VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Swarm Malware

Hejnový virus

2017

Lubomír Sikora

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Diploma Thesis Assignment

Student:	**Bc. Lubomír Sikora**
Study Programme:	N2647 Information and Communication Technology
Study Branch:	2612T025 Computer Science and Technology
Title:	Swarm Malware Hejnový malware
The thesis language:	English

Description:

The goal of this work is designing a modern malware in order to study its behavior. The object is a so-called swarm virus, which is going to be based on the swarm intelligence principle. The payload will be only simulated. The aim of the study is better understanding of the behavior of a hypothetic swarm virus. Until this point there are developed only some viruses based on the C&C (Command and Control) philosophy. The understanding principles of the swarm virus and its dynamicity will help with defense against such threat in the future.

1. The current state in the field of malware and swarming algorithms evaluation
2. Areas where the approach is used and where it is so far unused, eventually where it is used only marginally assessment
3. A basic swarm malware design and a chosen problem solution
4. A program creation and testing

References:

[1] Merhaut F., Zelinka I., Úvod do počítačové bezpečnosti [Introduction to Computer Security], Fakulta aplikované informatiky, UTB ve Zlíně, Zlín 2009
[2] Peter Szor, Počítačové viry – analýza útoku a obrana [Computer Malware – Analysis of Attack and Defense], Zoner Press
[3] Zelinka I., Oplatková Z., Šeda M., Ošmera P., Včelař F., Evolutionary techniques – principles and applications, BEN, Prague, 2008, 598 p.

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor:	**prof. Ing. Ivan Zelinka, Ph.D.**
Date of issue:	01.09.2016
Date of submission:	28.04.2017

[illegible]

doc. Dr. Ing. Eduard Sojka
Head of Departmen

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA
Fakulta elektrotechniky a informatiky
1D16
OSTRAVA

[illegible]

prof. RNDr. Václav Snášel, CSc.
Dean

I hereby declare that this master's thesis was written by myself. I have quoted all the references I have drawn upon.

Ostrava, April 28, 2017

[illegible]
. . . .

I would like to thank all the people, who helped me with this work. Special thanks to prof. Ing. Ivan Zelinka, Ph.D. for mentoring, willingness to help, and friendly approach, which I have always received from him.

Abstrakt

Práce má za úkol nastínit strukturu a chování hypotetického hejnového malwaru. Ukázat, jak lze za pomocí hejnové inteligence mapovat a procházet složkový systém běžného OS. Tato informace je uložena formou komplexní sítě, kterou vytváří jednotlivé instance viru v průběhu jejich života. Tato komplexní síť má určité vlastnosti spojené s její strukturou, které využívají samotné virové instance k navštívení zajímavějších souborů k napadení, nebo mohou být využity samotným hackerem. Hejnový virus může být také využit naopak k obraně proti podobným hrozbám nebo ke studiu jeho chování. Práce ukázala, že tento vzorec chování může být vštípen do malwaru, pokud již nebyl, a analyzován za účelem budoucí ochrany.

Klíčová slova: hejnový virus, hejnová inteligence, evoluční výpočetní techniky, optimalizace mravenčí kolonií, komplexní síť

Abstract

The goal of this work is to outline a structure and a behaviour of a hypothetical swarm malware. To show, how to map, and walk the file system of a casual OS using a swarm intelligence. The information is stored in a form of a complex network, which, during their life, create individual virus instances. The network has certain properties associated with its structure that benefit virus instances to visit files more interesting to infect or it could be used by a hacker himself. On the other hand, a swarm malware could also be used for a defence against similar threads or it could be used for study of its behaviour. As the work shows, the swarm behaviour pattern could be incorporated, if it had not been yet, to a malware, and analysed for a future protection.

Key Words: swarm malware, swarm intelligence, evolutionary computing, Ant Colony Optimization, complex network

Contents

List of symbols and abbreviations

ACO	–	Ant Colony Optimization
AIDS	–	Acquired Immune Deficiency Syndrome
BBS	–	Bulletin Board System
BIOS	–	Basic Input-Output System
C&C	–	Command and control
DAT	–	a file containing data
DE	–	Differential Evolution
DNA	–	deoxyribonucleic acid
EA	–	Evolutionary algorithms
ECT	–	Evolutionary computing
EP	–	Evolutionary programming
ES	–	Evolution strategy
EXE	–	executable
GA	–	Genetic algorithm
HTML	–	HyperText Markup Language
ISM	–	Immunology System Method
JPEG	–	Joint Photographic Experts Group
LCS	–	Learning classifier systems
MA	–	Memetic Algorithms
MP3	–	MPEG-1 or MPEG-2 Audio Layer III
MS	–	Microsoft
MPEG	–	Moving Picture Experts Group
NASA	–	National Aeronautics and Space Administration
NFLT	–	No Free Lunch Theorem
NTFS	–	New Technology File System
OS	–	operating system
PBIL	–	Population-based incremental learning
PC	–	personal computer
PE	–	Portable Executable
PRT	–	perturbation
PSO	–	Particle Swarm Optimisation
SOMA	–	Self-Organizing Migrating Algorithm
SS	–	Scatter Search
TSP	–	Travelling salesman problem
VBS	–	Visual Basic Scripting

List of Figures

List of Tables

Listings

1 Introduction

A content of this work is aimed at combining malware with other topics, i.e., swarm intelligence, and complex network analysis. Following chapters will set necessary prerequisites to create a hypothetical swarm malware. Swarm intelligence is phenomenon of the presence. It could solve, or find near optimum of any optimization problem. Problems like travelling salesman problem, finding weights of connections in artificial neural networks, training of support vector machines, or just finding global extremes of some complicated functions. All these problems are optimization problems. Swarm intelligence algorithms mimic behaviour of some species when doing something. It could be ants finding building materials for their anthill, wolves hunting prey, or flock of birds finding the highest peak to make a nest. These algorithms are Ant Colony Optimization (ants), Self Organizing Migration Algorithm (wolves), Particle Swarm Optimization (birds). There are much more of them but these algorithms are quite common. Other sort of optimization algorithms are evolutionary algorithms. These algorithms based on Darwinian theory of evolution, on the other hand, utilise cross-breeding, mutation, and natural selection. These are Differential Evolution, Evolution Strategy, basic Genetic Algorithm, etc.

The goal of this work is to use some of evolutionary algorithms, more precisely swarm intelligence algorithms, and make a malware upon them. It follows an idea that the malware is going to have a decentralized behaviour. There will be no master to operate all virus instances but each one of them will act on its own. They will utilise an intelligence originating from crowd stored as a complex network. The network will have certain structure reflecting this intelligence and will help virus instances to visit interesting files in a file system. Basic file system is a tree-based structure. That means no loops. This structure is not very feasible for a swarm dynamics so it will be mapped into the network.

This topic is not explored at all so there is almost no publication about the topic. Publication mentioned in this work will be basics of the evolutionary computation, malware, and complex network analysis. The structure and behaviour of the swarm malware described later is based purely on the combination of these topics.

2 Evolutionary computing

Evolutionary algorithms are search methods that can be used for solving optimization problems. [3] They mimic working principles from natural evolution by employing a population-based approach, labelling each individual of the population with a fitness and including elements of random, albeit the random is directed through a selection process. In this chapter, we review the basic principles of evolutionary algorithms and discuss their purpose, structure and behaviour. In doing so, it is particularly shown how the fundamental understanding of natural evolution processes has cleared the ground for the origin of evolutionary algorithms. Major implementation variants and their structural as well as functional elements are discussed. We also give a brief overview on usability areas of the algorithm and end with some general remarks of the limits of computing.

2.1 Central dogma of evolutionary computations techniques

The evolutionary computational techniques are numerical algorithms that are based on the basic principles of Darwinian theory of evolution and Mendel's foundation of genetics. [3] The main idea is that every individual of a species can be characterized by its features and abilities that help it to cope with its environment in terms of survival and reproduction. These features and abilities can be termed its fitness and are inheritable via its genome. In the genome the features/abilities are encoded. The code in the genome can be viewed as a kind of "blue-print" that allows to store, process and transmit the information needed to build the individual. So, the fitness coded in the parent's genome can be handed over to new descendants and support the descendants in performing in the environment. Darwinian participation to this basic idea is the connection between fitness, population dynamics and inheritability while the Mendelian input is the relationship between inheritability, feature/ability and fitness. Both views have been brought together in molecular Darwinism with the idea of a genetic code (first uttered in its full information-theoretical meaning by Erwin Schrödinger in his 1944 book *What is life?*) and the discovery of the structure of the DNA and its implications to genetic coding by James Watson and Francis Crick in 1953.

By these principles and in connection with the occurrence of mutations that modify the genome and hence may produce inheritable traits that enhance fitness, a development of individuals and species towards best adaptation to the environment takes place. Here, the multitude of individuals in a species serve two connected evolutionary aspects, (i.) provide the opportunity to "collect" mutations and pass traits that are or are not fitness enhancing to descendants on an individual level and (ii.) allow fitness enhancing genomes to spread in the species from one generation to the next if the traits bring advantages in survival and reproduction.

However, it should be noted that Darwin or, as the case may be, Mendel, were not the first. Already in the ancient era, there were thinkers who came with the same idea as Darwin and Mendel. An outstanding thinker, who supported the idea of evolution before Darwin, was

Anaximander, a citizen from Miletus, an Ionian city of Asia Minor. Anaximander's philosophical ideas are summarized in his philosophical tract *"On nature"*, however, this name is of a later date, because this book was not preserved. According to Anaximander, the original principle of the world and the cause of all being is "without a limit" (*"apeiron"* in Greek), from which cold and warm and dry and wet is separated - essentially, one can imagine this principle in the sense of unlimited and undifferentiated wetness, from which all other natural substances and individual species of living creatures arise.

