

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 in the past years. The possibility of infecting and using computers connected to the internet arbitrarily enables attackers to carry out all kinds of coordinated attacks, or simply use the machines for other malicious activities. The execution of malware (malicious software) on a confiscated machine makes any attack that can be written in code possible. With the newer and more resilient *P2P* botnet architectures, these threats have increased even further. 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 crawling the P2P botnet Sality. It analyzes the communication protocols, as well as the general architecture to find a way of traversing the botnet towards the root. This is done by implementing a sophisticated simulation of the network to analyze different malware distribution methods the *botmaster* could potentially use and find *crawlers* to exploit these methods. These crawlers should be tested on the real network in future research.
-INSERT SUCCESSFUL CRAWLER HERE-

Contents

1	Introduction	3
1.1	Motivation	3
1.2	Outline	4
2	Requirements and related work	4
2.1	Technical requirements	4
2.2	Related work	5
2.2.1	Botnets	5
2.2.2	Crawlers	6
2.2.3	Anti-crawling techniques	7
2.2.4	Sensors	7
2.2.5	Sality	8
2.2.6	Strobo crawler	9
3	Simulation design	11
3.1	Overview	11
3.2	Steps	14
4	Evaluation	16
4.1	Botmaster strategies	16
4.1.1	Distribution methods	16
4.1.2	Evaluation parameters	17
4.1.3	Results	18
4.2	Crawlers	27
4.2.1	Crawler versions	27
4.2.2	Evaluation parameters	28
4.2.3	Results	29
4.3	Summary	29
5	Conclusion	29
5.1	Results	29
5.2	Future work	29
	Bibliography	30
	List of figures	31
	List of tables	31
	Acronyms	31
	Glossary	31
	Source code	32

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. Additionally 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.

2 Requirements and related work

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

2.1 Technical 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.

2.2 Related work

2.2.1 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].

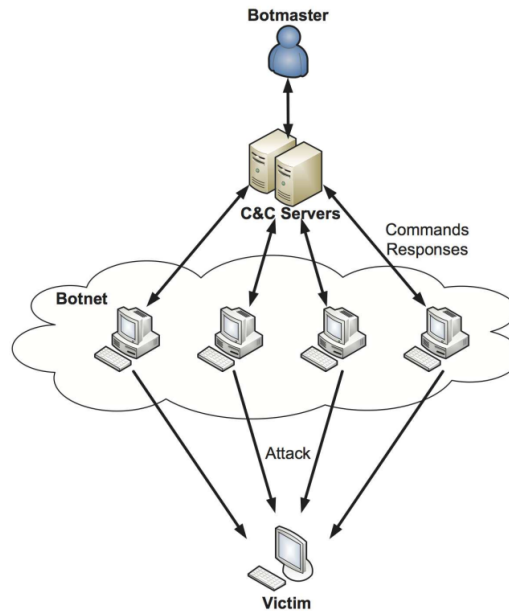


Figure 1: Centralised botnet architecture [4]

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 neighbours, once its neighbourlist reaches a low threshold of entries. This is done by contacting reliable neighbours and polling from their neighbourlist [2].

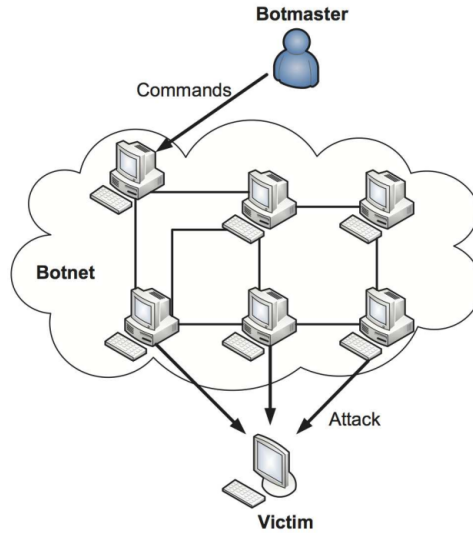


Figure 2: P2P botnet architecture [4]

2.2.2 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 Address 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].

2.2.3 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 addresses, that are known to monitor botnets. Alternatively some botnets just start a DDoS attack on the IP monitoring node [2].

Salinity 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 Salinity is able to send neighbourlist requests continuously to a peer until it converges towards the set of neighbours [2].

2.2.4 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.

2.2.5 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 resilient 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].

