sality botnet, given the communication patterns described in [2].

3. Passive Botmaster (PB): This botmaster is part of the network in form of controlled superpeers. These controlled peers simply increment their own sequence numbers periodically without the need to actively push it to a set of superpeers. This means, that other superpeers have to actively poll the new URL packs. Essentially this is equivalent to an AB2 without message delay or loss, since no network communication between the botmaster and the controlled peers play a role. However, since the botmaster has to own the superpeers, the amount of controlled peers is realistically limited. Furthermore he would not be able to simultaneously update all controlled peers without delay. Because of this, the PB is only tested with the botmaster as part of the network but no further controlled superpeers, since this is already done in AB2.

In additon to the distribution methods, the active botmaster is also able to choose the set of superpeers he distributes the URL pack towards. Different *peer_selection* strategies are explored:

- **Random selection** As the name suggests, the botmaster chooses a random subset of superpeers to propagate the malware towards. In the simulation this is accounted for by choosing an evenly distributed subset of superpeers amongst all bots. The botmaster propagates the new URL packs towards this same set of randomly chosen superpeers.
- **Most neighbours** The botmaster could achieve faster propagation time by contacting superpeers with a high neighbour count. If the botmaster actively monitors his own botnet, he is able to estimate neighbourhood information similar to the Strobo Crawler explained in section 2.5. Thus this version keeps a set of superpeers, that have the most connections.
- **Next MM cycle** Another valid strategy is to choose bots, that enter their respective MM cycles earliest according to the current timestep. This provides faster propagation, since these superpeers contact all their neighbours within the next cycle, directly sending the new URL pack. In this strategy the amount of connections a superpeer has is not considered. The botmaster is assumed to know all superpeers and choose the ones that enter the next MM cycle earliest.
- **Mixed MM connection strategy** This strategy accounts for the fact, that it is unlikely for the botmaster to know all superpeers and choose the ones that enter their corresponding MM cycle earliest, as is done in the next MM cycle strategy. Also a botmaster might not want to propagate the packs towards the same set of superpeers each time as done in the neighbourhood strategy, which is easily traceable. Thus the botmaster might hold a set of densely connected superpeers and choose a subset of bots that enter their MM cycle the earliest. This allows the botmaster to gain fast propagation times whilst not being easily traceable.

4.2.2 Evaluation parameters

To provide a meaningful comparison, different distribution parameters are evaluated. Each strategy is thus evaluated for a set of different hyperparameters:

- *sim_time_limit*: The simulation time limit in seconds. This is mainly used to get different propagation behaviour since MM-cycles play out differently depending on the time of the simulation.
- *botmaster_distribution_percentage*: If the botmaster is using one of the above mentioned active distribution methods, this percentage states the amount of peers he directly contacts.

Due to the different hyperparameter combinations, individual simulations are run with the following parameter sets:

- Passive botmaster: In this case only *sim_time_limit* = {15768000s, 31536000s} holds different values. No further hyperparameters are evaluated.
- AB with *peer_selection* strategies MM cycle and neighbourlist: *sim_time_limit* = {15768000s, 31536000s}, *botmaster_distribution_percentage* = {10, 20, 30, 40, 50}. Both active botmaster distribution methods are evaluated as the crossproduct of the *sim_time_limit* and *botmaster_distribution_percentage*, resulting in 10 different simulations each. The distribution percentages were evaluated via a trial and error approach to find a best fit to the botnet data.
- AB with mixed *peer_selection* strategy: For this strategy the botmaster is assumed to know 60% of the botnet. The *sim_time_limit* = {15768000s, 31536000s}, *botmaster_distribution_percentage* = {30, 40, 50, 60, 70, 80}. The *botmaster_distribution_percentage* is in regards to the 60% of known peers.

In order to retrieve meaningful statistics given certain hyperparameter settings, each simulation is run three times with different seeds. This affects the random propagation and release times of URL packs, such that each run yields different results. When considering the *sim_time_limit*, the seeds, the botmaster strategies and the different *peer_selection* strategies, this results in a total of 222 runs. This number is sufficient to gather average distribution statistics. The following run statistics are evaluated:

- *mean_propagation_delay* in seconds until $x\%$ superpeers receive a URL pack, calculated by:

$$\frac{1}{n} \cdot \sum_{i=1}^{n} (receive_i^{(x)} - release_i)$$

where $receive_i^{(x)}$ is the timestamp in seconds, at which $x\%$ superpeers received URL pack of sequence number i and $release_i$ the number timestamp in seconds, at which the botmaster released the URL pack of version i.

- *max_pack_delay*: Max propagation time in seconds until $x\%$ superpeers receive a URL pack. This is the maximum amount of seconds measured, until any URL pack was propagated by $x\%$.
- *min_pack_delay*: Min propagation time in seconds until $x\%$ superpeers receive a URL pack. This is the minimum amount of seconds measured, until any URL pack was propagated by $x\%$.
- *message_loss*, calculated by:

$$\frac{1}{n} \cdot \sum_{i=1}^{n} (numPeers - numPeers_i)$$

  where $numPeers$ is the total number of superpeers in the network and $numPeers_i$ the number of superpeers, that have received the URL pack of version i.

In order to retrieve meaningful results, the statistics of the individual runs are collected. Afterwards the mean values for the same runs with different seeds are taken to remove outliers and gather meaningful insights.

Sality evaluation

To be able to compare the simulation results to the real Sality behaviour, multiple message logs from the Strobo Crawler have been gathered. The format of these messages is displayed in section 4.1.
To evaluate the URL pack distribution time over the superpeers, only messages with the CMD of 'server' are analyzed. For all messages of these superpeers, that have been received, the message with the earliest timestamp is considered to be the time of the superpeer receiving the URL pack for the first time. Of course this is not conform to the real reception time, since there is always a delay δ_i for each superpeer message reception rm_i of the Receiver Module. This δ_i is defined by: $rm_i - recep_i$, where $recep_i$ denotes the time at which superpeer i received the URL pack for the first time from any other superpeer j.

For this task, the script 3 collects the messages for the given URL packs and extracts the average, minimum and maximum URL pack delay as also used in the simulation evaluation. This results in comparable statistics between the simulation and the real botnet data.

4.2.3 Results

Both in the real Sality network and the simulation all URL packs are eventually propagated, without one being missed out by certain peers, which results in a $message\_loss = 0$ for all simulation runs. This indicates that the superpeers form a fully connected graph, such that all messages eventually are propagated to all peers. The botnet infrastructure of the Sality network changes with each analysis of a new URL pack as a result of the node churn and Strobo crawler inaccuracies. It holds on average 300-500 superpeers, whilst the simulated network holds a constant 677 superpeers. Since only percentual distribution times are of concern, this inequality does not have a big influence on the comparability.

Figure 9 displays the propagation statistics for the passive approach compared to Salitys. It is noteworthy, that in the Sality network approximately 70% of the superpeers receive the new malware in less than 30 seconds. Afterwards the time towards 100% propagation rises exponentially. This is conform to the findings in section 4.1, where the set of densely connected superpeers p_d makes up about 70% of the network, leaving the other 30% in p_i loosely connected. Haas et al. [9] also pointed out, that the Sality botnet is split into a large set of closely connected superpeers and a smaller more isolated set. Peers thus either know and are known by most other superpeers, or nearly none. Bots joining the network tend to be in p_i for a rather long time. This is the result of the long delay between MM cycles of 40 minutes, which makes the process of gaining reputation tedious. A new superpeer first has to become part of other peers neighbourlists by sending individual messages to neighbouring superpeers. Then it has to gain high GoodCount ratings by being online for exended periods of time and thus responding to probes of these bots. Finally it is propagated throughout the network as a reliable neighbour. The closely connected superpeers on the other hand are very likely propagate the according packs towards each other, since each established one will receive multiple messages from other superpeers once a pack has been released. The newer outlier peers however often have to wait multiple MM cycles to receive a new pack.

The simulation data in figure 9 does not correspond to Salitys statistics. A new URL pack takes about 5 minutes to spread towards about 8% of the superpeers such that in following MM cycles it will be propagated faster. Thus the propagation function has an exponential shape. Salitys propagation function on the other hand is rather logarithmic. However, on the minimum delay, a drastic jump is visible from $5 - 60\%$ distribution percentage that is due to the simultaneous overlapping MM cycles of multiple superpeers, resulting in a more logarithmic shape and a faster spread of the URL pack early on.

This data suggests, that the botmaster is not just part of the network, but rather uses a different distribution mechanism. This makes sense, since the P2P structure of Sality would not efficiently be used, if the botmaster itself was part of it. It would lead to the botmaster being the C2 server, which could simply be traced and sinkholed. Because of this, the passive botmaster does not seem to be the strategy followed. Since the passive strategy is unlikely, the following sections display and interpret the results of the different *peer_selection* stategies for the active botmasters.