By his idea that the Earth, which he imagined as freely floating in space, was initially in a liquid state and later, when it was drying, gradually gave rise to animals, who at first lived in water and later migrated onto land, Anaximander in part anticipates the modern theory of evolution.

The ECT technology stands or falls with the existence of the so-called evolutionary algorithms (EA) that in principle form the majority of ECT. Besides evolutionary algorithms, there still exist other extensions, such as genetic programming, evolutionary hardware, etc. With respect to the fact that evolutionary algorithms are the backbone of ECT, attention will be paid in this chapter just to these algorithms, whose understanding is **absolutely necessary** for understanding the rest of this publication.

From the above mentioned main ideas of Darwin and Mendel theory of evolution, ECT uses some building blocks which the diagram in Fig. 1 illustrates. The evolutionary principles are transferred into computational methods in a simplified form that will be outlined now.

If the evolutionary principles are used for the purposes of complicated calculations (in accordance with Fig. 1), the following procedure is used:

1. Specification of the evolutionary parameters: For each algorithm, parameters must be defined that control the run of the algorithm or terminate it regularly, if the termination criteria defined in advance are fulfilled (for example, the number of cycles - generations). Part of this point is the definition of the cost function (objective function) or, as the case may be, what is called fitness - a modified return value of the objective function). The objective function is usually a mathematical model of the problem, whose minimization or maximization (generally therefore extremization) leads to the solution of the problem. This function with possible limiting conditions is some kind of "environmental equivalent" in which the quality of current individuals is assessed.

2. Generation of the initial population (generally $N \times M$ matrix, where N is the number of parameters of an individual - D is used hereinafter in this publication - and M is the number of individuals in the population): Depending on the number of optimized arguments of the objective function and the user's criteria, the initial population of individuals is generated. An individual is a vector of numbers having such a number of components as the number of optimized parameters of the objective function. These components are set randomly

and each individual thus represents one possible specific solution of the problem. The set of individuals is called population.

3. All the individuals are evaluated through a defined objective function and to each of them is assigned a) Either a direct value of the return objective function, or b) A fitness value, which is a modified (usually normalized) value of the objective function.
4. Now parents are selected according to their quality (fitness, value of the objective function) or, as the case may be, also according to other criteria.
5. Descendants are created by cross-breeding the parents. The process of cross-breeding is different for each algorithm. Parts of parents are changed in classic genetic algorithms, in a differential evolution, cross-breeding is a certain vector operation, etc.
6. Every descendant is mutated. In other words, a new individual is changed by means of a suitable random process. This step is equivalent to the biological mutation of the genes of an individual.
7. Every new individual is evaluated in the same manner as in step 3.
8. The best individuals are selected.
9. The selected individuals fill a new population.
10. The old population is forgotten (eliminated, deleted, dies,..) and is replaced by a new population; step 4 represents further continuation.

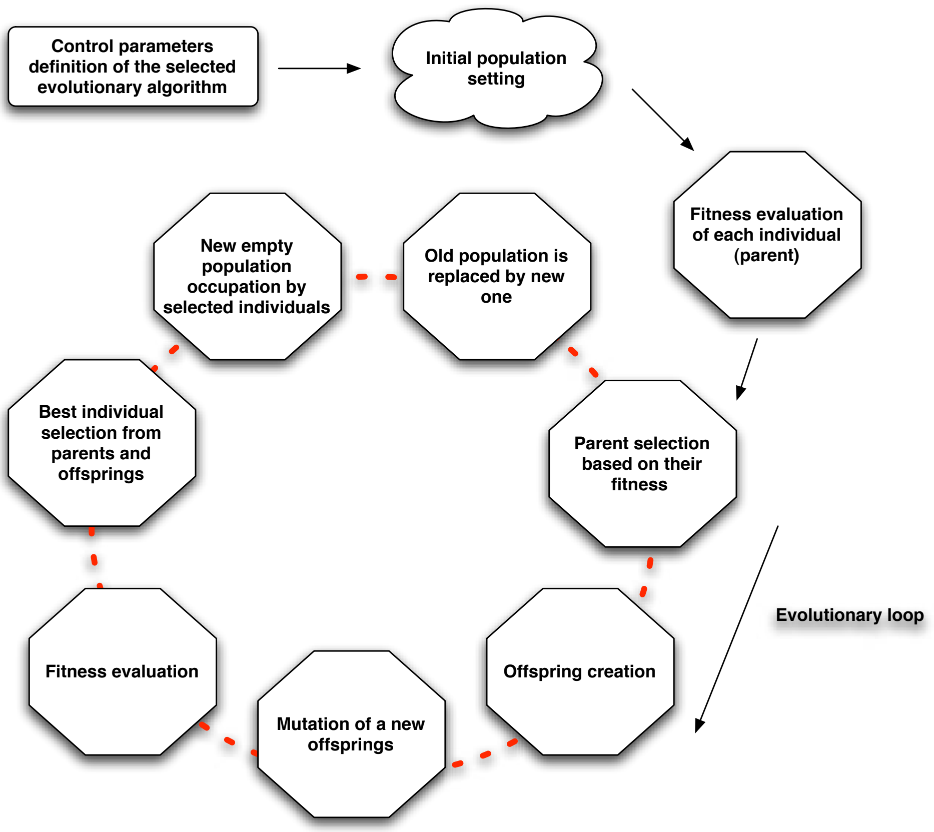

Control parameters definition of the selected evolutionary algorithm
Initial population setting
Fitness evaluation of each individual (parent)
New empty population occupation by selected individuals
Old population is replaced by new one
Best individual selection from parents and offsprings
Parent selection based on their fitness
Evolutionary loop
Fitness evaluation
Offspring creation
Mutation of a new offsprings

Figure 1: General cycle of the evolutionary algorithm. The termination of the evolution after n generations and the selection of the best individual are not indicated in this figure - solution from the last population. Source: [3]

Steps 4 - 10 are repeated until the number of evolution cycles specified before by the user is reached or if the required quality of the solution is not achieved. The principle of the evolutionary algorithm outlined above is general and may more or less differ in specific cases. So, methods that work by an algorithmic structure as outlined in the steps 1-10 share the following main evolutionary principles:

- **Biological inspiration:** The algorithms mimic and use in an abstracted way working mechanisms of biological systems.
- **Population-based calculations:** By structuring data in the algorithm by the individual-and-species model, individual search is coordinated to other individuals and so to the whole population. This has the effect of parallelism in the search which is assumed to be the main reason for success of evolutionary search.
- **Repeated calculation of fitness for all individuals:** This principles provides a spectrum of fitness to the population from which search can be guided by noticing and discriminating individuals of different fitness.

- **Generational search:** Repeated generational search guided by the fitness spectra allows to accumulate individuals with high fitness.
- **Stochastic and deterministic driving forces:** Random influences, for instance in form of mutations are balanced by the deterministic elements in the flow of the algorithm.
- **Coordination between individuals:** Some kind of communication on (or even in-between) the individuals of the population (e.g. in the selection (crossover) or recombination process) allows to recognize and exploit individual differences in fitness.

There are also exemptions that do not adhere to steps 1 - 10; in such a case, the corresponding algorithms are not denoted as evolutionary algorithms, but usually as algorithms that belong to ECT. Some evolutionists exclude them completely from the ECT class. The ACO algorithm (*Ant Colony Optimization*), see [13] and [14] may be an example - it simulates the behaviour of an ant colony and can solve extremely complicated combinatorial problems. It is based on the principles of cooperation of several individuals belonging to the same colony - in this case ants.

The evolution diagrams are not only popular because they are modern and differ from classical algorithms, but mainly because of the fact that they are able to replace a man in the event of a suitable application. This is illustrated in Fig. 2. There are two methods of the problem solution illustrated in this figure. The first one represents steps of a human investigator, the second one represents the procedure if ECT is used.

This publication thus deals with ECT's that in most cases adhere to the above indicated evolutionary scheme; nevertheless, exemptions are also indicated.

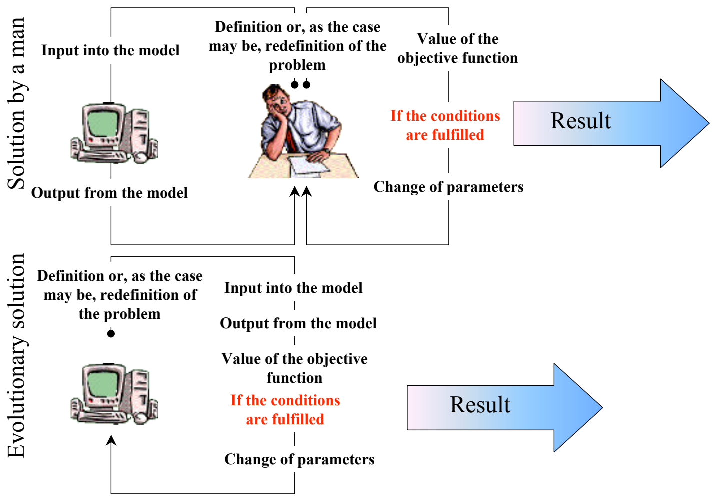

Solution by a man
Input into the model
Definition or, as the case may be, redefinition of the problem
Value of the objective function
If the conditions are fulfilled
Result
Output from the model
Change of parameters
Evolutionary solution
Definition or, as the case may be, redefinition of the problem
Input into the model
Output from the model
Value of the objective function
If the conditions are fulfilled
Result
Change of parameters

Figure 2: Comparison of the problem solution by means of ECT versus a man. Simplified illustration, Source: [3]

2.2 Evolutionary algorithms and importance of their use

Comparing to standard optimization techniques, evolutionary algorithms can be used on *almost* arbitrary optimization problem, however, it is important to remember that with different performance. [3] There are problems with different level of complexity, from the simplest (solvable by standard techniques) to the most complex, whose solution would take much more longer time, than our universe exist. Thus, some simple problems, that can be very easily and quickly solved by gradient based techniques, should not be solved by heuristic methods, because its use would be expensive, i.e. user would "pay" by big number of cost function evaluations. Another important fact, having impact on EA use is so called No Free Lunch Theorem (NFLT), see [26]. Main idea of this theorem is that there is no ideal algorithm which would be able to solve **any** problem. Simply, if there are for example two algorithms **A** and **B**, then for certain subset of possible problems is more suitable algorithms **A** and for another subset algorithm **B**. All those subsets can be of course totally disconnected, or/and overlapped.