2.2.6 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 displays 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.

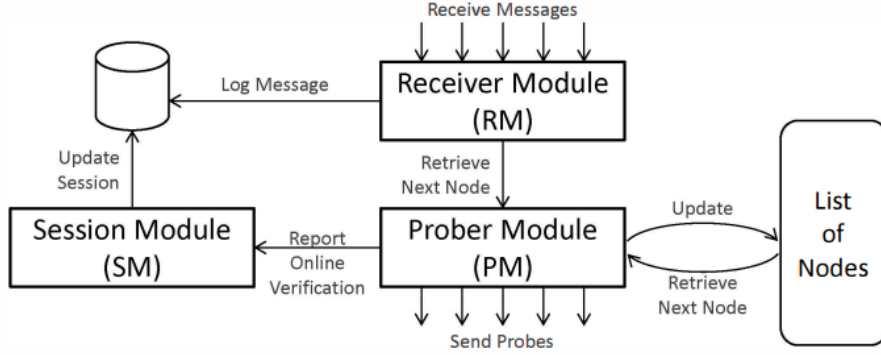


Figure 3: Strobe Crawler introduced in [9]

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 constrained 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 Strobe 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 2.2.5, only these peers can be retrieved.

3 Simulation design

This chapter describes the system design of the simulation environment. Firstly a brief overview of the botnet is given. Afterwards the individual entities 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 4.1.1.
- **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 2.2.6. These superpeers behave conform to the Sality protocol explained in 2.2.5.
- **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 4.2.1.

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 2.2.5 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.

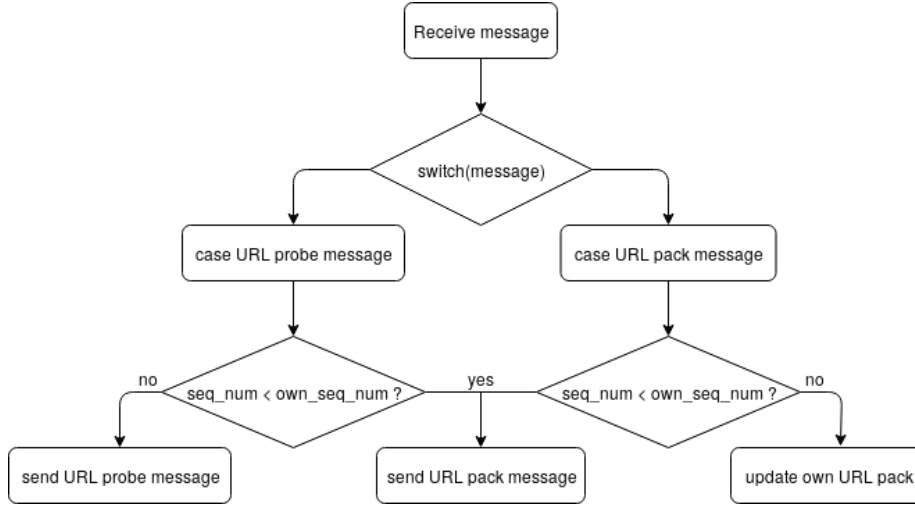


Figure 5: Superpeer behaviour on receiving a message

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 was found out to fit existing network behaviour very well.

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 default 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 4.2. 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 4.1 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 4.2 to better simulate real crawling behaviour.

3.2 Steps

To run a simulation of the Sality botnet, first off the network topology has to be established. For this task, the output of the Strobo crawler 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 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 additionally introduces a botmaster to the topology and optionally a crawler.

Once a network structure has been established, different simulations can be run depending on the botmaster strategy and crawler version of interest. This is done using a shell script (runSimulation.sh) and supplying the wanted version as the first argument. Versions are of the following format: $V\{number\}$, where *number* denotes the simulation to be run. More information about the run files and simulation to number mappings can be found in the projects READMEs.

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. Different scripts have been written to further analyze botmaster and crawler behaviour.

4 Evaluation

4.1 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 Strobe Crawler 2.2.6 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.

4.1.1 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 2.2.5.
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.
- **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 2.2.6.
- **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.

4.1.2 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.
- *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.
- Active botmasters: *sim_time_limit* = {15768000s, 31536000s}, *distribution_percentage* = {10, 20, 30, 40, 50}. Both active botmaster distribution methods are evaluated as the crossproduct of the *sim_time_limit* and *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.