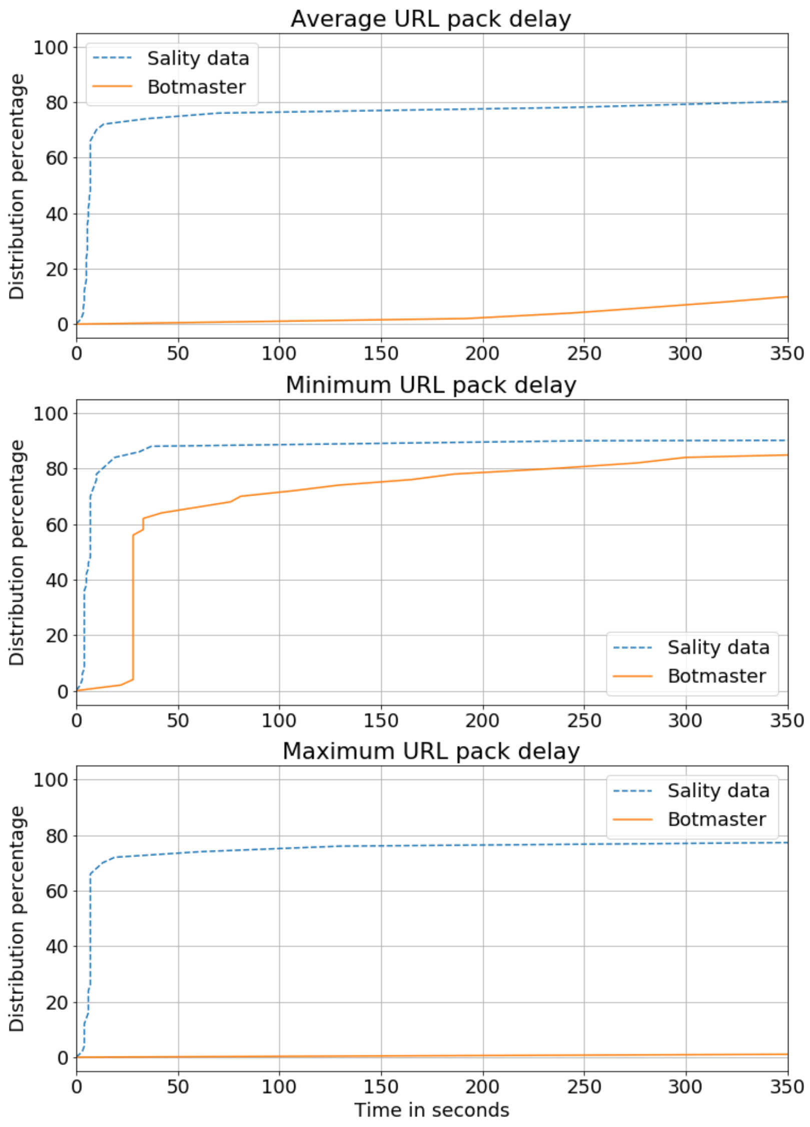


Figure 9: Distribution statistics of the PB

Random selection

The figure 10 shows the statistics for the AB2 method with different values for the *distribution_percentages*. The average URL pack delays seem to fit Salitys propagation characteristics. The vertical lines seen below the 50 second mark indicate that a closely connected superpeer in $peers_c$ has spread the new pack towards its neighbours. Because p_d makes up about 70% of the network, the horizontal jumps end between $60 - 70\%$, depending on the connectiveness of the peer that spread the pack. Afterwards the time towards full propagation rises exponentially as in Salitys statistic Since the botmaster chooses his set of known superpeers randomly in the beginning, there is a varying delay until a superpeer in $peers_c$ spreads the new URL pack towards its neighbours. This can be seen by comparing the average to the minimum and maximum pack delay. If a superpeer in p_d receives the URL pack quickly and enters its corresponding MM cycle fast, the pack is transmitted to approximately 70% of the botnet within the first few seconds of its release as visualized by the minimum URL pack delay. If, on the other hand, the receiving superpeers are not in p_d or do not enter their MM cycles quickly, the delay until the jump can take multiple minutes as seen in the maximum URL pack delay. Knowing this, the botmaster might consider only choosing well connected superpeers for propagation as analyzed in the next section.

The AB1 distribution mechanism does not differentiate from the AB2 a lot. The statistics are nearly identical. This is due to the fact, that the message propagation delay is only a fraction $\approx 0.008\%$ of the MM cycle delay. This results in the MM cycle delay being way more influential on the message propagation than communication overhead of the protocol. Thus, for the evaluation of the remaining *peer_selection* strategies only the AB3 approach is used. This is because after evaluation of the different strategies, the delay due to the communication protocol does not influence the output of any *peer_selection* strategy in a relevant way.

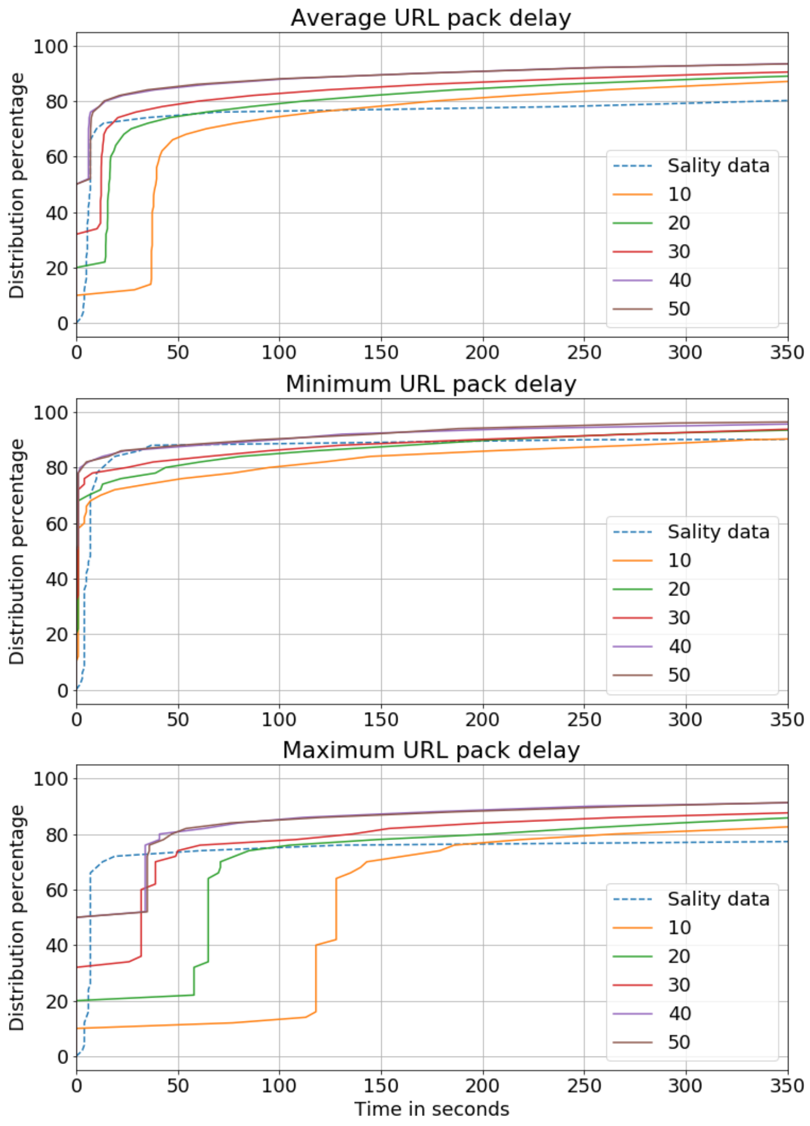


Figure 10: AB2 selecting peers randomly

Selection based on the neighbourlists

Figure 11 displays the evaluation of the neighbourlist *peer_selection* strategy. It is noteworthy that the propagation behaviour is similar to that of the random selection strategy. Once a superpeer in $peers_c$ propagates the URL pack, a vertical jump can be seen.

The main advantage of this strategy in comparison to the random selection lies in the maximum URL pack delay being smaller. This is a direct result of choosing only closely connected superpeers to propagate the URL packs that have many outgoing edges. Once such a superpeer of $peers_c$ propagates the pack, it always reaches a subset of superpeers in $peers_d$. Thus, bots in $peers_c$ are able to contact a high percentage of the network simultaneously, reach densely connected superpeers and as a result propagate the URL pack faster than the isolated superpeers. However, the outlier peers that are not densely connected still take a long time to receive and propagate new packs, which still results in an exponential increase of propagation time towards 100%.

Overall this strategy is a direct improvement in comparison to the random selection strategy. Whilst yielding similar distribution curves, the delay is smaller. It is also easier to maintain, since the set $peers_c$ is less prone to node churn and more stable. Node churn leads to a slow decrease in the percentage of known superpeers, which leads to higher delay.

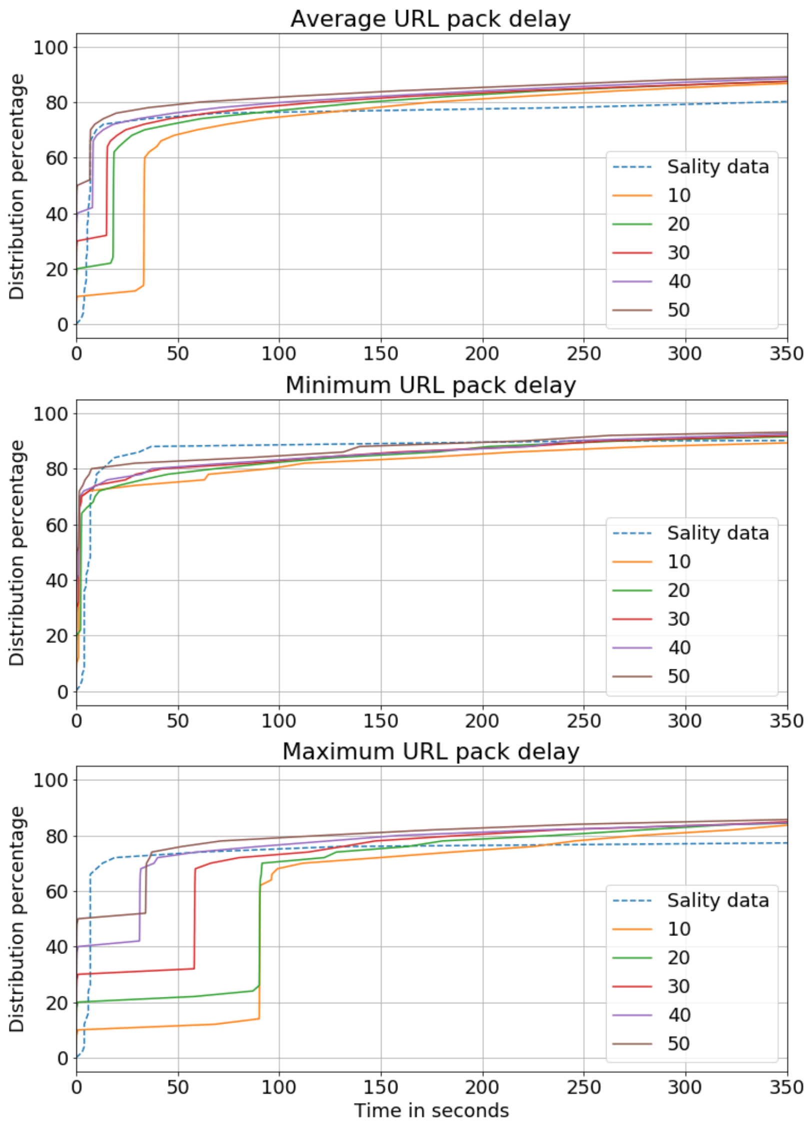


Figure 11: AB3 selecting peers based on neighbourlist connectivity

Selection based on the next MM cycle

Figure 12 displays the statistics for the MM cycle *peer_selection* strategy. The botmaster in this case chooses between $10 - 50\%$ of superpeers from the botnet to propagate the URL pack towards, based on the delay to their next MM cycle. The closest ones are chosen. As mentioned this strategy does not consider the connectiveness of the chosen peers. The average delay displays that for a *botmaster_distribution_percentage* of at least 40%, the URL Pack is propagated to about 70% of the network in less than 40 seconds. This is a result of the fact that p_d makes up about 70% of the network. Thus by taking a subset of at least 40%, a set of superpeers in p_d is always chosen for propagation. This results in a high chance of choosing superpeers in $peers_c$ to propagate the new pack. The average delays for a *botmaster_distribution_percentage* between $10 - 30\%$ are thus longer, since superpeers in p_d are not necessarily chosen for the initial propagation.

Compared to the neighbourlist *peer_selection* strategy, the distribution takes longer. Especially the maximum URL pack delay is larger. This is due to the fact, that any superpeer that is close to running its corresponding MM cycle is chosen for distribution. However since this method does not consider the per connectivity, outliers might be chosen that are not able to propagate the URL packs very quickly. The minimum URL pack delay on the other hand is very close to Salitys data, even for a low percentage of known superpeers. This could happen, if the botmaster chose very connected superpeers to distribute the URL packs towards. These bots would then go on to further propagate the packs towards a large set of other superpeers. Also, the minimum URL pack delay is very close to that of the neighbourlist *peer_selection* strategy. This is due to the fact that both strategies behave similarily in the case that leads to the minimum pack delay: For the neighbourlist strategy, the chosen superpeers happen to enter their corresponding MM cycle quickly. For the MM cycle strategy, the chosen superpeers happen to be well connected.