Based on those facts it is important to remember that evolutionary algorithms are suitable for problems which are more complex rather simple, and also that their selection and setting depend on user experiences, expertise etc.

2.2.1 The outline of the principles of action of evolutionary algorithms

Genetic algorithm (GA) This algorithm is one of the first successful applied ECT methods [4, 10]. In GAs the main principles of ECT are applied in their purest form. The individuals are encoded as binary strings (mostly over the alphabet $[0, 1]$), which can be understood as a model of the biological counterpart, the genome,[1] and represent possible solutions to the optimization problem under study. After initially a population of binary strings is created randomly, the circle as given in Figure 1 is carried out with the steps fitness evaluation, selection, offspring generation (crossover) and mutation until the algorithm terminates. The application area of these algorithms are wide and it seem particularly sensible to use them if the problem description allows a straightforward coding of the objects to optimize as binary string over a finite alphabet, for instance in combinatorial optimization problem timetabling and scheduling.

Evolutionary strategy (ES) This algorithm also belongs to the first successful stochastic algorithms in history. It was proposed at the beginning of the sixties by Rechenberg [11] and Schwefel [12]. It is based on the principles of natural selection similarly as the genetic algorithms. Contrary to genetic algorithms, the evolutionary strategy works directly with individuals described by vectors of real values. Its core is to use candidate solutions in the form of vectors of real numbers, which are recombined and then mutated with the help of a vector of random numbers. The problem of accepting a new solution is

[1]The genome is coded over the alphabet $[A, C, G, T]$, which stand for the amino acids adenine A, cytosine C, guanine G, thymine T

strictly deterministic. Another distinctive feature is that ES use self-adaptation, that is the mutation strength for each individual is variable over the generational run and subject to an own evolutionary adaptation and optimization process.

Evolutionary programming (EP) EP algorithms [6] have much similarity to ES in using vectors of real numbers as representation. The main operator in the generational circle is mutation, in most (particularly early) implementations no recombination is carried out. In recent years, by adopting elements of their algorithmic structure EP more and more tends to become similar to ES.

Learning classifier systems (LCS) LCS [7] are machine learning algorithms which are based on GAs and reinforcement learning techniques. Interestingly, LCS were introduced by Holland[2] [4] and for a certain time regarded as a generalization of GAs. LCS optimize over a set of rules that are intended to best-fit inputs to outputs. The rules are coded binary and undergo an adaptation using GA-like optimization that modifies and selects the best rules. The fitting of the rules is determined by reinforcement learning methods.

Population-based incremental learning (PBIL) PBIL was proposed by Baluja [8] and combines ideas from evolutionary computation with methods from statistical learning [9]. It uses a real valued representation that is usually restricted to the interval $[0, 1]$ and can be interpreted as the probability to have a "1"-bit at a certain place in a binary string. From these probabilities, a collection of binary strings is created. These strings are subjected to a standard evolutionary circle with fitness evaluation, selection and discarding of inferior samples. In addition, based on the evaluation of the fitness, a deterministic statistical-learning-like updating of the probability vector takes place, which afterwards is also altered by random mutation.

Ant Colony Optimization (ACO), [13] This is an algorithm whose action simulates the behaviour of ants in a colony. It is based on the following principle. Let there be a source of ants (colony) and the goal of their activity (food), see Fig. 3. When they are released, all the ants move after some time along the shorter (optimum) route between the source and goal. The effect of finding the optimum route is given by the fact that the ants mark the route with pheromones. If an ant arrives to the crossroads of two routes that lead to the same goal, his decision along which route to go is random. Those ants that found food start marking the route and when returning, their decision is influenced thanks to these marks in favour of this route. When returning, they mark it for the second time, which increases the probability of the decision of further ants in its favour. These principles are used in the ACO algorithm. Pheromone is here represented by the weight that is assigned to a given route leading to the goal. This weight is additive, which makes it possible to add further "pheromones" from other ants. The evaporation of pheromones is also taken

[2]Holland is also know as the father of GAs.

into account in the ACO algorithm in such a way that the weights fade away with time at individual joints. This increases the robust character of the algorithm from the point of view of finding the global extreme. ACO was successfully used to solve optimization problems such as the travelling salesman problem or the design of telecommunication networks, see [14].

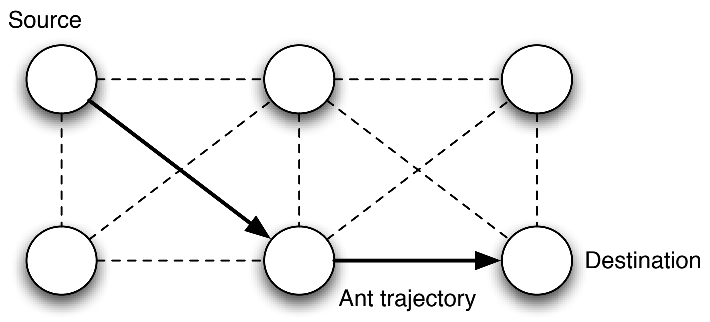


Figure 3: Principle of the ACO algorithm. Source: [3]

Immunology System Method (ISM) This algorithm is unusual by its algorithm based on the principles of functioning of the immunology system in living organisms. As indicated in [14], there are several principles based on this model. In this work, the immunology system is considered as a multivalent system, where individual agents have their specific tasks. These agents have various competencies and ability to communicate with other agents. On the basis of this communication and a certain "freedom" in making decisions of individual agents, a hierarchic structure is formed able to solve complicated problems. As an example of using this method, antivirus protection can be mentioned in large and extensive computer systems [18], [19].

Memetic Algorithms (MA) This term represents a broad class of metaheuristic algorithms [14], [15], [16], [17]. The key characteristics of these algorithms are the use of various approximation algorithms, local search techniques, special recombination operators, etc. These metaheuristic algorithms can be basically characterized as competitive-cooperative strategies featuring attributes of synergy. As an example of memetic algorithms, hybrid combinations of genetic algorithms and simulated annealing or a parallel local search can be indicated. Memetic algorithms were successfully used for solving such problems as the travelling salesman problem, learning of a neural multilayer network, maintenance planning, nonlinear integer number programming and others (references see [14].

Scatter Search (SS) This optimisation algorithm differs by its nature from the standard evolutionary diagrams. It is a vector oriented algorithm that generates new vectors (solutions) on the basis of auxiliary heuristic techniques. It starts from the solutions obtained by means of a suitable heuristic technique. New solutions are then generated on the basis of a subset of the best solutions obtained from the start. A set of the best solutions is then selected from these newly found solutions and the entire process is repeated. This algorithm was used for the solution of traffic problems, such as traffic control, learning neural network, optimisation without limits and many other problems [14], [20].

Particle Swarm (PSO) The "particle swarm" algorithm is based on work with the population of individuals, whose position in the space of possible solutions is changed by means of the so-called velocity vector. According to the description in [14], [23] and [21], there is no mutual interaction between individuals in the basic version. This is removed in the version with the so-called neighbourhood. In the framework of this neighbourhood, mutual interaction occurs in such a manner that individuals belonging to one neighbourhood migrate to the deepest extreme that was found in this neighbourhood.

Differential Evolution (DE) Differential Evolution [22] is a population-based optimisation method that works on real-number coded individuals. For each individual $\overrightarrow{x}_{i,G}$ in the current generation G, DE generates a new trial individual $\overrightarrow{x'}_{i,G}$ by adding the weighted difference between two randomly selected individuals $\overrightarrow{x}_{r1,G}$ and $\overrightarrow{x}_{r2,G}$ to a third randomly selected individual $\overrightarrow{x}_{r3,G}$. The resulting individual $\overrightarrow{x'}_{i,G}$ is crossed-over with the original individual $\overrightarrow{x}_{i,G}$. The fitness of the resulting individual, referred to as perturbed vector $\overrightarrow{u}_{i,G+1}$, is then compared with the fitness of $\overrightarrow{x}_{i,G}$. If the fitness of $\overrightarrow{u}_{i,G+1}$ is greater than the fitness of $\overrightarrow{x}_{i,G}$, $\overrightarrow{x}_{i,G}$ is replaced with $\overrightarrow{u}_{i,G+1}$, otherwise $\overrightarrow{x}_{i,G}$ remains in the population as $\overrightarrow{x}_{i,G+1}$. Differential Evolution is robust, fast, and effective with global optimisation ability. It does not require that the objective function is differentiable , and it works with noisy, epistatic and time-dependent objective functions.