In order to retrieve meaningful statistics given certain hyperparameter settings, each simulation is run multiple times with different seeds. This affects the

random propagation and release times of URL packs, such that each run yields different results. When also considering the different *peer_selection* strategies, this results in a total of $3 \times (3 \times (10 + 10) + 1) = 183$ 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:

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

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:

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

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 .

4.1.3 Results

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.

Both in the real Sality network and the simulation all URL packs are eventually propagated, without one being missed out by certain peers, which means that $message_loss = 0$ for all simulation runs. This indicates that the superpeers form a fully connected graph, such that all messages eventually are propagated to all peers. Haas et al. [9] also pointed out, that the Sality botnet is rather dense. Peers either know most other superpeers, or nearly none if they just joined. This is probably a result of the long intervals between MM cycles.

Figure 8 displays the propagation statistics for the passive approach. The data does not fit Salitys statistics very well, especially the average/maximum URL pack delay are off. However, on the minimum delay, a drastic jump is visible from 5 – 60% distribution percentage that is probably due to the simultaneous overlapping MM cycles of multiple superpeers. This result 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 basically lead to the botmaster being the C2 server, which could simply be traced and sinkholed. Thus, the passive botmaster does not seem to be the strategy followed. The following sections display and interpret the results of the different *peer_selection* strategies for the active botmasters.