Using the insights gained from the neighbourlist- and MM cycle- strategy, a botmaster might incorporate both methods for better results. The botmaster might only know a percentage of the closely connected set of superpeers $peers_c$ that have a small churn rate. For each release of a new URL pack he might choose $x\%$ of these bots that are close to their MM cycle. This strategy is also simpler to follow, since the botmaster does not have to monitor all superpeers.

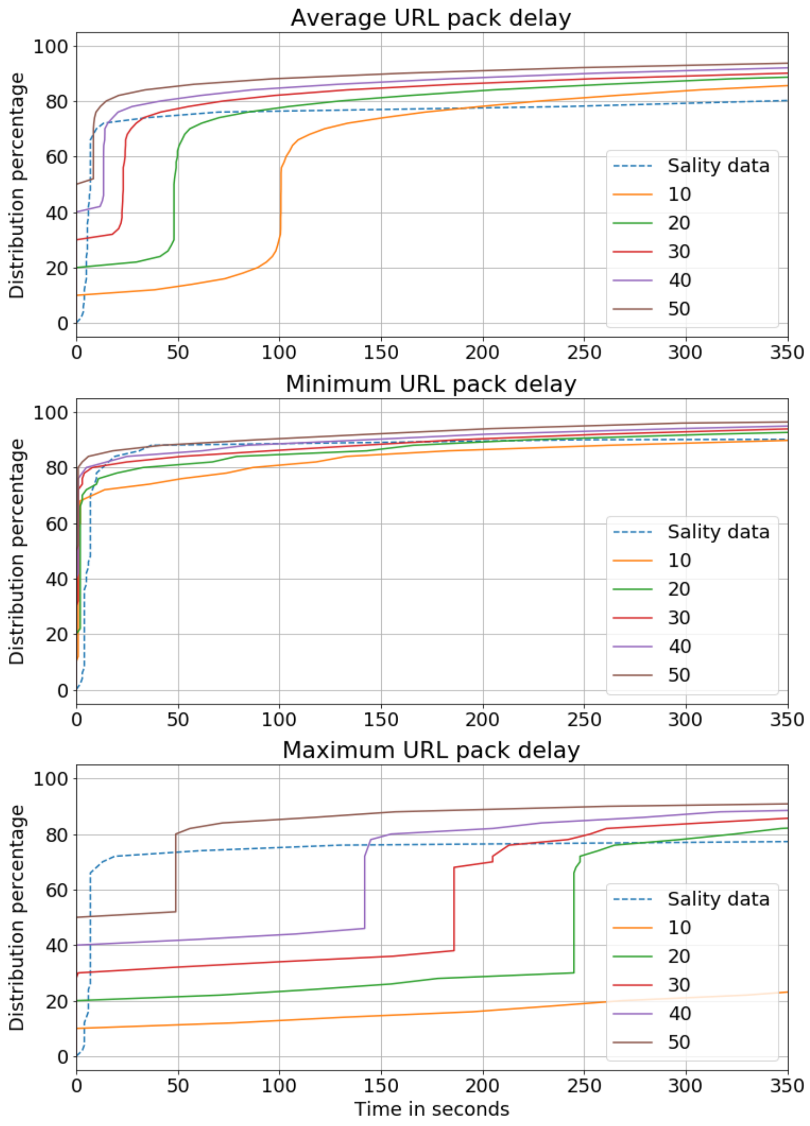


Figure 12: AB3 selecting peers based on the next MM cycle delay

Selection based on the mixed strategy

Figure 13 displays the statistics for the AB2 selecting superpeers based on the mixed strategy. In this case, the botmaster is assumed to know about 85% of the botnets superpeers that are in $peers_c$, which results in them making up about 65% of the network, denoted by p_b. This is feasable, since these bots have a high reputation, which results in them being easily discovered and monitored. These densely connected superpeers also tend to be part of the botnet for a longer time than loosely connected ones. For each new release of a URL pack he then chooses a subset:

$$p_p \subset p_b : |p_p| \approx \frac{x \cdot |p_b|}{100}$$

This subset consists of x% of the superpeers in p_b that are closest to their MM cycle.

The average delay curves are very similar to one another. The smaller x is, the longer the delay until the vertical jump towards approximately 70% propagation takes. This is due to the fact that a higher percentage x corresponds to a higher chance of a shorter delay to the next MM cycle of a superpeer in p_p. Thus the higher percentage x leads to a longer average delay. For the minimum URL pack delay all curves are similar. This happens when the next MM cycles for a subset of the chosen p_p come up in a matter of seconds. Since all bots of p_p are also in p_b, this results in a quick jump towards approximately 70% propagation. The maximum URL pack delay displays the importance of a larger percentage x. For $x = 30 - 40\%$ the delay until a bot in p_p that is very well connected propagates the pack can take about 3 minutes. For a percentage $x \geq 50$, there is a delay limit of about 50 seconds. This limit can vary depending on the current botnet state, but is a result of the different MM cycle offsets.

In comparison to the neighbourlist strategy the mixed strategy has slower propagation times, but is harder to track. This is because the neighbourlist strategy always chooses the same set of peers that have the most neighbours whilst the mixed strategy often chooses a set of less connected neighbours that differs each URL pack. In comparison to the MM cycle strategy, the mixed strategy is just better, since no isolated peers are chosen to propagate the URL pack.

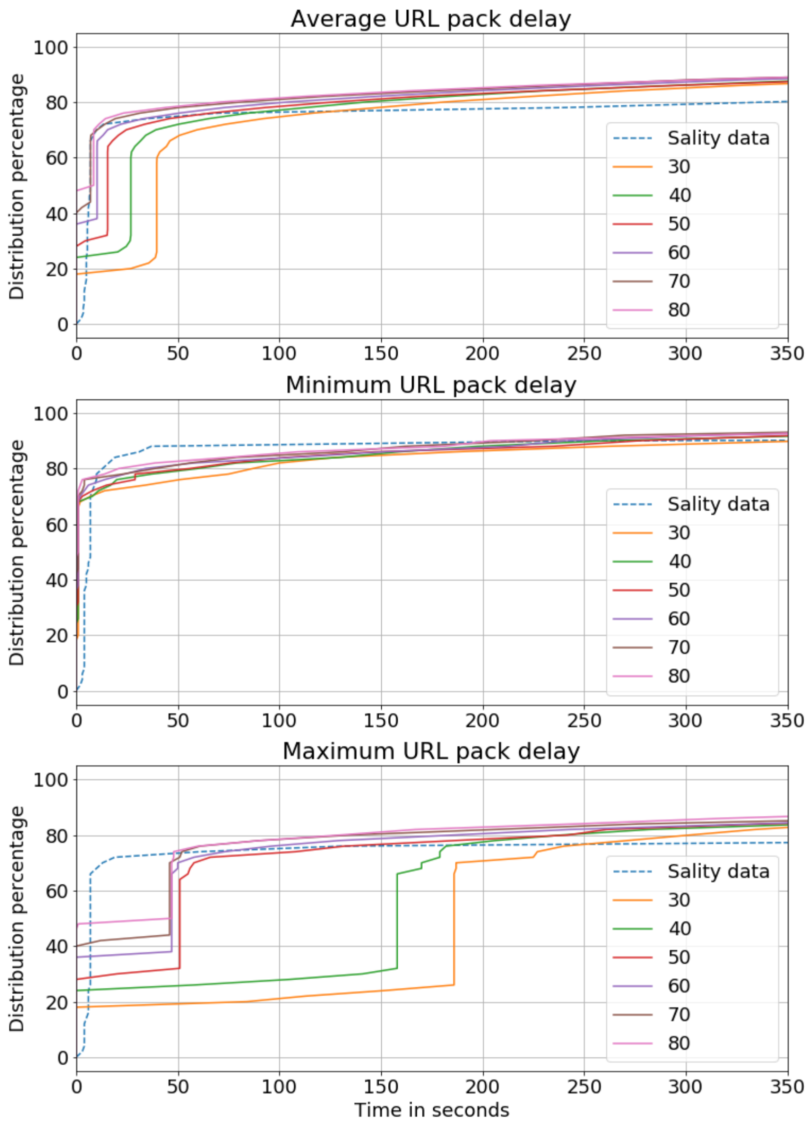


Figure 13: AB3 selecting peers based on the mixed strategy

After evaluating the different botmaster strategies, the passive botmaster can be excluded from the list of valid strategies. The AB1 and AB2 methods do not differ much. However, since the AB2 method has the advantage of less communication overhead it is considered to be more effective whilst not having any downsides. For the *peer_selection_strategies* the neighbourlist- as well as the mixed strategy are the most effective ones. This is because the neighbourlist strategy is more effective than the random strategy in each aspect, the mixed strategy is more effective than the MM cycle strategy in each aspect. Between those two strategies the neighbourlist delivers faster propagation times, whilst being easier to track. The mixed strategy on the other hand delivers acceptable delay whilst being harder to track. No strategy however fits Salitys data perfectly. This is to be expected, since the networks topology of the simulation differs a lot from the topology of the current Sality network.

4.3 Crawlers

This section evaluates a crawler that traverses the network towards the malware source, evaluating the AB2 in combination with the neighbourlist- and mixed-peer_selection_strategies found in section 4.2.3.

A crawler has been implemented, that focuses on the two most likely *peer_selection_strategies* (neighbourlist and mixed). It thus covers the different requirements these strategies come with: In the neighbourlist strategy, the set of superpeers the botmaster uses is statically chosen in the beginning and does not change during the simulation. On the other hand, with the mixed strategy, the set changes with each propagation of a new URL pack.

Graph model

For the sake of evaluation of the crawler, a formal model of the botnet is used. Following is a description of all relevant parts:

- V_S The set of all superpeers in the botnet.
- V_B The subset of superpeers the botmaster chooses for propagation: $V_B \subseteq V_S$. For both strategies this set is determined at the beginning of the simulation.
- V_E The resulting set of superpeers, that a crawler outputs. This set contains all superpeers that have been found are potentially connected to the botmaster. $V_E \subseteq V_S$ thus is a necessary condition.
- seq_{max} The maximum sequence number of any URL pack that has been propagated by the botmaster.
- seq_u The maximum sequence number bot $u \in V_S$ holds.

4.3.1 Evaluation metrics

Different metrics are established to measure the success of the crawler:

1. **Recall** The recall of a crawler is defined as:

$$\frac{|V_E \cap V_B|}{|V_B|} \cdot 100$$

   This is the percentage of chosen superpeers successfully retrieved by the crawler.

2. **Precision** The precision of a crawler is defined as:

$$\frac{|V_E \cap V_B|}{|V_E|} \cdot 100$$

   This is the percentage of superpeers a crawler rightfully retrieved.

4.3.2 Crawler Design

The main focus of the crawler design is summarized in the following points:

1. Work with the neighbourlist and mixed strategy, since these are the most likely ones to be used.

2. Applicable in the real botnet. Because URL packs are on average released every month according to the provided Sality data (REF???), the crawler has to output a relatively good estimate of V_E with few URL packs.

3. Easy integration into existing crawlers. It should especially work with crawlers and sensors already deployed in the Sality botnet.