SOMA (Self-Organizing Migrating Algorithm) is a stochastic optimisation algorithm that is modelled on the social behaviour of cooperating individuals [24, 25]. It was chosen because it has been proven that the algorithm has the ability to converge towards the global optimum [24, 25]. SOMA works on a population of candidate solutions in loops called *migration loops*. The population is initialized randomly distributed over the search space at the beginning of the search. In each loop, the population is evaluated and the solution with the highest fitness becomes the leader L. Apart from the leader, in one migration loop, all individuals will traverse the input space in the direction of the leader. Mutation, the random perturbation of individuals, is an important operation for evolutionary strategies (ES). It ensures the diversity amongst the individuals and it also provides the means to restore lost information in a population. Mutation is different in SOMA compared with other ES strategies. SOMA uses a parameter called PRT to achieve perturbation. This parameter has the same effect for SOMA as mutation has for GA. The novelty of this approach is that the PRT Vector is created before an individual starts its journey over the search space. The PRT Vector defines the final movement of an active individual in search space. The randomly generated binary perturbation vector controls the allowed dimensions for an individual. If an element of the perturbation vector is set to zero, then the individual is not allowed to change its position in the corresponding dimension. An individual will travel a certain distance (called the path length) towards the leader in n steps of defined length. If the path length is chosen to be greater than one, then the individual will overshoot the leader. This path is perturbed randomly.

The evolutionary algorithms can be essentially used for the solution of very heterogeneous problems. Of course, for the solution of the optimisation problems, there are many more algorithms than were indicated here. Because their description would exceed the framework of this text, we can only refer to the corresponding literature, where the algorithms indicated above are described in more details.

3 Ant Colony Optimisation

The Ant Colony Optimisation is an important part of this work so it deserves closer attention. Ant Colony Optimisation (ACO) is a swarm intelligence algorithm mimicking ants searching building materials for their anthill, or food. If an ant finds a good source it lays pheromone on its way back to also help others find the source. Although the ACO can be used for solving optimisation problems in continuous space, the main usage is solving network-associated problems like TSP, Shortest path, routing problems, etc., and combinatorial optimisation problems. The ACO is known in many variants, e.g., Elitist ant system, Max-min ant system, Ant colony system, Rank-based ant system, etc. The basic algorithm is described here:

```
Data: graph
Result: the shortest path
initialization;
while stopping criterion not satisfied do
    position each ant in a starting node;
    do
        foreach ant do
            choose next node by applying the state transition rule given by formula 1;
        end
    while every ant has build a solution;
    update best solution;
    apply pheromone update given by formula 2, 3, and 4;
end
```

Algorithm 1: basic ACO pseudocode

In Ant system, m (artificial) ants concurrently build a tour of the TSP. Initially, ants are put on randomly chosen cities. [13] At each construction step, ant k applies a probabilistic action choice rule, called random proportional rule, to decide which city to visit next. In particular, the probability with which ant k, currently at city i, chooses to go to city j is

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta}, \quad \text{if } j \in N_i^k \tag{1}$$

where $\eta_{ij} = 1/d_{ij}$ is a heuristic value that is available a priori, α and β are two parameters which determine the relative influence of the pheromone trail and the heuristic information, and N_i^k is the feasible neighbourhood of ant k when being at city i, that is, the set of cities that ant k has not visited yet (the probability of choosing a city outside N_i^k is 0). By this probabilistic rule, the probability of choosing a particular arc (i, j) increases with the value of the associated pheromone trail τ_{ij} and of the heuristic information value η_{ij} The role of the parameters α and β is the following. If a $\alpha = 0$, the closest cities are more likely to be selected: this corresponds to a classic stochastic greedy algorithm (with multiple starting points since ants are initially randomly distributed over the cities). If $\beta = 0$, only pheromone amplification is at work, that is, only pheromone is used, without any heuristic bias. This generally leads to rather poor results and, in particular, for values of $\alpha > 1$ it leads to the rapid emergence of a stagnation situation, that is, a situation in which all the ants follow the same path and construct the same tour, which, in general, is strongly suboptimal [13]

After all the ants have constructed their tours, the pheromone trails are updated. This is done by first lowering the pheromone value on all arcs by a constant factor, and then adding pheromone on the arcs the ants have crossed in their tours. [13] Pheromone evaporation is implemented by

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij}, \quad \forall (i,j) \in L \tag{2}$$

where $0 < \rho \leq 1$ is the pheromone evaporation rate. The parameter ρ is used to avoid unlimited accumulation of the pheromone trails and it enables the algorithm to "forget" bad decisions previously taken. In fact, if an arc is not chosen by the ants, its associated pheromone value decreases exponentially in the number of iterations. [13] After evaporation, all ants deposit pheromone on the arcs they have crossed in their tour:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}, \quad \forall (i,j) \in L \tag{3}$$

where $\Delta\tau_{ij}^{k}$ is the amount of pheromone ant k deposits on the arcs it has visited. It is defined as follows:

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{1}{C^k}, & \text{if arc } (i,j) \text{ belongs to } T^k; \\ 0, & \text{otherwise}; \end{cases} \tag{4}$$

where C^k, the length of the tour T^k built by the k-th ant, is computed as the sum of the lengths of the arcs belonging to T^k. By means of equation 4, the better an ant's tour is, the more pheromone the arcs belonging to this tour receive. In general, arcs that are used by many ants and which are part of short tours, receive more pheromone and are therefore more likely to be chosen by ants in future iterations of the algorithm. [13]

4 Malware

This section is all about malware; overview, history, infection...

4.1 Computer virus - an introduction

The term "virus" is established in subconscious of people the most; therefore, it often refers to any type of infiltration, regardless of whether it really is a virus, Trojan horse, or worm. [1] Even some publications ignore this fact. The division of infiltration is a problem, types of infiltrations often blend together. The following division is therefore one of many options.
The name is derived from certain biological similarities with the original. Virus is capable of self-replication, but in the presence of an enforceable host to which it is attached. Hosts can be, for example, executable files, system area of the disc, or files that cannot be run directly but only by using specific applications (Microsoft Word, Visual Basic scripts, etc.). Once the host is running (performing), it performs also the virus code. During this moment, the virus usually attempts to provide further self-replication by attaching to other suitable hosts.
It has already happened a couple of times, that journalists were cheering the discovery of the first virus that can infect files and MP3 format JPEG3. In this case, we cannot talk about infection but a simple completely unusable code (virus body) to the above-mentioned format. Since the body of the virus is not adhered to the original code, the player (in the case of MP3), or viewer (JPEG) have no common structure, consider the body of the virus as "garbage". In addition, JPEG and MP3 are data formats, while the body of the virus is a binary code. A related issue is understanding the difference between a size of the file and its extension. If we talk about the size, we talk about the internal structure of chosen extensions, it may not be actually responsible for. Therefore, there are cases where, for example, are infected files with a. DAT (otherwise completely uninteresting by the virus), but only on the grounds that its internal structure corresponds to EXE file format specification.
According to the site of action viruses are divided into further categories:

- A file virus - it is distributed in the form of an executable code. Usually, the author of the pre-defined set of operating system with which the virus is able to carry out their activities. File virus then can create copies, overwrite other files (overwriting viruses), or to connect to other executable files (parasitic viruses).
- Boot viruses - they attack master boot record and the boot sector of a removable media. The virus will start as soon as it passes BIOS boot. It is loaded into memory (and becomes memory resident), casts address system services, and, of course, takes the original system.
- Script Viruses - viruses created with the scripting language of the system. It is possible to use only those options that are accessible through scripting language. They are saved as a script inside a batch file and it can be very easy to write these viruses.

- Macroviruses - some programs allow the user to automate and simplify the work using a "macro". Most often office applications provide this function. They are therefore similar to script viruses, but they serve other than the operating system software. Macroviruses function use only the parent program and the target macro language.

Viruses can be stored in other ways - such as data structures, but they must be implemented in the application in which they are intended to run, otherwise they become worthless.
The emergence of the first virus (Brain) dates back to 1986. It was written by two brothers Basit Farooq Alvi and Amjads from Lahore, Pakistan. Reportedly, they gave it as a bonus to foreigners who were illegally buying software from them. Needless to say, it was a very good work (use of "stealth" technology), which caused a number of local epidemics. Brain was a boot virus, spreading across the disc, forgotten when starting PC in the drive.
We should not forget the experiments of Mr. Frank Cohen, who published an article "Computer Viruses: Theory and Experiments" in 1983. His first experiments with viruses took place on 10 September 1983 on VAX 11/750 under UNIX, over which he lost control after half an hour.
In 1987, he gradually discovered several viruses: Lehigh, Stoned, Vienna, Cascade. The last one named became literally anecdote about his famous falling letters on the screen (like the virus requesting biscuits).
In 1988, the famous virus Jerusalem appeared and as Eugene Kaspersky mentions in his book, number of experts could not believe in the existence of computer viruses. Even the legendary Sam Peter Norton (Norton Commander and author of Symantec products) stated that no viruses exist. Surprising is that few years later he created the Norton Anti-Virus. Another important date is November 1988 when Robert T. Morris, Cornell University student, launched his "Morris worm". Worms attacked around 6000 Unix computers (he reportedly had "somehow" got it out of control), he blocked the network for 36 hours. Indirectly, this event touched millions of people. The worm even got to the Lawrence Livermore National Laboratory, where scientists worked on the clandestine nuclear program in NASA and the damage amounted to $ 100 million.
An era of antivirus programs also began in 1988. One of the oldest is McAfee VirusScan, as well as Dr. Solomon AVTK. The last mentioned was later bought by the first one. In December 1989, about 20 thousand copies of the "AIDS Information Diskette Version 2.0." were sent. In fact, it was a Trojan horse that undermined the data on the disk after being launched more than 90 times. It also demanded payment of $ 189 to a P.O.box in Panama. In 1990, polymorphic viruses began to appear; that is, viruses for that each specimen looks different "on the outside". Existing anti-virus companies had to develop new detection methods, because they formerly reliable way to search for characteristic strings had proved to be not working anymore.
Lots and lots of viruses were originated on the territory of Bulgaria. The biggest part of the whole production was created by person called "Dark Avenger". It is not surprising that in the same state the first BBS (Bulletin Board System), designed exclusively for the exchange of viruses and source codes, was established. In addition to polymorphic viruses, "stealth" viruses, viruses that can camouflage in the system, are increasingly showing. Example could be the virus