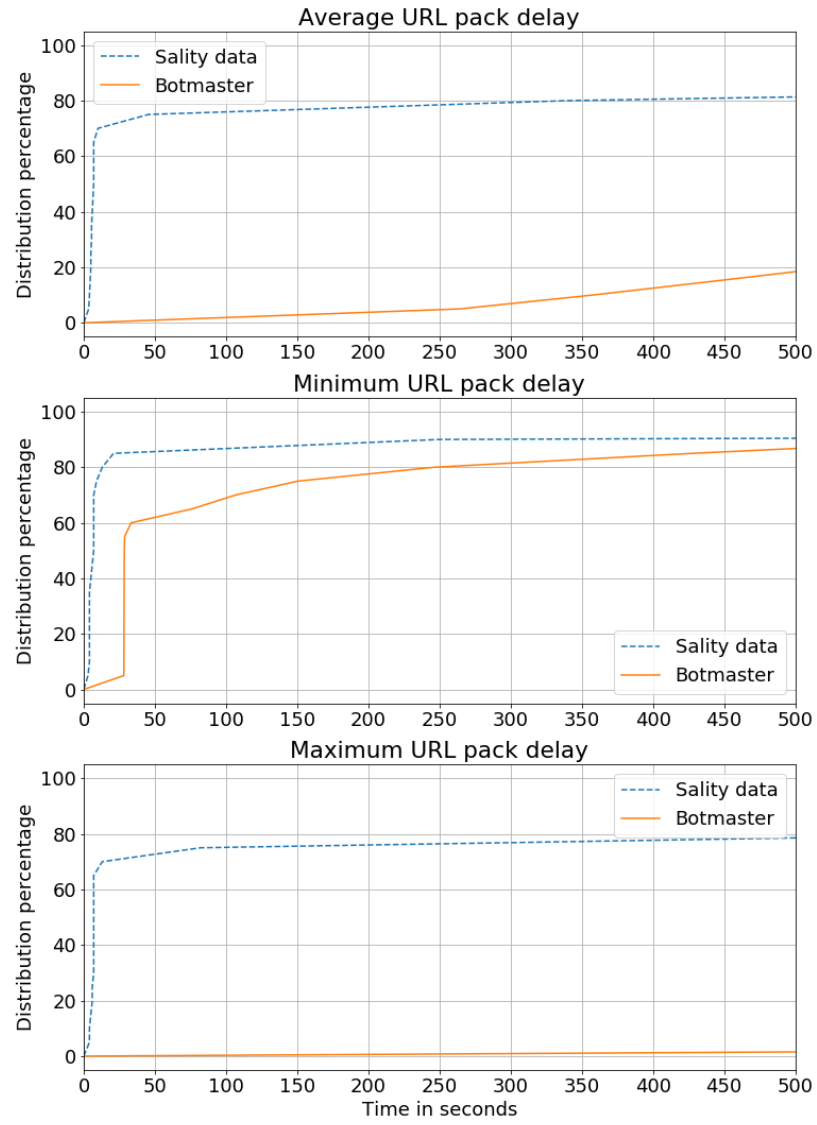


Figure 8: Distribution statistics of the PB

Random selection

The figure 9 shows the statistics for the AB2 method with different values for *distribution_percentage*. These curves approximate the real data very well. Especially the minimum URL pack delay seems to follow Salitys distribution statistics for nearly all different distribution percentages. It is noteworthy, that in the Sality network approximately 70% of the superpeers receive the new malware in less than a minute. Afterwards the time towards 100% propagation rises exponentially. This could be due to the above referenced Sality density and connection attributes. The closely connected superpeers probably 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 the new pack. This could mean, that the Sality network consists of approximately 70% closely connected and 30% outlier superpeers. This leaves the question on how many peers the botmaster directly contacts to propagate the new malware. According to the average URL pack delay, the 30% curve seems to fit very well. This could likely be the case if the botmaster itself utilizes a botnet monitoring mechanism to follow node churn.

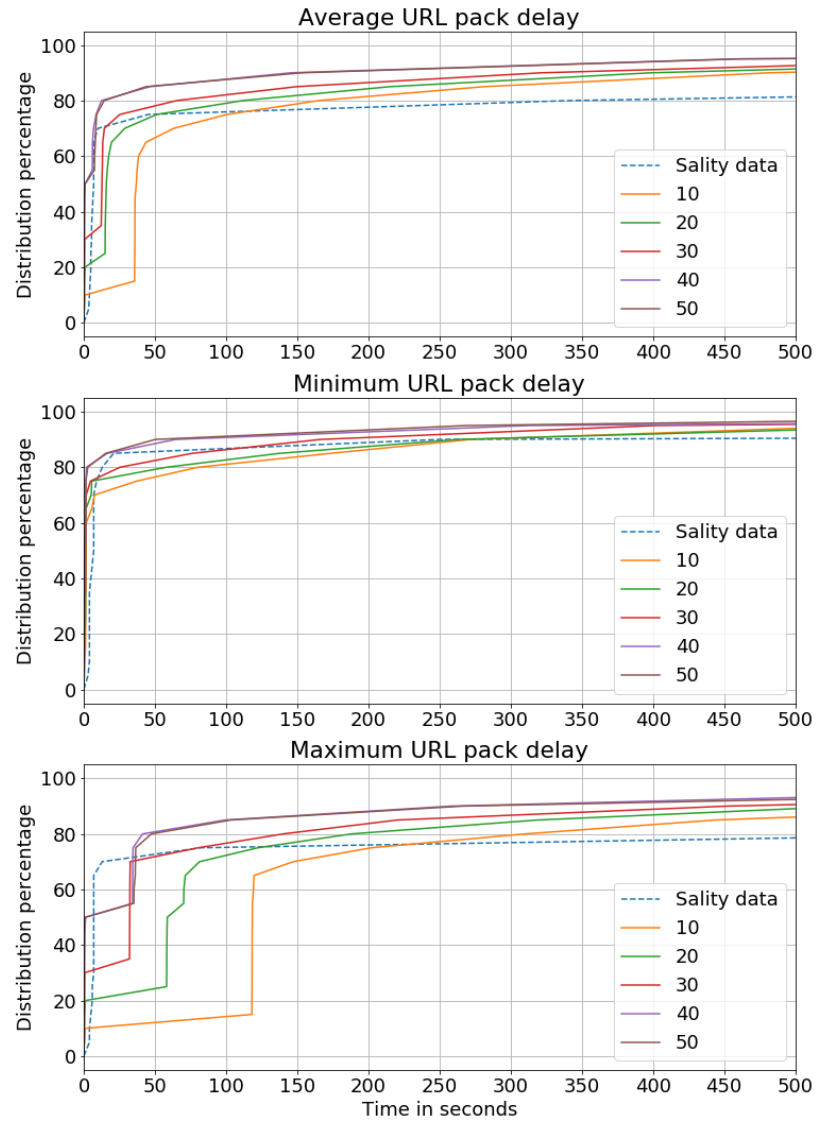


Figure 9: AB2 selecting peers randomly

The AB1 distribution mechanism 10 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 following subsections only the AB3 approach is further analyzed. 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.