Since there are existing structures of Sensors in Sality crawlers that can be leveraged, as in the Strobo Crawler 2.5, the crawler also has sensor functionality. Thus in the simulation each superpeer holds the crawler in its neighbourlist. Similarly the Receiver Module of the Strobo Crawler also has a high reputation and is known by most peers (CITE???) in the Sality botnet. Additionaly the crawler holds each superpeer in its neighbourlist and thus is able to contact each individual peer. This is equivalent to the Strobo Crawler knowing every superpeer in the real network. These conditions are highly optimized but similar in reality.

Internally the crawler holds two important but distinctive sets:

- V_{Eu} A set that maps each superpeer u to its last known seq_u and the timestamp at which this new URL pack was propagated by u.

- V_G A map of superpeer IDs to their crawler good counts.

In order to choose a set of potential botmaster candidates V_E after each new propagation of a URL pack, the crawler keeps its own good count table for superpeers, represented by the set V_G. Each propagation of a new pack increases the good count of superpeers that are likely to be the first ones receiving the pack and decreases the good count of all other peers. This enables the crawler to filter out botmaster peers dynamically even when new ones join the network.

The crawler is in one of three phases at any given time:

- **Standby phase** In the standby phase, the crawler acts as a sensor and additionally polls all superpeers every 6 seconds via hello messages. Thus it receives any messages from neighbouring MM cycles and additionaly the hello responses. Once the crawler receives a message with a new pack number, it enters the according data into V_{Eu} and enters the polling phase.
- **Polling phase** In the polling phase the crawler still acts as a sensor and now polls all superpeers repeadedly for 60 seconds. Once a superpeer responds with the newly measured seq_{max}, the according mapping is inserted into V_{Eu}. After the polling time limit ends, the evaluation phase is entered.
- **Evaluation phase** In the evaluation phase the good counts are adjusted. This can happen in different ways, depending on how the crawler is tweaked. For the base version each superpeer $u \in V_{Eu}$ that holds seq_{max} and is at maximum 9 seconds off from the first timestamp received in the standby phase increases its good count by one all other superpeers decrease their respective good counts by one. The necessary condition for this is that the botmaster propagates the URL pack at the same time towards all superpeers, which holds true in the simulation. The 9 seconds represent a maximum one directional network delay of three seconds. From the botmaster towards the superpeer and additionaly between crawler and superpeer. In the real network this value can be tweaked depending on the evaluation of botmaster behaviour.

Algorithm 1 displays a crawler cycle.

Algorithm 1: Crawler Algorithm

```
initSets()
phase = standby
cycleDelay = 6
while true do
    listenForMessages()
    if phase == standby then
        if newPackReceived then
            cycleDelay = 0
            phase = poll
            enterData(peerId, timestamp, packId, $V_{Eu}$)
            setPollEnd(60)
        end
        scheduleSuperpeerPoll(cycleDelay)
    else if phase == poll then
        scheduleSuperpeerPoll(cycleDelay)
        if newPeerEntry then
            enterData(peerId, timestamp, packId, $V_{Eu}$)
        end
    else
        calculateGoodCountChanges()
        cycleDelay = 6
        phase = standby
    end
end
```

First off all variables are initialized. At any time the crawler listens for incoming messages. If the crawler is in standby phase, it checks if it should send out hello messages as indicated by the method scheduleSuperpeerPoll(int delay). If during the standby phase a message has been received that contains a new URL pack the according data is entered via the enterData(int peerId, int timestamp, int packId, map peerMap) method and the polling phase is initialized. The method setPollEnd(int seconds) creates a callback that is executed after the time given and transitions from the polling into the evaluation phase. If the crawler is in the polling phase it constantly polls all superpeers, checks for new pack updates and enters new data. Once the time for the polling phase is up, the registered callback is triggered and the evaluation phase entered. Now all good count changes are applied. Afterwards the standby phase is entered again and the cycle is repeated.

Good count evaluation methods

Once a whole cycle from standby- to evaluation phase has finished the good counts can be evaluated. For this task in general different approaches can be taken:

- **Percentage evaluation** When choosing a fixed percentage x of superpeers to be known by the botmaster, the $x\%$ superpeers $u \in V_G$ with the highest good counts are chosen. This thesis does not follow this approach since the percentage of propagation the botmaster chooses is likely to be dynamic.

- **Ranged evaluation** Evaluation can also be done with a dynamic percentage of superpeers. In this case all superpeers $u \in V_G$ that match certain good count parameters can be chosen. One such parameter could be to only choose the superpeers with the absolute highest good count. However this implies that the botmaster uses a fixed sized set as in the neighbourlist strategy and likely does not work for the mixed strategy. The approach used in the thesis is to use a range of applicable good counts. This avoids sorting out relevant superpeers because they hold a slightly lower good count than the maximum. It is very likely, that good counts vary between botmaster peers because of measuring errors, network delay etc.

The superpeer evaluation done by the crawler follows the dynamic ranged pattern. This means that no percentage evaluation is done in this thesis. The idea behind the usage of a dynamic evaluation pattern is that the superpeer propagation percentage might change over time. In order to also factor in differences in good counts between two superpeers chosen by the botmaster not only the single highest good count is used for the selection of superpeers. Rather, the selection happens by calculating a target good count

As a measurement for which peers are considered to be in V_E at a given time, a target good count for the iteration i: $gc_t^{(i)}$ is calculated. Each superpeer u with a corresponding $gc_u \geq gc_t^{(i)}$ belongs to $V_E^{(i)}$. The target good count gc_t is calculated in a way, which ensures that every superpeers good count gets factored int in a biased way. This means that more relevant superpeers with a higher good count have a bigger influence on gc_t than less relevant ones.

For the calculation of target good counts the following approaches are used:

- **Geometric series averaging (GSA)** This approach factors in relevance of superpeers on the target good count via a geometric series:

$$gc_t^{(i)} := \left\lfloor \sum_{i=1}^{num_v} \frac{1}{2^i} \times V_{GC}^{(i)}[i] \right\rfloor$$

  The brackets denote and index operator starting at 1, also the whole operation is rounded down, as good counts are always integers. $V_{GC}^{(i)}$ represents a sorted version of $V_G^{(i)}$ descending by good count at iteration i. This ensures the correct weighting of superpeers. Since the sequence $\sum_{i=1}^{\infty} \frac{1}{2^i}$

converges towards 1, $gc_t^{(i)}$ is always in the range $[gc_{min}, gc_{max}]$, where gc_{min} is the minimum good count of any superpeer, gc_{max} the maximum. Each superpeer u with a $gc^{(u)} \geq gc_t^{(i)}$ is in $V_E^{(i)}$.

- **Maximum good count drop (MGCD)** This approach splits the superpeers into two distinctive sets based on their good counts. The first set gc_{high} holds superpeers that are later considered botmaster peers. The second set gc_{low} holds all other superpeers. The target good count is defined as the lowest good count of a peer in gc_{high}. The peers are split into the two sets at the maximum drop in good counts between any adjacent superpeers in $V_{GC}^{(i)}$. $V_{GC}^{(i)}$ is calculated as in the geometric series averaging. $gc_t^{(i)}$ is calculated as follows:

$$gc_t^{(i)} := gC(V_{GC}^{(i)}[j]) :$$

$$gC(V_{GC}^{(i)}[j]) - gC(V_{GC}^{(i)}[j+1]) \geq gC(V_{GC}^{(i)}[x]) - gC(V_{GC}^{(i)}[x+1]) \ \forall x \neq j$$

  $gC(V_{GC}^{(i)}[index])$ is a function that returns the good count of an entry of $V_{GC}^{(i)}$ at the given index.

- **Maximum good count drop averaging (MGCDA)** This method build on the MGCD method. Firstly the set gc_{high} of the MGCDA is calculated. Then the target good count is the mean of all good counts in gc_{high}:

$$gc_t^{(i)} := \frac{1}{size(gc_{high}^{(i)})} \sum_{j=1}^{size(gc_{high}^{(i)})} gC(gc_{high}^{(i)}[j])$$

The different crawler versions are run on variants of the neighbourlist and mixed botmaster strategies. For the neighbourlist one, simulations are run on botmaster distribution percentages from $10-50\%$. For the mixed strategy for peer selection percentages from $30-80\%$. This matches with the parameters of section 4.2. Each simulation run is repeated twice and the results are averaged to get meaningful data. The crawler logs updates of V_G. These logs are collected by the script 4, which outputs csv files that contain the relevant statistical metrics.

4.3.3 Simulation results

GSA

The GSA is strongly biased on more relevant superpeers. Since the decay of relevance of good counts rises exponentially, the influence of the fifth best superpeer on the target good count is already only at about 3%. This results in a very high target good count, that often matches the maximum good count.

Figure 14 displays the precision and recall of the GSA crawler on a botmaster using the neighbourlist strategy. It works quite well, since the botmaster peers do not change, which leads to a linear increase of good counts for these superpeers. Occasionally the crawler chooses superpeers that are not botmaster peers to be in V_E, which is visualized as precision drops in the figure. This happens once non-botmaster peers increase their good counts high enough to lower the target good count below a threshold, which leads to more non-botmaster peers joining V_E. However, since the likelihood of botmaster peers increasing their good count is far greater than the likelihood of non-botmaster superpeers, the more URL packs are propagated, the less likely these drops become. The general precision thus converges towards 100%. The recall is also always at 100% because of the high chance of botmaster peers increasing their good counts. If network- or measuring errors occur, it is possible that botmaster peers are dropped from V_E and the recall declines. The resulting set V_E contains about 95% botmaster peers with a recall of 100% on average after 20 URL pack propagations.

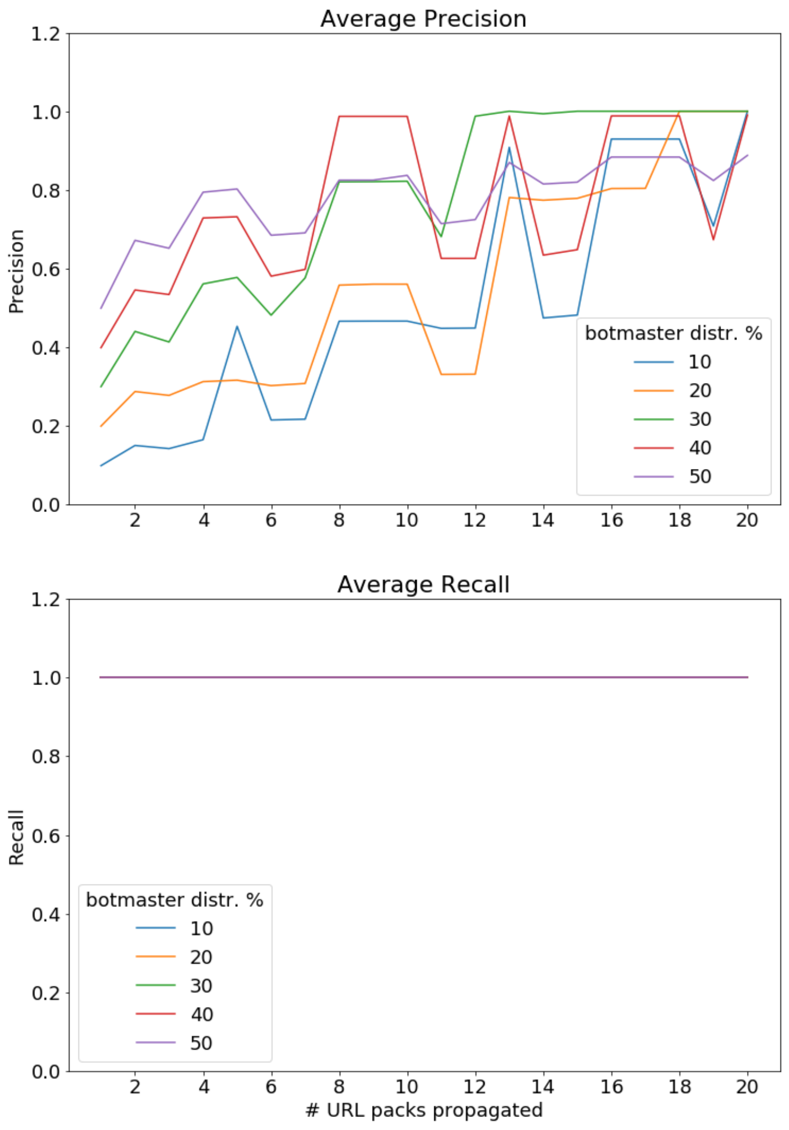


Figure 14: Precision and Recall of the GSA crawler for the neighbourlist strategy