Frodo, which watched handling of files and presented harmless versions of these in fact infected files to the user or to the antivirus. Demonstrational virus was also the first multipartite virus Tequila in 1991. Multipartite viruses can attack the system area of the disc as well as the files. Tequila was moreover polymorphic and stealth as well.
1992 was the year of generators. Ordinary users could create a custom virus in matter of seconds. It was only about setting the parameters of future virus (dissemination, speech, etc.), pressing "generate" and new virus, according to the defined requirements, was born. Especially PS-MPC generator expanded significantly. Otherwise it is impossible to explain why several thousands of viruses were generated in PS-MPC.
During 1994 appeared one of the legends - virus One_Half.3544.A. This powerful polymorphic and multipartite virus from Slovakia caused a sensation. The fact that it could not be wholly detected by antiviruses made it very famous. Decryption algorithm of One_Half virus was divided into several interconnected "islands" that were "scattered" through the infected file. Then widespread antivirus McAfee VirusScan failed to overcome this obstacle and even several months later a significant part of specimen remained outside detection. One_Half progressively encoded contents of your hard disc by a particular key, which it was carrying. If the One_Half was removed (including the key), it also meant loss of the encoded data.
Even though the rise of Windows 95 operating system promised termination of computer viruses, the future was predetermined by Form boot virus, which the company distributed on floppy discs with Microsoft Windows 95 to beta testers.
In August 1995, the concept of a macro virus first appeared, spreading out in the documents created by Microsoft Word. Macro language in Microsoft products was so enforced, so that it was even possible to create a self-replicating program - macro virus. Concept of a macro virus belonged, for a long time, to the most common viruses, also, due to the fact, that antiviruses have not been prepared for that type of infiltration. The relative calm prevailed among owners of the Czech version of Word, since localization affected the names of each function of the macro language! Concept and another large group of macro viruses was not able to operate under the Czech Word. This "problem" was solved with airing of MS Office 97.
The first true virus for Windows 95 was Win95/Boza.A in early 1996. Although this was not a miracle program (Boza was able to operate only under certain versions of Windows 95 in certain language versions), Boza was impulse for the emergence of other viruses for Windows, and rejected the argument that with the launch of Windows 95 came an era of faith.
During 1996 appeared the first macro virus for Microsoft Excel – Laroux, and later so-called "cross" macro viruses, able to spread as a Word document as well as Excel workbooks.
At the end of the year the first "memory resident" virus for Windows 95 - Win95/Punch appeared, who survived in the memory as a VxD driver and infected all the opening PE EXE files.
Development increasingly continued, so, in 1998, the first 32-bit polymorphic viruses for Windows 9x - Win95/HPS and Win95/Marburg were created.
In June 1998, the virus Win95/CIH, which was later named by journalists as "Chernobyl", be-

gan to spread. Virus "Chernobyl" was interesting; with each April 26 (depended on the variant) tried to wipe the Flash BIOS on the motherboard and, in addition, part of the data on the disc. If erasing of the Flash BIOS was successful, then the computer was not able to operate. The problem was not resolvable other than by interference into the hardware. Virus Win95/CIH thus partially disrupted until then reliable claim that the virus could not damage the hardware. On the other hand, it was possible, in most cases, to effectively solve this problem.
In the same year, first scripted viruses appear, specifically VBS / Rabbit or HTML / Internal. A year later, in 1999, similar scripts begin to appear in "mass-mailing". This opens a new era of viruses spread by e-mail. One of the first viruses were e-mail Happy99, followed by the famous macroviruses W97M/Melissa.a@mm. Since it was a macro virus, logically it had to steal a Word document with every spread.

4.2 Infection mechanisms

With regards to infecting executable files there are several methods: prepending, appending, inserting. Executable file consists of a header and a body. The head contains, among other things, the entry point. The entry point is an address to the beginning of the body. The execution starts at the address the entry point is pointing at. More detailed description is in [2].

4.2.1 Prepending

Prepending is the easiest method of the infection. The virus (also executable file) is placed at the beginning of a host file, even before its head. When starting the infected file the virus part of the file is executed. The virus executes its body which copies the original file out and also starts it. Figure 4 shows, how the infected file looks like.

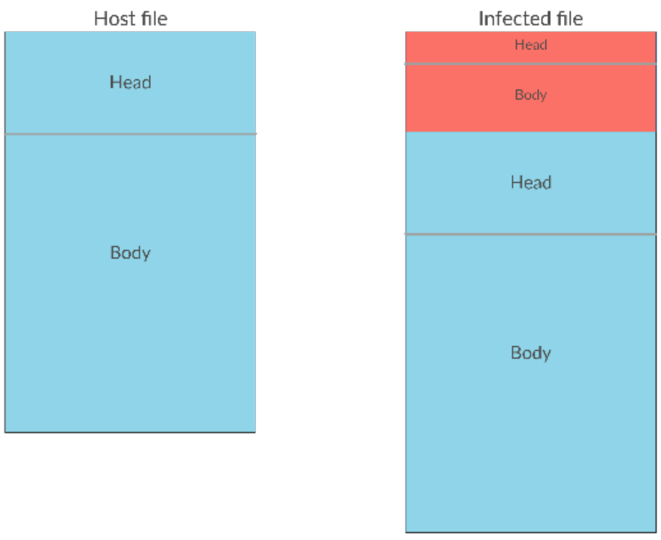


Figure 4: Prepending

4.2.2 Appending

Appending mechanism looks a bit different. The virus is placed at the end of a file and some values of its header are changed. Namely, the entry point. The entry point is set to an address of the virus, so when executing the infected file, the virus takes control. The jump instruction is placed at the end of the virus and its destination is the beginning of the body of the original file. The effect is, after execution of the virus, that the original file takes control. The figure 5 shows, how such infected file looks like.

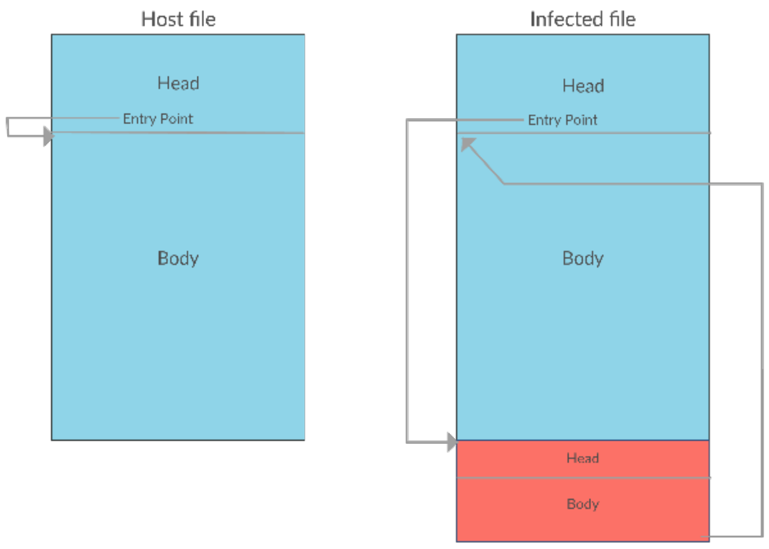


Figure 5: Appending

4.2.3 Inserting

The least utilised method is the inserting. This method gives us several possibilities how to deal with the entry point. The entry point does not have to be changed if the virus is placed at the exact address the entry point is pointing at. Otherwise, the entry point address must be changed to the address the virus is placed at. Inserted virus is shown in the figure 6. Other methods are possible, for example cavity viruses place themselves into the gaps of host files. The body is not a continuous stream of instructions and data so it allows a malicious code to be inserted. Benefit is the size of the infected file remains unchanged if the total size of all gaps is bigger than the size of the virus.

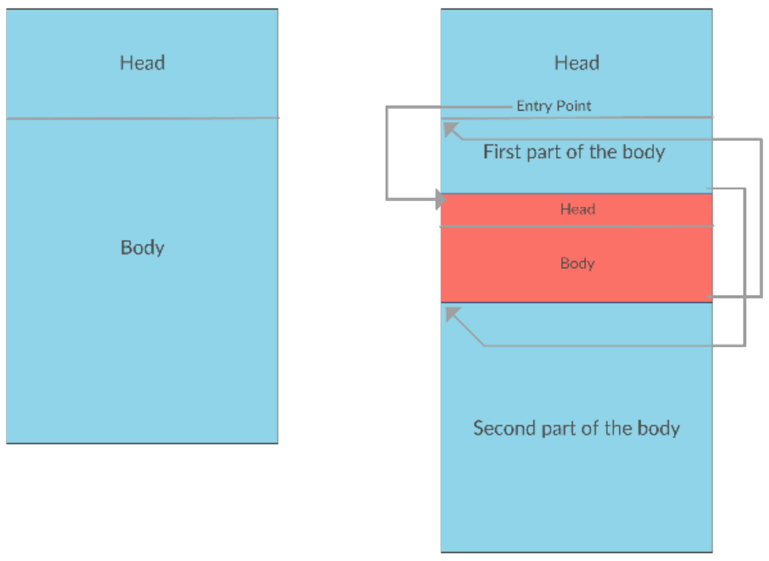


Figure 6: Inserting

5 Swarm Virus, evolution, behaviour and networking

In this chapter we would like to outline how behaviour of malicious software, i.e. computer virus can be connected with evolution and visualization of its spreading as the network. The approach presented here is not based on single classical virus spreading, but more on hypothetical swarm virus and its dynamics of spread in PC. The latest development of virus code shows, that C&C technology (command and control) has been used as in the case of Stuxnet virus or Botnet malware. It is logical to expect that development of the viral code will never stop at this level, but will continue up to viruses that will evolve according to the Darwinian theory of evolution and will mimic swarm in the nature, such as the swarm algorithms already do. The aim of our research is not developing a swarm virus, but using its expectable behaviour we show that its dynamics can be then modelled as the network structure and thus likely controlled and stopped. The same methodology can be used not only in laboratory conditions on a single PC, but also on virus spreading over the network or Internet, if real data is available.