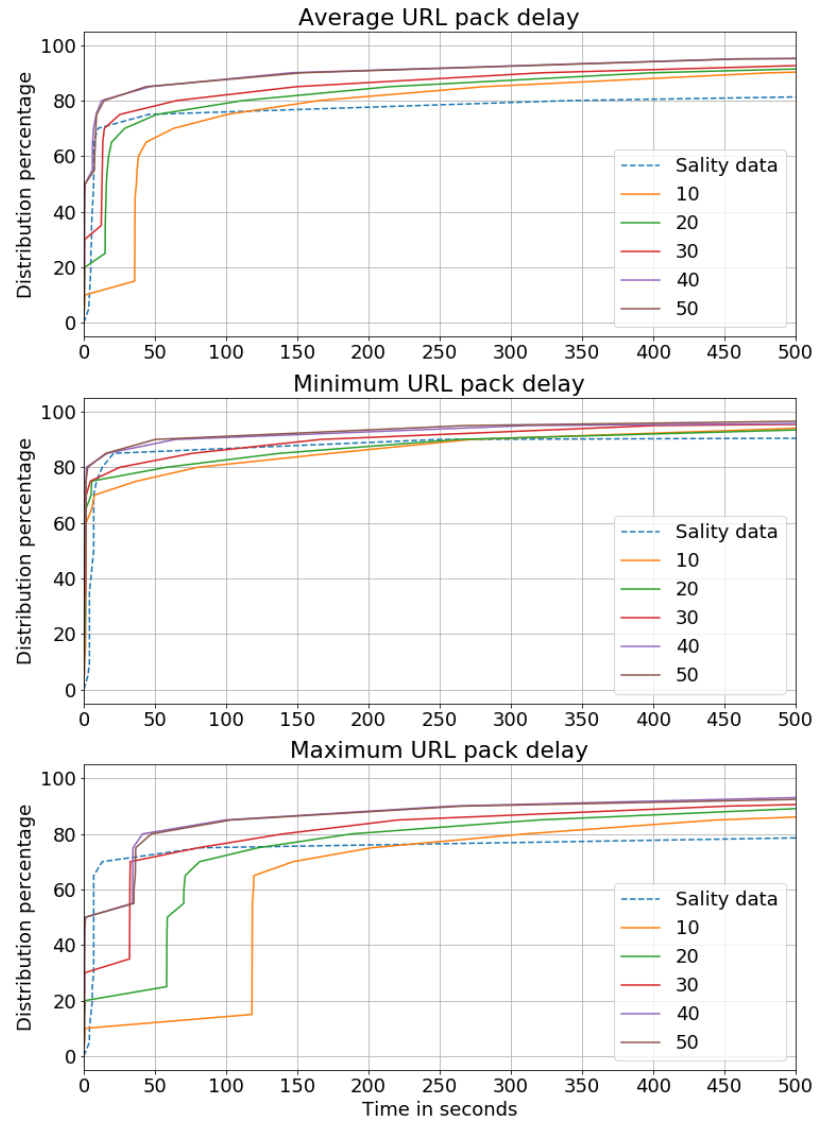


Figure 10: AB1 selecting peers randomly

Selection based on the neighbourlists

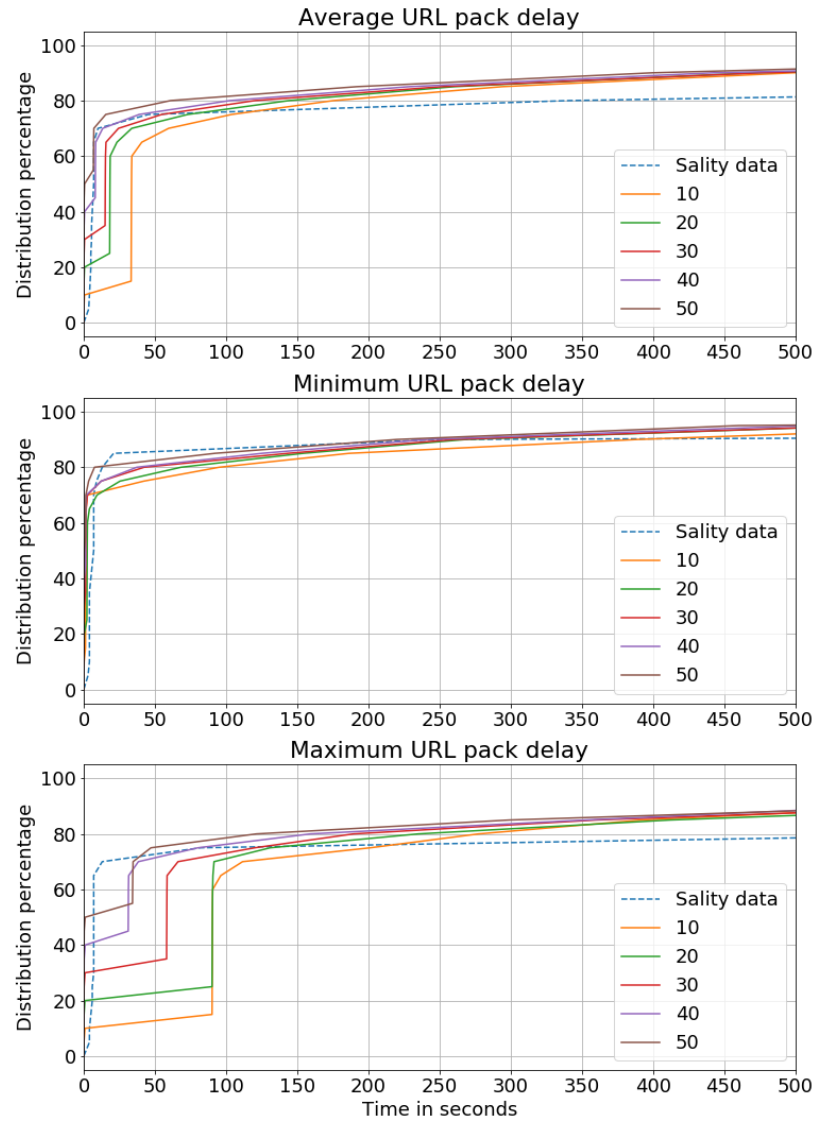


Figure 11: AB3 selecting peers based on the neighbourlists

Selection based on the next MM cycle

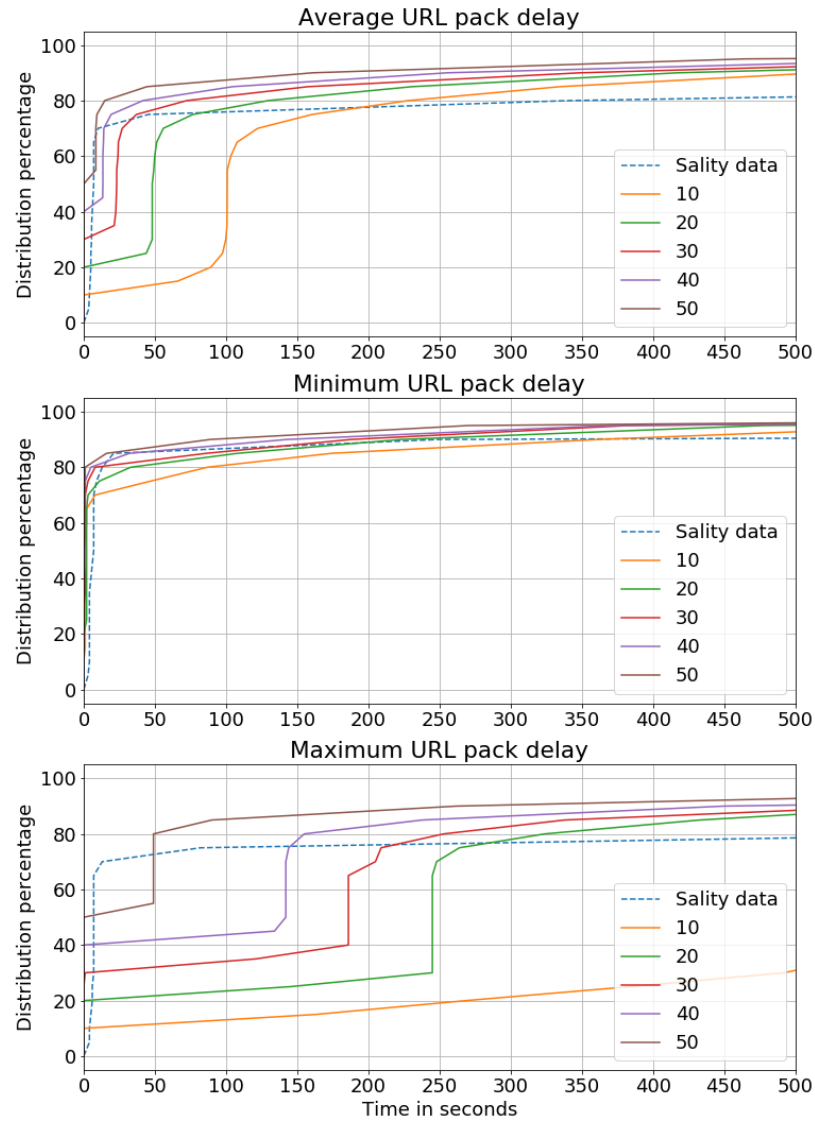


Figure 12: AB3 selecting peers based on the MM cycle

After evaluating the different botmaster strategies, the AB1 and AB2 methods seem to fit the real data well???. The communication overhead of following Salty's communication protocol does not seem to influence the message propagation in a relevant way. The PB approach is not likely to be used in the real botnet. This is not only due to the unfitting data, but also the drawbacks of being part of the network. For further evaluation of the crawlers, the AB2???? approach is used, since it seems to be the most likely one to be deployed by the real botmaster. Furthermore, crawlers are tested on 20 – 40% *distribution_percentage*. To account for node churn and a botmaster that utilizes monitoring techniques for a quickly changing networks, only 10% of the known peers are held constant, whereas the others are randomly changing over time.

4.2 Crawlers

This section evaluates different crawlers that traverse the network towards the malware source, using the distribution method found in 4.1.3.

Notation

For easier comparison between crawlers, 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 whole botnet.
- 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 of the URL pack bot $u \in V_S$ holds.