Figure 15 displays the GSA crawlers precision and recall for the mixed strategy. In this case the previously mentioned relevance decay property of superpeers with a low good count on $gc_t^{(i)}$ influences the recall in a negative way. Since most of the botmaster peers change each propagation of a new URL pack, the good counts between them drift apart. The further these good counts are apart, the less likely botmaster peers with a slightly lower good count than $gc_{max}^{(i)}$ are to be selected by the crawler at propagation i. This is clearly seen in the general negative slopes in the recall. This effect is worse the smaller the botmaster distribution percentage is, which is due to the fact that a botmaster peer has a lower chance of being selected for the next propagation of a URL pack if the set of selected botmaster peers is smaller. The precision on the other hand rises in general, since not only botmaster peers, but also non-botmaster peers are sorted out quickly. In the end the resulting set V_E contains on average about 97% botmaster peers after 20 URL pack propagations, however it also only has a recall between $20 - 60\%$.

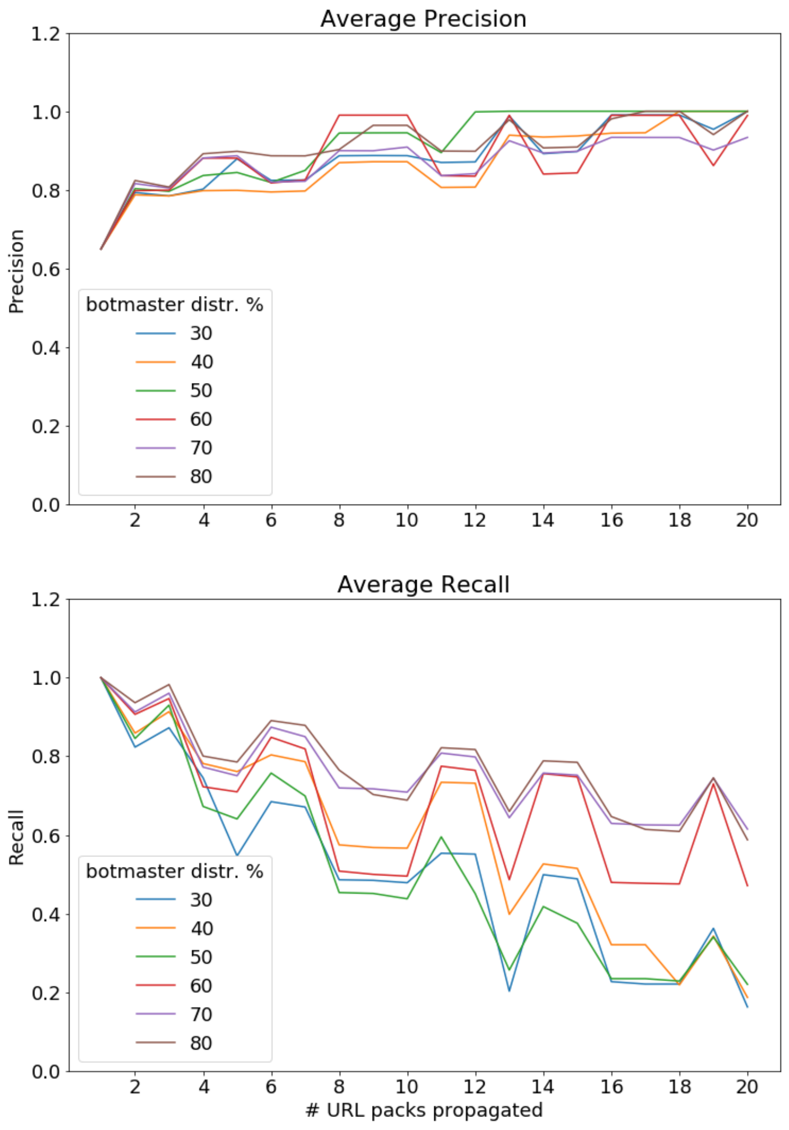


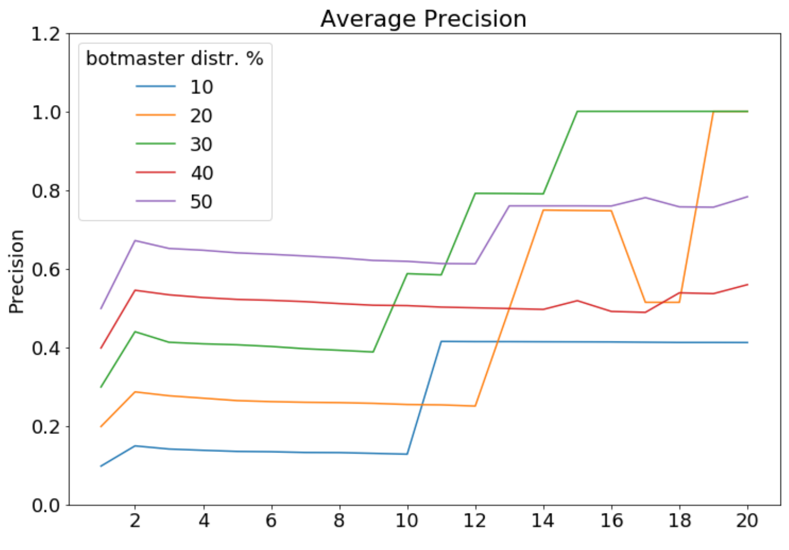
Figure 15: Precision and Recall of the GSA crawler for the mixed strategy

The geometric series averaging is a great dynamic evaluation method if the botmaster chooses the neigbourlist strategy and thus sticks with a certain set of superpeers. This method provides a slow but steady increase of precision, whilst likely always holding a recall of 100%. However, this upside on a steady set comes with the downside of loosing recall on dynamically chosen botmaster peers as well as being a slow method. On average 16 URL pack propagations are needed to stay at a precision of $> 80\%$.

MGCD

The MGCD is less biased than the GSA on relevant superpeers. This results in a generally lower $gc_t^{(i)}$ when comparing the number of propagations i to the GSA crawler.

Figure 16 displays the precision and recall of the MGCD crawler for the neighbourlist strategy. In general it takes longer to filter out non-botmaster peers from V_E than in the GSA crawler. This is due to the fact, that $gc_t^{(i)}$ is lower in nearly all occasions and more dynamic in general. Because of the lower target good count, it takes on average about 10 URL pack propagations to see an increase in precision. Then, similarly to the GSA crawler, the average precision rises slowly over time converging towards 100%. The recall also always is at 100%. The explanation for that is the same as for the GSA crawler: A higher likelihood in botmaster peers increasing their respective good counts leads to a slow but steady filtering out of non-botmaster peers. One advantage of this version is that the precision is not as affected by errors as in the GSA crawler. Since the maximum good count drop is the relevant factor for this evaluation strategy, the differences between botmaster peers good counts do not matter, as long as they are smaller than the ones between botmaster and non-botmaster peers. After 20 URL pack propagations the precision lies between $40 - 100\%$ and the recall is always at 100%.



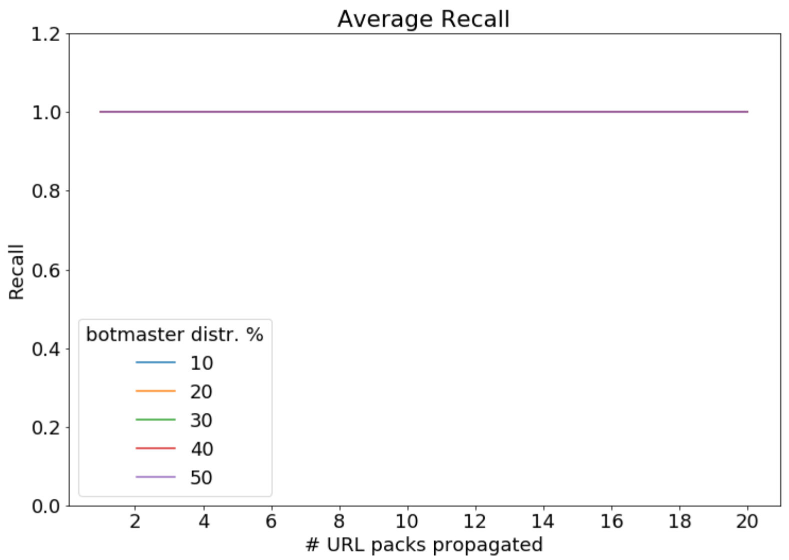


Figure 16: Precision and Recall of the MGCD crawler for the neighbourlist strategy

Figure 17 displays the precision and recall of the MGCD crawler for the mixed strategy. In this scenario the precision is on average between $75 - 80\%$. The recall drops to 80% after 3 URL pack propagations, but then converges towards 100% again. This is because on average the botmaster peers good counts early on are not too distinct from the non-botmaster peers and thus some are considered to be in gc_{low}. A lower botmaster distribution percentage comes with a larger loss of recall, since the chances are higher that botmaster peers have not yet been chosen by the botmaster. After about 5 URL pack propagations however nearly all botmaster peers tend to be in gc_{high}, since they increase their respective good counts on average more often than non-botmaster peers. The different botmaster peer sets that are chosen with each new propagation of a URL pack account for a larger difference in good counts between botmaster and non-botmaster peers. This leads to a stable precision between $75 - 80\%$ over all propagations. Aother positive aspect of this evaluation method is that the precision instantly rises to $70 - 80\%$ after the first propagation of a URL pack. This sudden rise comes at the expanse of loosing recall early on. After 20 URL pack propagations the precision lies around 75% whilst the recall is at 100%.