5.1 Swarm virus

Swarm virus is a virus utilising some attributes of the swarm systems in the nature, or swarm algorithms. These algorithms mimic behaviour of some other organisms like birds, fish, ants etc. in certain situation like hunting, searching... In the dynamics of these algorithms intelligent-like behaviour emerges coming from a crowd. Currently there is a number of such algorithms, namely Particle Swarm Optimisation (PSO), Ant Colony Optimisation (ACO), Self-Organizing Migration Algorithm (SOMA) and many others. These algorithms have one feature in common. Optimise or solve problems they are facing to. ACO is very efficient when solving network-like problems, such as Travelling Salesman Problem. PSO or SOMA are, on the other hand, excellent in finding extremes in continuous functions and could be used for example for training an artificial neural network, when a minimal error is needed.

On the other side, there exists a similar sort of algorithms called Evolutionary Algorithms. These algorithms do the same thing but use different techniques based on Darwin's theory of evolution, and use mutation and cross-breeding on optimised solutions. Some of these algorithms are Differential Evolution (DE), Evolution Strategy (ES).

If we want to program the swarm virus, we need to overcome some difficulties. These are movement, communication between virus instances, decentralized behaviour, data storing, and the most important one: behaviour pattern.

5.2 Dynamics of spreading

When it comes to the movement, it depends on the environment we want to move in. If it is the internet or just a file system in an OS. Moving in the internet could be done by email as it was done many times before. Into this category we put viruses like Melissa, or Loveletter also known as ILOVEYOU. Moving only in the file system could be done by using well-known techniques like prepending, appending, or inserting into *.exe files. Such movement must be done not only by infecting file we want to move into but we must clean up the file we want to move from. The virus becomes overpopulated and after a while it will be difficult to find an uninfected host, if we do not settle this detail. If we need to operate all virus instances simultaneously, too many virus instances could cause slowing down the system and subsequent detection.

Another difficulty is that if we move into another host, the new virus instance does not have any data like previous location; local extreme, if something interesting is detected; or ID to distinguish the virus instances from each other. ID is particularly useful when debugging or for further behaviour analysis, or scrum-based behaviour like "virus with ID=4 have to perform something", or contract-based behaviour like "someone has to do this job" and virus with ID=7 accepts it. Command line arguments are useful for this data transaction between moves. We can parse these arguments and set some properties in new virus instance.

5.3 Communication

In communication, we have two options. The first one is that every virus instance can create an offspring (new virus instance) with a set of commands, distributed to it also by command line arguments. It can have implemented few classes responsible for behaviour with same interface and use command line arguments to tell which one to instantiate. These "behaviour classes" can be classes like *payload*. We can specify which one to instantiate by its ID. We can use this technique to determine other aspects of behaviour such as *infector* or classes responsible for swarming which is described later. In tests this behaviour was used.

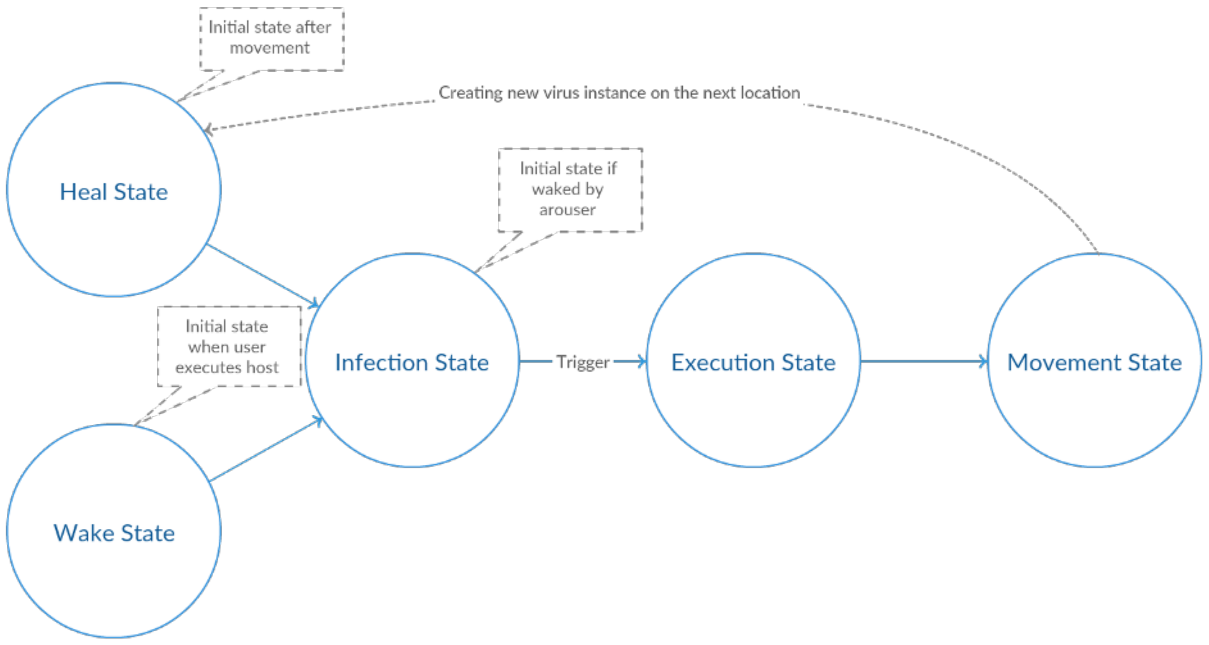
Other way of communication could be feasible by using another file. That file can contain commands or contracts. As stated before, commands are lines such as: Virus with ID=1, do something harmful to a file with path C:\Windows\System32\SomethingImportant.exe. Contracts are lines such as: someone do something harmful with file C:\Windows\System32\SomethingImportant.exe. It is not a bad idea to use both options.

In a swarm virus you have multiple virus instances. None of them is activated by system start up. By double clicking some infected host, you wake up other virus instances. Assuming that you know where other sleeping virus instances are, you can use command line arguments to wake them up. This is necessary if you execute a host file, both virus and the pure host file are executed. So the initiating file must tell others to skip this procedure, because it belongs only to user executed host.

One difficulty to solve is synchronizing file access. The virus instances are decentralized, so multiple file access can occur. You can solve it simply by using ALOHA-like collision protocol. You simply wait for random amount of time to try to access the file.

5.4 Decentralized behaviour

This is an easy task to solve. When writing a swarm intelligent algorithm, you are using a cycle to iterate through each individual. Now you only write the behaviour for one instance and then its execution terminates. You want the behaviour pattern to be like in figure 7.



Initial state after movement
Creating new virus instance on the next location
Heal State
Initial state if waked by arouser
Initial state when user executes host
Infection State
Trigger
Execution State
Movement State
Wake State

Figure 7: Virus instance behaviour pattern

- Wake state: This state occurs when none of other virus instances are operating. User executed host file sends a signal to others to wake up as described in the section 5.3.
- Infection state: This state can be implemented this way: create as many instances as there is total of n. It is also a defence mechanism in case antivirus recognises some of the virus instances and sends them to a chest. So this state maintains constant count of the virus instances.
- Execution state: This is just payload of the virus. This state operates only when a trigger is pulled. If it is not, this state is ignored.
- Move state: All swarming logic is implemented in this state. Detailed description can be found in the section 5.6.
- Heal state: As stated in the section 5.2, there is need to remove the old virus instance, thus instance occupying a file we want it to move from.

The virus starts executing in given state based on the way the virus instance was started. If it was the very first execution after operating system start up, the execution starts in Wake state. If the virus instance was awakened by other virus instance, we want to start execution in Infection state. If the virus instance was created and then started in consequence of moving, we want to start in Heal state.

You can create more complex virus behaviour using this finite automaton technique. This way of implementing swarm virus is also development-friendly and it is easy to imagine individual behaviour from general point of view.

5.5 Data storing

This stage is more about virus writing skill rather than swarm intelligence. You can have remote server to communicate with, but it can produce suspicious amount of traffic and premises to be on-line all the time. Another way is to use alternate streams. These streams are hidden to user and most antiviruses do not check them, but user must have a NTFS file system. Alternate streams look like usual path, but end with colon followed by filename with data. It can look like this: C:\Windows\System32\SomeFile.exe:hiddenData.txt. If you use this technique, there is no way to distribute this virus with memory sticks, as they are not usually formatted to the NTFS file system. Another way is using shadow copy. Basically the data is deleted and cannot appear in windows explorer, but physically it is present in the storage.

You must solve collisions when accessing files once again, but that was mentioned in the section 5.3.