4.2.1 Crawler versions

The following different crawler versions are compared:

- **Crawler V1** This version holds a mapping of $V_E^{(u)} \rightarrow seq_u$, where u is the neighbouring bot that is potentially connected to the botmaster. Thus it saves for each eligible bot the corresponding sequence number in V_{Eu} . V_{Eu} is initially set to all known peers, with each having a sequence number of 1. The peer filtering happens by constantly iterating over V_{Eu} , kicking bots that do not hold seq_{max} . It does this by sending a URL probe message as explained in 3 at timestamp t_i , where i is the number of the iteration. For each response, it saves the corresponding mapping in V_{Eu} . If superpeers do not answer after a time period of δ has passed, they are probed again n times. If they have not answered after the n th message, they are assumed to be offline and kicked from

V_{Eu} . Thus, one probing cycle takes at maximum $n \times \delta$ seconds. All remaining superpeers are now compared. Every bot that does not hold the current seq_{max} is also kicked from V_{Eu} . This process is repeated until no changes in V_{Eu} happen for x cycles. Figure 4.2.1 displays the corresponding algorithm. It is noteworthy, that V_{Eu} has to be iterated two times in order to drop all failing superpeers. In the first iteration seq_{max} is found over all bots, already dropping ones that have a lower sequence number. In the second one, all peers u with $seq_u < seq_{max}$ are dropped. Overall, the time complexity of this algorithm is in $O(n \times (2 + \delta))$, which is feasible, since Salty usually contains less than 1000 superpeers.

- **Crawler V2** This crawler utilizes timestamps and neighbour information in order to filter out superpeers.

```

count ← 0
while count < x do
  count ← count + 1
  for v in VEu do
    pollUntilResponseOrTimeout(v)
  end for
  currentMax ← 1
  for v in VEu do
    if seqv ≥ currentMax then
      currentMax = seqv
    else
      VEu.drop(v)
      count ← 0
    end if
  end for
  for v in VEu do
    if not seqv = currentMax then
      VEu.drop(v)
      count ← 0
    end if
  end for
end while

```

Figure 13: Algorithm of the Crawler V1

4.2.2 Evaluation parameters

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

1. Size of subset $V_E \subset V_S$ of superpeers potentially connected to the bot-master. The smaller this size, the better the crawler.

2. Average steps of a superpeer p for $p \in V_E$ to the initial superpeer that received the package. The higher this metric, the worse the crawler performed.
3. Number/percentage of superpeers p for $p \in V_S, p \notin V_E$ that are closer to the initial source as stated in 2.. This is the amount of superpeers that have not been found, but are potentially connected to the botmaster.

4.2.3 Results

4.3 Summary

5 Conclusion

5.1 Results

5.2 Future work

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. 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 Address 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 successful 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