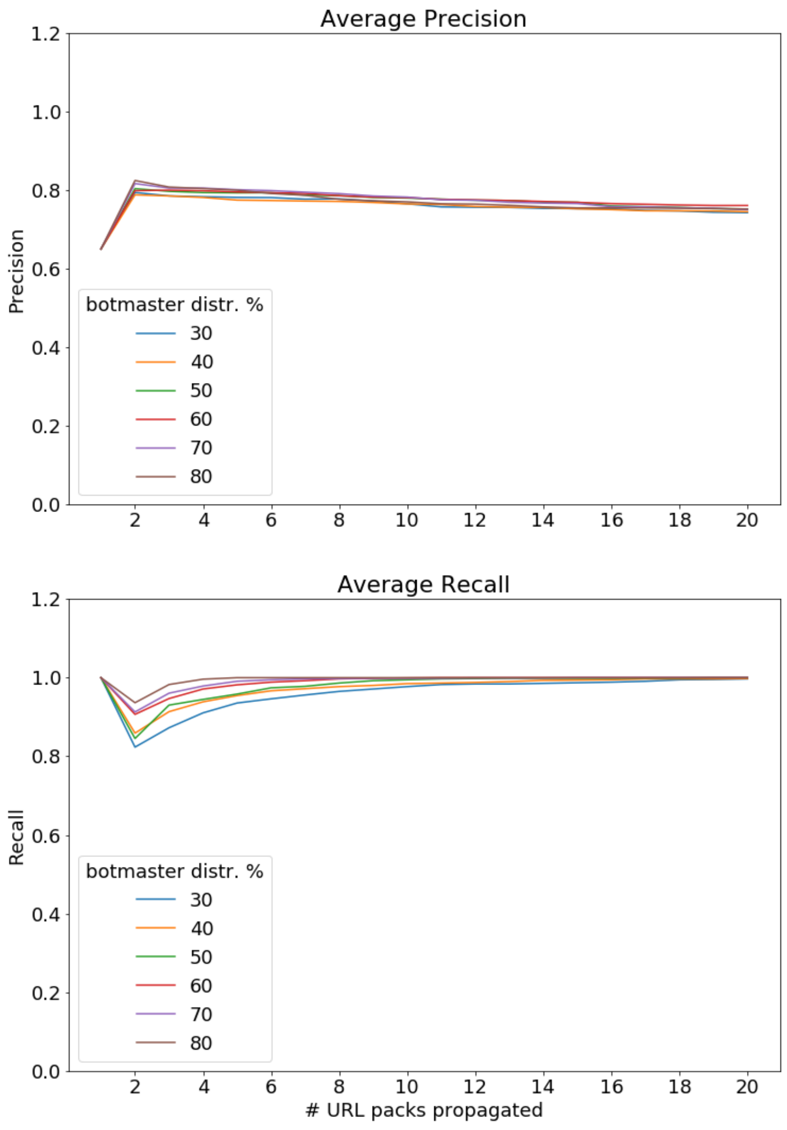


Figure 17: Precision and Recall of the MGCD crawler for the mixed strategy

Whilst the MGCD crawler is slow but consistent for the neighbourlist strategy, it is a good fit for the mixed strategy. The most notable effect lies in the unchanging precision this version puts out for the mixed strategy, that is independent of URL pack propagations and lies at about 75%, whilst still consistently staying at around 100% recall.

MGCDA

The thought of the MGCDA crawler is to combine the advantages of the GSA crawler for the neighbourlist strategy with the advantages of the MGCD crawler for the mixed strategy. Thus it is more biased on relevant superpeers than the MGCD crawler, but less than the GSA crawler. This potentially results in filtering out non-botmaster peers faster than the MGCD crawler.

Figure 18 displays the precision and recall of the MGCDA crawler for the neighbourlist strategy. Even though it performs slightly better than the regular MGCD crawler, no really notable improvements are visible. Thus it can be said that it is pretty much equal in this regard, averaging about the same precision and recall for each URL pack propagation number. This is due to the fact that the good counts of botmaster peers in gc_{high} for the neighbourlist are relatively similar, because they generally rise linearly without considering measuring errors. Taking an average value does not change gc_t much and thus does not provide a significant difference to the MGCD crawlers statistic. As with the MGCD crawler, after 20 URL pack propagations the precision lies between $40 - 100\%$ and the recall is always at 100%.

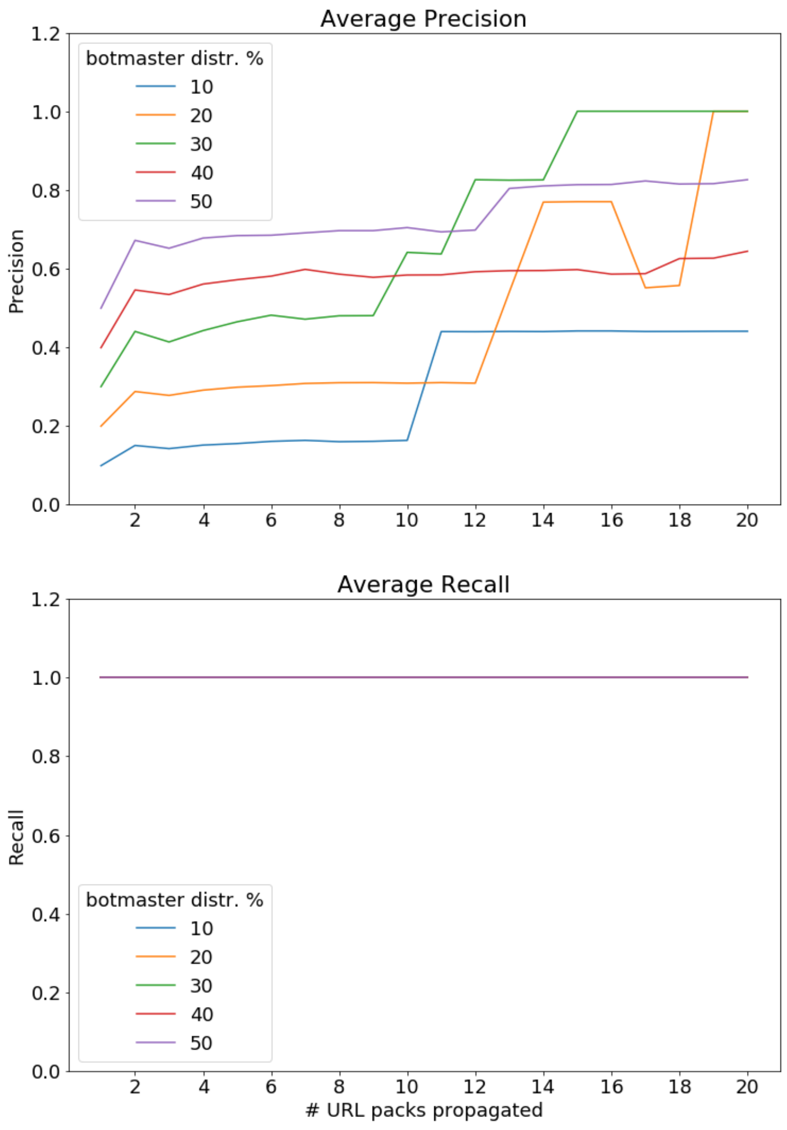


Figure 18: Precision and Recall of the MGCDA crawler for the neighbourlist strategy

Figure 19 displays the precision and recall of the MGCDA crawler for the mixed strategy. In this notable differences to the MGCD crawler are visible. While the precision is slightly higher on average, the recall converges towards an average value of only at about $80-90\%$. This is due to the fact that in comparison to the neighbourlist strategy, in the mixed strategy the botmaster peers good counts do not rise linearly. Whilst averaging over these good counts has nearly no effect when they are similar, in this case the effect is visualized in the drop of recall. The MGCDA crawler offers a general more stable and higher precision than the MGCD crawler, which comes at the price of a lower recall. A higher botmaster distribution percentage results in a higher recall, since botmaster peers have an even higher chance of increasing their good counts fast in relation to non-botmaster peers. After 20 URL pack propagations the precision lies between $80 - 100\%$ while the recall lies between $80 - 90\%$.

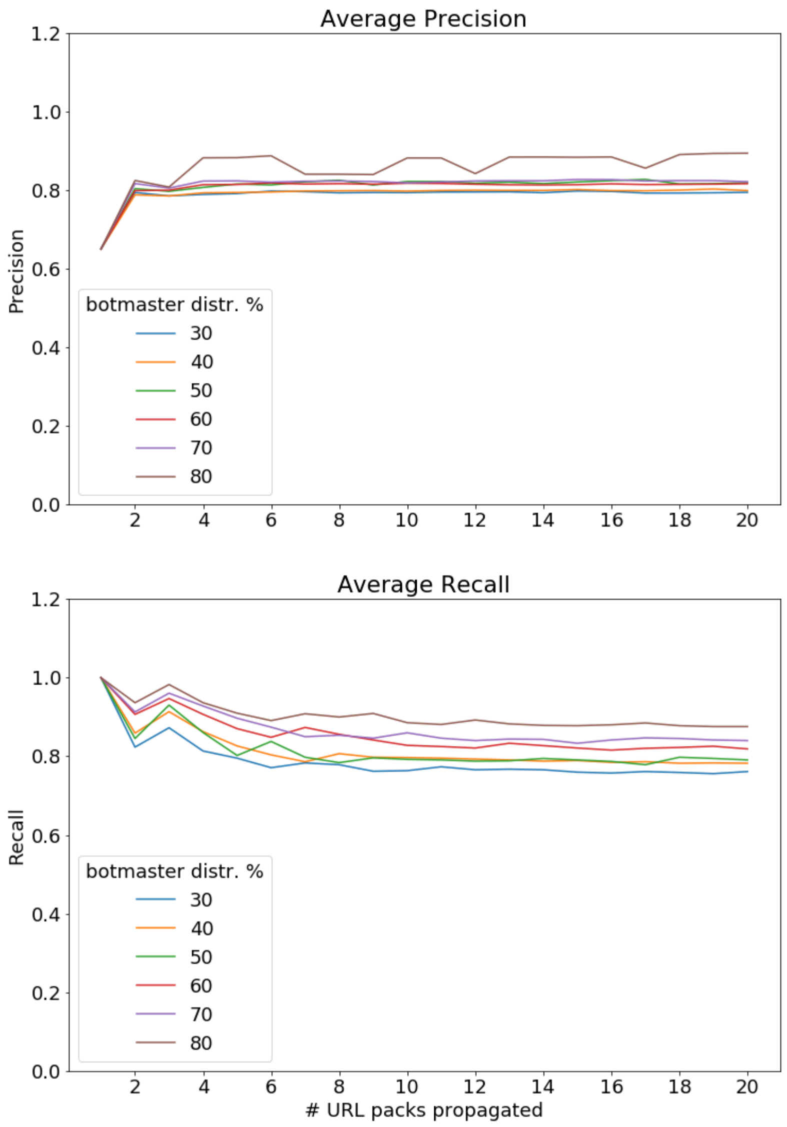


Figure 19: Precision and Recall of the MGCDA crawler for the mixed strategy

The MGCDA crawler provides no meaningful increase over the regular MGCD crawler. The slight performance increase in the neighbourlist strategy is outweighed by the recall loss in the mixed strategy. The averaging of good counts thus has no visible advantage where it is wanted, but a significant disadvantage where it is not.

5 Conclusion

This section briefly summarizes the results of the thesis and provides groundwork for further research in the area.

5.1 Results

The provided data of the Sality botnet architecture in form of recreated graphml files from Strobo Crawler messages was used to recreate the botnet in a simulation environment. The network topology was explored, showing that Sality contains a core set of superpeers that make up about 75% of the network. These superpeers are highly interconnected and spread URL packs fast towards each other. On the other hand the 25% left over superpeers are more isolated.