5.6 Behaviour pattern

Let's talk about behaviour pattern, because this is the most significant problematic in swarm viruses. As it was said before, suitable behaviour patterns come from swarm intelligence algorithms like Particle Swarm Optimisation, Ant Colony Optimisation or others. If you want to move around in the file system, its tree structure is not pleasant environment to move coordinately. In the tree structure there are lots of dead ends and no cycles. Good strategy to avoid this problem is to transfigure this tree structure into some network with cycles, but connections leaving somewhere else are not enough, because the leaf can be very far from a root and none of the virus instances could reach it. It was tested in a way which uses Ant Colony Optimisation together with Bianconi-Barabási model that produces networks like those you can see in the section 6.2.

Bianconi-Barabási model produces scale-free community networks. This model works on the principle that you have some basic small network and set of vertexes you want to add. The first link of the new vertex is attached to a random vertex i_1 of the network. The second link is attached to a random vertex of the network with a probability $1-p$, while with the probability p is attached to a node chosen randomly among the neighbours of the node i_1. The whole process of attaching new vertex is shown in figure 8. This model has two values to set. The probability p, and the number m which determines how many links to attach for each new vertex. Bianconi-Barabási model can be enhanced with a *fitness*. It basically means that some vertexes in the network are preferred more than others when creating a new link. The *fitness* is a value representing quality of the file based on your objective. Your objective could be anything based on what you as virus developer want to find and/or infect in victim's OS. Whatever you want to find files based on their size, number of executions, or other criteria, these files then become centres in the network. That means they have a lot of neighbours. More detailed information can be found in [27].

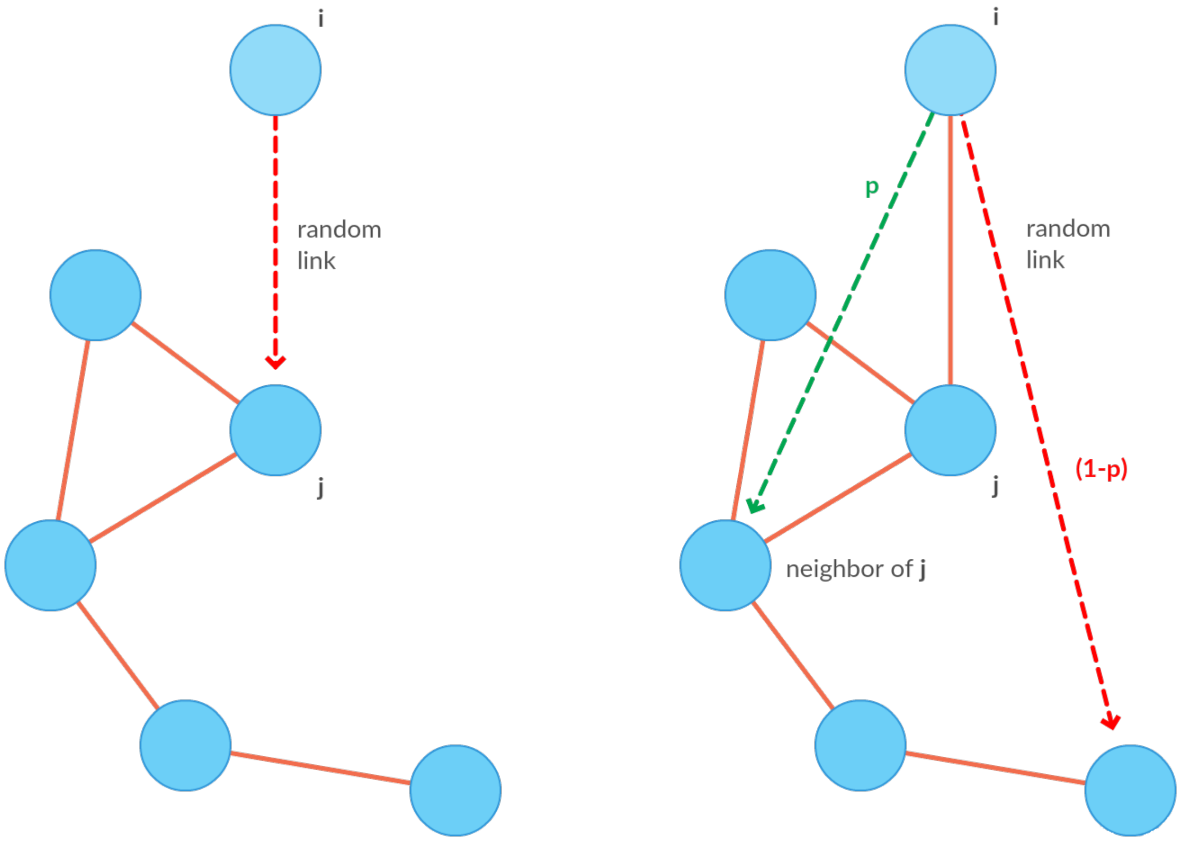


Figure 8: Bianconi-Barabási model attach

General network building process can be described as follows:

```
random initial spread of the virus instances in the file system;
while swarm virus is active do
    foreach virus instance do
        • Update TabooList (TabooList is a list of recently visited vertexes.
          Its purpose is preventing circular movement amongst small number
          of locations. Its length is set to $n$, which means the virus instance
          must visit at least $n$ different vertexes in row);
        • Map neighbourhood (finding nearby host files);
        • Add each host file, or some number of them from neighbourhood
          into network, if it has not been there yet;
        • Vaporize pheromone from all edges;
        • Use any behaviour pattern to determine where to move next;
        • Add some additional links from that new location the way it is described
          in Bianconi-Barabási model (first link was not attached to random vertex,
          but to actual position the virus instance currently stands);
        • Lay pheromone between current and next vertex;
        • Move to newly chosen location;
    end
end
```

Algorithm 2: General network building pseudocode

You can even use a pheromone to slightly prioritize some edges. The whole process is graphically described in the figure 9. Process starts from the center of the star.

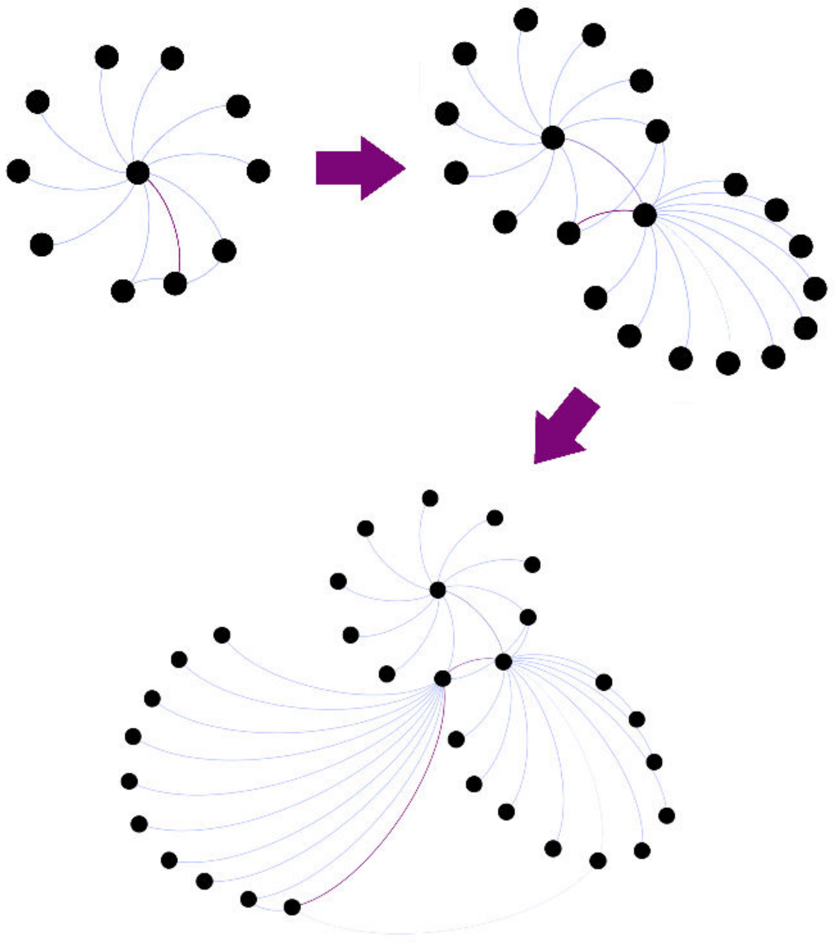


Figure 9: Network building

5.7 Experiment design

When it comes to implementing viruses, the usual choice of the programming language is C, C++ or Assembler. This swarm virus was written in higher programming language, namely C#. The resulting size of the virus was much greater, thus not very convenient for its viral purpose. But it was very convenient for purposes related to a research. It is not difficult at all to write the code in C/C++, when the goal is a real swarm virus designed in order not to be detected by an antivirus due to its size. The virus was tested in Windows 10.

In following lines, we describe details of the code and explain, why the code looks the way it looks.

5.7.1 Modules

The main functionality of this project is found in modules. In modules there are actual implementations of the technology. In this swarm virus, there are six modules. Namely Infector, Objective, Payload, Swarm, Trigger and Communication.