Different malware propagation strategies have been tested. The neighbourlist- and the mixed strategy resulted to be the most likely ones to be used by a botmaster. They are also the most probable ones to be used since they provide significant advantages whilst being easily maintainable. A botmaster that only monitors the core network of intertwined superpeers can achieve fast propagation while not having to monitor outliers.

Finally a set of crawlers has been specificially addressed the two resulting strategies to exploit them. The MGCD crawler results to be subjectively the most balanced one, averaging high precision and recall for both propagation strategies. It also achieves high precision for the mixed strategy within few URL pack propagations.

5.2 Future work

On the basis of this thesis the defined crawlers could be implemented for the real Sality botnet. One method to do so is to include them into the Strobo Crawler 2.5. Once the crawlers are up and running, the statistics can be evaluated to confirm the usage of one of the botmaster strategies 4.2 or find out about new ones. The output of these crawlers could be used to find a subset of superpeers that are possibly being used as entry points for malware propagation. The following section gives an overview of a potential inclusion of the crawlers into the Strobo Crawler.

Inclusion into the Strobo Crawler

In addition to the Prober- and Receiver Modules, a new Crawler Module (CM) could be added. This module implements the discussed crawling algorithms. The corresponding necessary sets of superpeers V_{Eu} and V_G are also added to the list of nodes. The set V_E, which makes up the found botmaster peers for a new URL pack propagation id logged to the database. Additionally it is connected to the RM and can access the list of online nodes V^O. V_{Eu}, V_G and V_E are always made up of peers in V^O.

Figure 20 displays the Strobo Crawler in addition of the CM. For the polling of the CM the PM is utilized. The RM is responsible to also forward messages towards the CM, which enables the Sensor functionality. If timeouts of superpeers occur at any given time, the corresponding nodes are moved to the set of offline nodes by the Strobo Crawler. Then they are also not considered by the CM, which only cares for online nodes and thus removed from V_{Eu} and V_G. The good counts of these superpeers should however be saved, since the reason of the peer going offline is unknown and could only be temporary. Thus for V_{Eu} and V_G two additional sets V_{Eu}^{Off} and V_G^{Off} are added to the list of nodes. Once they come back online, they are reconsidered by the CM, starting at their latest state. The different states of the crawler explained in section 4.3 are also be implemented. Once the RM receives a new URL pack number for the first time outside of a crawling cycle, the CM enters the polling phase in which the PM is used. In the evaluation phase, the corresponding sets are updated and V_E is logged to the database.

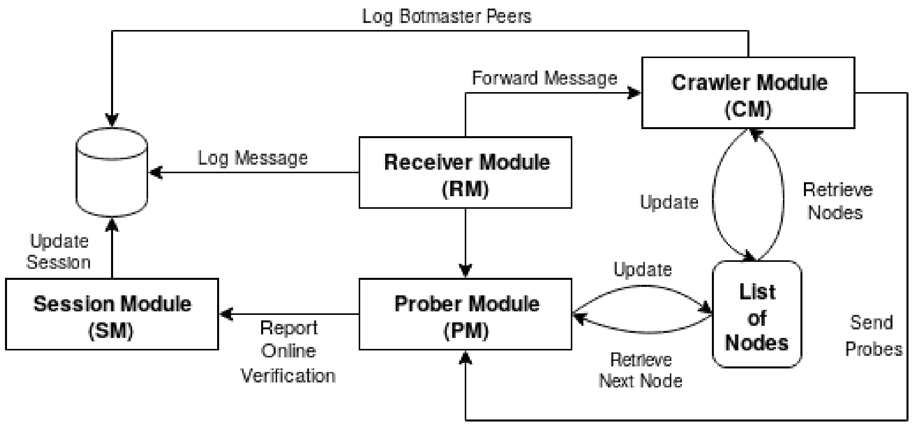


Figure 20: Strobo Crawler with integrated Crawler Module

Bibliography

References

[1] M. Abu Rajab, J. Zarfoss, F. Monrose, and A. Terzis, "A multifaceted approach to understanding the botnet phenomenon," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pp. 41–52, ACM, 2006.

[2] S. Karuppayah, *Advanced Monitoring in P2P Botnets. A Dual Perspective.* Springer, 2018.

[3] J. B. Grizzard, V. Sharma, C. Nunnery, B. B. Kang, and D. Dagon, "Peer-to-peer botnets: Overview and case study.," *HotBots*, vol. 7, pp. 1–1, 2007.

[4] M. Mahmoud, M. Nir, A. Matrawy, *et al.*, "A survey on botnet architectures, detection and defences.," *IJ Network Security*, vol. 17, no. 3, pp. 264–281, 2015.

[5] B. B. Kang, E. Chan-Tin, C. P. Lee, J. Tyra, H. J. Kang, C. Nunnery, Z. Wadler, G. Sinclair, N. Hopper, D. Dagon, *et al.*, "Towards complete node enumeration in a peer-to-peer botnet," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pp. 23–34, ACM, 2009.

[6] D. Andriesse, C. Rossow, and H. Bos, "Reliable recon in adversarial peer-to-peer botnets," in *Proceedings of the 2015 Internet Measurement Conference*, pp. 129–140, ACM, 2015.

[7] C. Rossow, D. Andriesse, T. Werner, B. Stone-Gross, D. Plohmann, C. J. Dietrich, and H. Bos, "Sok: P2pwned-modeling and evaluating the resilience of peer-to-peer botnets," in *2013 IEEE Symposium on Security and Privacy*, pp. 97–111, IEEE, 2013.

[8] N. Falliere, *Sality: Story of a Peer-to-Peer Viral Network.* Symantec Corporation, 2011.

[9] S. Haas, S. Karuppayah, S. Manickam, M. Mühlhäuser, and M. Fischer, "On the resilience of p2p-based botnet graphs," in *2016 IEEE Conference on Communications and Network Security (CNS)*, pp. 225–233, IEEE, 2016.

[10] A. Virdis, *Recent Advances in Network Simulation: The OMNeT++ Environment and its Ecosystem.* Springer.

[11] A. Varga *et al.*, "Omnet++ user manual, version 5.5," *https://doc.omnetpp.org/omnetpp/manual/*, accessed 2019.

List of figures

List of tables

Acronyms

P2P Peer to Peer

C2 Command & Control

DDoS Distributed Denial of Service

DGA Domain Generation Algorithm

MM Membership Maintenance

NAT Network Adress Translation

OMNeT++ Objective Modular Network Testbed in C++)

Glossary

Botmaster Person in control of the botnet. Can propagate malware throughout the network to be executed.

Botnet Set of compromised machines connected to the internet. These computers carry out malicious commands from the botmaster.

Bot Infected machine and part of the botnet, that carries out attacks of the botmaster.

Peer Synonym to bot.

Crawler Entities that traverse the botnet in order to discover bots.

Sinkholing Redirecting traffic over a controlled server.

Entry Point Superpeers, that the botmaster contacts in order to distribute new malware in a P2P botnet.

Superpeer A bot in a P2P botnet, that is routable and can thus exchange neighbourlist information.

Neighbourlist A list each superpeer in a P2P botnet owns. It contains information about other superpeers that can be contacted.

URL pack A message spread by the botmaster in the Sality botnet. It contains links to servers that hold new malware for the bots to execute.

Sequence number The number uniquely identifying a URL pack version.

LastOnline The timestamp for a neighbour in a neighbourlist of a bot in the botnet Sality, that states when the neighbour was successfully probed the last time.

GoodCount A value for a neighbour in a neighbourlist of a bot in the botnet Sality, that states how reliable the neighbour is. This depends on how many succesfull responses he has given.

Sensor A peer of a botnet that evaluates the network traffic and peer behaviour. The goal is to make the IP of a sensor known to all peers, such that the whole communication can be analyzed.

Node churn The rate at which peers join or leave the botnet. If this rate is high, the network infrastructure changes often.

Source code

Source Code 1: Running an initialization configuration

```
#!/bin/bash

opp_runall -j6 ./run -m -u Cmdenv -c $1
```

Source Code 2: Converting a graphml file to a NED file

```
import click
import re
import collections

@click.command()
@click.option('--path', help='Path to GraphML file.',
 ↪  required=True)
@click.option('--output', default='Sality.ned', help='Path to
 ↪  output NED file.')
@click.option('--pb', default=2, help="""Version of the
 ↪  botmaster used:
```

```
1: Passive, 2: Active.""")
@click.option('--ce', default=0, help="""Enable/Disable creation
↪  of a crawler:
0: No crawler, 1: Crawler created.""")
@click.option('--se', default=0, help="""Enable/Disable sensor.:
0: No sensor, 1: Sensor created.""")

def main(path, output, ce, se, pb):
    edges = {}

    extractEntities(path, edges) # Extract all nodes and Edges
    ↪  from the Graphml file
    edges = collections.OrderedDict(sorted(edges.items(), key =
    ↪  lambda kv: len(kv[1]), reverse=True)) # Sort by number
    ↪  of connections
    peers = list(edges.keys()) # peers also are sorted by number
    ↪  of connections, index 0 == most connections.

    nedFile = open(output,"w+")
    writePreamble(nedFile, len(peers), ce)
    writeConnections(nedFile, peers , edges, ce, se)
    writePostamble(nedFile)

    nedFile.close()

def extractEntities(path, edges):
    with open(path) as graphFile:
        for line in graphFile:
            line = line.strip()
            m = re.search(r"<edge source=\"(\w+)\"
            ↪  target=\"(\w+)\" />", line)
            if m:
                edgeSource = m.group(1)
                edgeTarget = m.group(2)
                if not edgeSource in edges:
                    edges[edgeSource] = set()
                if not edgeTarget in edges:
                    edges[edgeTarget] = set()
                edges[edgeSource].add(edgeTarget)

def writePreamble(nedFile, numPeers, ce):
    crawlerImport = "import sality.ned_files.Crawler;\n" if ce
    ↪  == 1 else ""
    crawlerCreation = "\t\tcrawler: Crawler;\n" if ce == 1 else
    ↪  ""
    preamble = "package sality.ned_files;\n\n" + \
        "import sality.ned_files.Superpeer;\n" + \
        crawlerImport + \
        "import sality.ned_files.Botmaster;\n\n" + \
        "network Sality\n{\n\ttypes:\n\t\tchannel Channel
        ↪  extends ned.DelayChannel\n" + \
        "\t\t{\n\t\t\tdelay = 50ms;\n\t\t}\n\tsubmodules:\n" + \
        "\t\tpeer["+str(numPeers)+"]: Superpeer;\n" + \
        crawlerCreation + \
        "\t\tbotmaster: Botmaster;\n" + \
        "\tconnections:\n"

    nedFile.write(preamble)

def writeConnections(nedFile, peers, edges, ce, se):
    for source, targets in edges.items():
        sourceString = generateConnectionString(source, peers)

        for target in targets:
            targetString = generateConnectionString(target,
            ↪  peers)
            nedFile.write("\t\t%s.outputGate++ <--> Channel <-->
            ↪  %s.inputGate++;\n" %(sourceString,
            ↪  targetString))

    if ce == 1:
        generateGodmodeConnection(nedFile, "crawler",
        ↪  len(peers))
    if se == 1:
        generateSensor(nedFile, len(peers))
    generateGodmodeConnection(nedFile, "botmaster", len(peers))

def generateConnectionString(node, peers):
    return ("peer[%d]" % (peers.index(node)))

def generateGodmodeConnection(nedFile, entity, numPeers):
    for i in range(numPeers):
        nedFile.write("\t\t%s.outputGate++ <--> Channel <-->
        ↪  peer[%d].inputGate++;\n" %(entity, i))

def generateSensor(nedFile, numPeers):
    for i in range(numPeers):
        nedFile.write("\t\tpeer[%d].outputGate++ <--> Channel
        ↪  <--> crawler.inputGate++;\n" %(i))

def writePostamble(nedFile):
    nedFile.write("}\n")

if __name__ == '__main__':
    main()
```

Source Code 3: Extract Salitys propagation statistics from Strobo Crawler messages

```
import click
import pandas as pd
import numpy as np
import datetime
import csv
import re
from os import listdir
from os.path import isfile, join

@click.command()
@click.option('--directory', default='messages/',
↪  help='Directory of the message file')
#@click.option('--seqNum', required=True, help='The sequence
↪  number of the URL Pack to analyze')

def main(directory):
    # seqNum, time
    distributionTimes = {}
    # seqNum, dataframe
    dataframes = {}

    # initialize dicts
    for i in range(2, 101, 2):
        distributionTimes[i] = list()
    distributionTimes[0] = [0]

    extractDataframes(dataframes, directory)

    for seqNum, df in dataframes.items():
        extractPackPropagation(seqNum, df, distributionTimes)

    with open('results/sality/V3packDistribution.csv', 'w') as
    ↪  csvfile:
        filewriter = csv.writer(csvfile, delimiter=',',
                                quotechar='|',
                                ↪  quoting=csv.QUOTE_MINIMAL)
        filewriter.writerow(['Percentage', 'Propagation Time',
         ↪  'Min Delay', 'Max Delay'])

        for i in range(0, 101, 2):
            minDelay = min(distributionTimes[i])
            maxDelay = max(distributionTimes[i])
            meanDelay = np.mean(distributionTimes[i])
            filewriter.writerow([i, meanDelay, minDelay,
             ↪  maxDelay])

'''
Extracts all message logs of URL pack distributions into
↪  dataframes.
'''
def extractDataframes(dataframes, directory):
    for f in listdir(directory):
        filePath = join(directory, f)

        if not isfile(filePath):
            continue

        m = re.search(r"messages_for_v3_pack_(\d+).csv", f)
        seqNum = m.group(1)
        df = pd.read_csv(filePath, sep='\t')
        dataframes[seqNum] =  cleanData(df, int(seqNum))

'''
Drops unnecessary columns from dataframe and sorts new pack
↪  arrival by timestamp for IPs.
Returns the new cleaned dataframe.
'''
def cleanData(df, packid):
    df = df.loc[df['PackID'] == packid]
    df = df.loc[df['NodeType'] == 'server']
    df.drop(columns=['Port', 'PackID', 'CMD', 'NodeType'],
     ↪  inplace=True)
    df.sort_values(by=['Timestamp'], inplace=True)
    df.drop_duplicates(inplace=True, keep='first', subset='IP')

    return df

'''
Extracts propagation statistics for a singular URL pack.
'''
def extractPackPropagation(seqNum, df, distributionTimes):
    numPeers = df.shape[0]
    startTime = datetime.datetime.strptime(df.iloc[0, 0],
     ↪  '%Y-%m-%d %H:%M:%S')
    print('Sequence number: ', seqNum)
    print('\tTotal number of peers: ', numPeers)
    print('\tPack first seen: ', startTime)

    for i in range(2, 101, 2):
        index = int(i / 100 * numPeers) - 1

        currentTime = datetime.datetime.strptime(df.iloc[index,
         ↪  0], '%Y-%m-%d %H:%M:%S')
        distributionTimes[i].append((currentTime -
         ↪  startTime).total_seconds())

if __name__ == '__main__':
    main()
```

Source Code 4: Extract crawler logs and create statistics csv files

```
import click
import re
import numpy as np
import csv
import collections

from os import listdir
from os.path import isfile, join

@click.command()
@click.option('--path', default='../simulations/results',
    help='Path to OMNeT++ results directory, that contains the
     ↪  log files.')
@click.option('--evalversion', default=1,
    help='''The version of the good count evaluation method.
    1: Geometric series averaging
    2: Maximum good count drop
    3: Maximum good count drop averaging''')

def main(path, evalversion):
    filesDict = {}

    extractFiles(filesDict, path)
    resultsDict = extractStatistics(filesDict, evalversion)
    resultsDict = averageResults(resultsDict)
    createCSVs(resultsDict, evalversion)
```

```
def averageResults(resultsDict):
    averagedDict = {}

    for runName, resultsList in resultsDict.items():
        averagedDict[runName] = {}

        for packNum, statisticsDict in resultsList[0].items():
            if not packNum in resultsList[1] or not packNum in
            ↪  resultsList[2]:
                continue

            averagePrecision = statisticsDict['precision']
            averageRecall = statisticsDict['recall']
            for i in range(1, 3):
                averagePrecision +=
                ↪  resultsList[i][packNum]['precision']
                averageRecall +=
                ↪  resultsList[i][packNum]['recall']
            averagePrecision /= 3
            averageRecall /= 3

            averagedDict[runName][packNum] = {'precision':
            ↪  averagePrecision, 'recall': averageRecall}

    return averagedDict

def extractFiles(filesDict, path):
    for f in listdir(path):
        filePath = join(path, f)

        if not isfile(filePath):
            continue

        m = re.search(r"Crawler-PS(\d+)-(\d+)-\#(\d+).out", f)
        if m:
            peerSelectVersion = m.group(1)
            distributionPercentage = m.group(2)

            runName = 'Crawler-PS' + peerSelectVersion + '-' +
            ↪  distributionPercentage

            if not runName in filesDict:
                filesDict[runName] = []

            filesDict[runName].append(filePath)
```

```
def extractStatistics(filesDict, evalVersion):
    resultsDict = {}
    for runName, runFiles in filesDict.items():
        resultsDict[runName] = []
        for runFile in runFiles:

            ↪  resultsDict[runName].append(extractFileStatistics(runFile,
            ↪  evalVersion))

    return resultsDict

def extractFileStatistics(runFile, evalVersion):
    botmasterPeerList = []
    statisticsDict = {}

    with open(runFile) as f:
        for line in f:
            m = re.search(r"GoodCounts after pack:
            ↪  (\d+):       (((\d+:\d+),)+)", line)
            if m:
                packNum = m.group(1)
                peerList = list(m.group(2).split(','))
                goodCountDict = extractGoodCounts(peerList)
                foundPeers = extractFoundPeers(goodCountDict,
                ↪  evalVersion)

                extractCrawlerStatistics(foundPeers,
                ↪  botmasterPeerList, statisticsDict, packNum)
            else:
                m = re.search(r"botmaster peers:((\d+,)+)",
                ↪  line)
                if m:
                    botmasterPeerList =
                    ↪  list(m.group(1).split(','))
                    botmasterPeerList.remove('')

    return statisticsDict

def extractGoodCounts(peerList):
    goodCountDict = {}

    for entry in peerList:
        regM = re.search(r"(\d+):(\d+)", entry)
        if regM:
            peerId = regM.group(1)
            goodCount = int(regM.group(2))
            goodCountDict[peerId] = goodCount

    return goodCountDict

def extractCrawlerStatistics(foundPeers, botmasterPeerList,
 ↪  statisticsDict, packNum):
    statisticsDict[packNum] = {}
    numberRightPeers = len(list(
        filter(
            (lambda x: x in botmasterPeerList),
            foundPeers
        )
    ))

    statisticsDict[packNum]['precision'] =
     ↪  float(numberRightPeers) / len(foundPeers)
    statisticsDict[packNum]['recall'] = float(numberRightPeers)
     ↪  / len(botmasterPeerList)

def extractFoundPeers(goodCountDict, evalVersion):
    foundPeers = []

    sortedGoodCounts = collections.OrderedDict(
        sorted(goodCountDict.items(), key=lambda kv: kv[1],
         ↪  reverse=True)
    )

    targetGoodCount = 0

    if evalVersion == 1:
        targetGoodCount = goodCountGSA(sortedGoodCounts)
    elif evalVersion == 2:
        targetGoodCount = goodCountDrop(sortedGoodCounts)
    elif evalVersion == 3:
        targetGoodCount =
         ↪  goodCountDropAveraging(sortedGoodCounts)

    for peerId, goodCount in sortedGoodCounts.items():
        if goodCount >= targetGoodCount:
            foundPeers.append(peerId)

    return foundPeers

def goodCountDropAveraging(sortedGoodCounts):
    prevGoodCount = 0
    maxDrop = 0
    lastPeerId = 0

    # danach averagen mit den übrigen superpeers???
    for peerId, goodCount in sortedGoodCounts.items():
        if prevGoodCount == 0:
            prevGoodCount = goodCount
            continue

        drop = prevGoodCount - goodCount
        if drop >= maxDrop:
            maxDrop = drop
            lastPeerId = peerId

        prevGoodCount = goodCount

    targetGoodCount = 0
    count = 0
    for peerId, goodCount in sortedGoodCounts.items():
        if peerId == lastPeerId:
            break

        targetGoodCount += goodCount
        count += 1

    return int(targetGoodCount / count)

def goodCountDrop(sortedGoodCounts):
    prevGoodCount = 0
    maxDrop = 0
    lastGoodCount = 0

    # danach averagen mit den übrigen superpeers???
    for goodCount in sortedGoodCounts.values():
        if prevGoodCount == 0:
            prevGoodCount = goodCount
            continue

        drop = prevGoodCount - goodCount
        if drop >= maxDrop:
            maxDrop = drop
            lastGoodCount = prevGoodCount

        prevGoodCount = goodCount

    return lastGoodCount
```

```
def goodCountGSA(sortedGoodCounts):
    targetGoodCount = 0

    for goodCount in sortedGoodCounts.values():
        targetGoodCount += (float(goodCount) / 2 ** delta)
        delta += 1

    return int(targetGoodCount)

def createCSVs(resultsDict, evalVersion):
    for runName, results in resultsDict.items():
        evaluationString = ''
        if evalVersion == 1:
            evaluationString ='GSA'
        elif evalVersion == 2:
            evaluationString = 'DROP'
        elif evalVersion == 3:
            evaluationString = 'MGCDA'

        with open('results/simulation/' + runName + '-' +
         ↪  evaluationString + '.csv', 'w') as csvfile:
            filewriter = csv.writer(csvfile, delimiter=',',
                                    quotechar='|',
                                     ↪  quoting=csv.QUOTE_MINIMAL)
            filewriter.writerow(['PackNum', 'Precision',
             ↪  'Recall'])

            for packNum, statisticsDict in results.items():
                filewriter.writerow([packNum,
                 ↪  statisticsDict['precision'],
                 ↪  statisticsDict['recall']])

if __name__ == '__main__':
    main()
```