- Infector: the infector contains four methods. *Infect*, *Heal*, *IsInfected*, and *GetOriginalData*. The *Infect* method implements the simplest infection technique among others, that is prepending. Description of prepending can be found in chapter 4.2.1. It is possible to use other techniques, but for the sake of simplicity, the prepending was chosen. The *Heal* method is responsible for healing the host file. It is used after moving to a new location. The *IsInfected* method checks, whether the file is infected or not. It is unwanted to infect already infected file. The last method gets original data from infected file. It is used for the starting original program when the user executes a infected file.
- Objective: This module contains two methods. The *SuitablePaths* returns paths in file system, which are used to initial spread of the virus. The *Fitness* method evaluates a file. It tells how suitable the file is regarding to what we want to infect. In following chapters the $fitness$ value equals a size of the file.
- Payload: the module with this name contains one method: *Run*. In this swarm virus, the payload is just some console output with debugging information, to test the swarm functionality.
- Trigger: the trigger is just predicate, if the payload should be performed. If the condition is not satisfied, the whole execution state is ignored.
- Communication: module responsible for communication with external sources and parsing signals. Its functionality is described in the section 5.7.2.
- Swarm: this module contains the only method - *NextPlace* method. This is the most important method in the swarm virus. It is so important that its functionality is described in its own section 5.7.5. The method returns the next place to move.

5.7.2 External Sources

It is necessary to store some information. These are locations of all virus instances; network, which is currently build; and a log file for testing purposes. The communication module is responsible for communication with these files. It contains methods like *AddHost*, *RemoveHost*, *GetHostList*, *ProcessSignal*, *WriteLogEntry...*

The only external source the communication module is not responsible for is network XML. Reading and writing XML with network is performed in classes representing the network, nodes and edges. The XML format is in fact GEXF format readable by software called Gephi. Gephi is network analysis software.

5.7.3 Signals

Simply said signals are command line arguments of the *main* method. They are some initial values, flags etc. They set the initial state of a virus instance. Signals are cookies, because this virus, similarly to HTML protocol, is stateless and forgets everything after moving to the new location. The virus uses these signals:

- ID: unique value through all virus instances. It is not necessary to have an ID for it, but it is useful for debug purposes to track the path of the virus instance.
- Visited: This is a list of the last n visited locations. It prevents circular visits of small number of locations in a file system.
- WakeUp: flag send from the one who woke it, telling the host not to execute an original program, but only to execute the virus part. Rouser is the host file executed by user. Rouser executes both the virus part and the original program of the host.
- Heal: this signal flag is set when moving to a new location, and contains the order for the old location to heal, to perform the whole movement.

Signals are parsed at the beginning of virus instance execution and set an initial state of the virus.

5.7.4 States

States were described in the chapter 5.7.4. Each state is Singleton and inherits from *Abstract-State*, which has only one method *Run*. Whole class looks like this:

```
public abstract class AbstractState<TType>:
IState where TType: AbstractState<TType>, new()
{
    public static TType Instance { get; } = new TType();
    public abstract void Run();
}
```

Listing 1: AbstractState class

The first thing is, the class using combination of self-referencing generic constraint and a new() constraint. That ensures that every child class will always have a parameterless constructor, so TType Instance = new TType(); will always work. This means that the Singleton pattern is reusable and its child classes will look like this:

```
public class ConcreteState: AbstractState<ConcreteState>
{
}
```

Listing 2: Singleton inheritance example

The class also must implement *IState* interface with one method *Run*. This simplifies work with individual states. We do not want implement the *Run* method yet, so we promise to implement it later by tagging it as abstract. The concrete state classes will implement the *Run* method. The concrete states in tested swarm virus example are heal state, wake state, infection state, execution state and movement state. Here is detailed description of individual states and its usage.

- Heal state: the heal state calls two modules in its *Run* method - the infector and the communication. The infector module is responsible for infection and healing the host files and contains the *Heal* method. The communication module contains method *RemoveHost*, to remove host from our host-list mentioned earlier. Calling this method allows correct start-up even after the computer restart.

- Wake state: the wake state contains additional two other methods besides the *Run* method. *WakeUp* method, and a *WakeUpAll* method. The *WakeUp* method by default starts process of some host with only two signals. The *Id* signal and the *WakeUp* signal. The method starts process the way that it only starts the virus part of the host. The *WakeUpAll* method just calls *WakeUp* for each host file given by the *GetHostList* method in communication module. The *Run* method just calls *WakeUpAll* method.

- Infection state: this state contains four methods. *GetFiles* method takes *N* randomly picked files delivered from Objective module's *SuitablePaths* method. The method *InfectionRoutine* infects a file by calling *Infect* method from the Infector module and adds the file to the host-file by calling *AddHost* from the Communication module. The *StartInfecting* method just cyclically calls the *InfectionRoutine* for each file from *GetFiles*. The *Run* method only counts, how many files to infect and infects them maintaining a constant number of virus instances. It might happen that some virus instances will be found by antivirus.

- Execution state: the *Run* method of this state only calls the *Run* method of the Payload module.

- Movement state: This state contains only the *Run* method. The method gets a path to move from Swarm module's *NextPlace* method and infects it by *InfectionRoutine* from the Infection state. Then it calls the *WakeUp* method placed in the Wake state and adds two more signals: The *Heal* signal containing current path, and *Visited*, containing actual

TabooList. There is no point to continue the execution in current location and the current process is terminated. Now the whole process starts in a new location again and the whole life cycle of one virus instance in one file is over.

Whole functionality is set in class called *StateController*. The controller looks like this:

```
public abstract class AbstractStateController
{
    private readonly Dictionary<IState, List<KeyValuePair<Func<bool>, IState>>>
         _automaton;
    public IState InitialState;

    protected AbstractStateController()
    {
        _automaton = new Dictionary<IState, List<KeyValuePair<Func<bool>,
            IState>>>();
    }

    public void Run()
    {
        var currentState = InitialState;
        do
        {
            currentState.Run();
        } while ((currentState = NextState(currentState)) != null);
    }

    protected IState NextState(IState currentState)
    {
        try
        {
            return _automaton[currentState].First(kv => kv.Key()).Value;
        }
        catch (Exception)
        {
                return null;
        }
    }
    public void AddState(IState state)
    {
        _automaton[state] = new List<KeyValuePair<Func<bool>, IState>>();
    }

    public void AddTransition(IState fromState, Func<bool> predicate, IState
        toState)
    {
        _automaton[fromState].Add(new KeyValuePair<Func<bool>, IState>(
            predicate, toState));
    }

    public void AddTransition(IState fromState, IState toState)
    {
        AddTransition(fromState, () => true, toState);
    }
}
```

Listing 3: AbstractStateController class

The controller is just a state automaton. It contains methods *AddState* for adding states. *AddTransition* for adding transition amongst states with or without triggers. *NextState* for transition after execution of the current state is over. *Run* method is there to start the whole process. *InitialState* is a property that tells us in which state to begin.
Concrete *StateController* could look like this:

```
public class StateController: AbstractStateController
{
    public StateController()
    {
        AddState(HealState.Instance);
        AddState(WakeState.Instance);
        AddState(InfectionState.Instance);
        AddState(ExecutionState.Instance);
        AddState(MovementState.Instance);

        AddTransition(HealState.Instance, InfectionState.Instance);
        AddTransition(WakeState.Instance, InfectionState.Instance);
        AddTransition(InfectionState.Instance, Virus.Trigger.IsTrigged,
            ExecutionState.Instance);
        AddTransition(ExecutionState.Instance, MovementState.Instance);
    }
}
```

Listing 4: Concrete StateController example

The *InitialState* depends on received signal. When the *WakeUp* flag is present, the *InitialState* becomes *InfectionState*. When there is the *Heal* signal, the *InitialState* will be the *HealState*. If no signal is present, the *InitialState* will be set to the *WakeState*, that means an infected program was executed by user. It is not possible to receive more than one of these signals, thus the *InitialState* will be unambiguous.
This code corresponds with figure 7.

5.7.5 Virus dynamics

Virus dynamics is the most important feature in this kind of virus. In this chapter we describe how the *NextPlace* method in the Swarm module is implemented. This swarm virus uses combination of ant colony behaviour and slightly adjusted Bianconi-Barabási model for creating scale-free networks.

5.7.5.1 Ant Colony part As far as the movement part is concerned, picking a next place is given by an ant colony formula 1 found in the chapter 3. Pheromone is not so important at this point. Parameter α related to the pheromone could be set to 1. A vaporization factor ρ should be set to number, which guarantees slow vaporization, so the vaporization factor should be somewhere around the value 0.03. Behaviour of the tested ant colony utilises the max-min variant so minimal pheromone $\tau_{min} = 1$ and maximal pheromone $\tau_{max} = 10$. As we do not want cyclical visits of the same group of nodes, the number n specifying the length of the TabooList should be high, perhaps $n = 100$. The q parameter is just a normalisation factor and depends on the $fitness$. Because we want to lay pheromone proportionally to the fitness of the next node ($fitness_j$) and tested fitness was somewhere in a rage $< 0; 12000000 >$ the formula to update the pheromone from node i to node j looks like this:

$$\tau_{ij} \leftarrow \tau_{ij} + \frac{fitness_j}{q} \tag{5}$$

$$\tau_{ij} \leftarrow \begin{cases} \tau_{min}, & \tau_{ij} < \tau_{min} \\ \tau_{max}, & \tau_{ij} > \tau_{max} \\ \tau_{ij}, & \text{otherwise} \end{cases} \tag{6}$$

q was set to 10000. The weight of an edge is $1/d$, where d is distance of files in file system. The distance is measured by the *number of folders between files (vertexes) + 1*. For example:

- Distance(C:\a\b.exe, C:\a\b.exe) = 0
- Distance(C:\a\b.exe, C:\a\e.exe) = 1
- Distance(C:\a\b.exe, C:\e.exe) = 2
- Distance(C:\a\b.exe, C:\d\e.exe) = 3

Complete set of parameters with recommended values is shown in the table 1.

5.7.5.2 Network creating part As it was mentioned before, the network was created by Bianconi-Barabási model. Although it had been slightly changed. The major change is in the fitness calculation. Here the $fitness$ of the new node was known. As mentioned before, the $fitness$ of a file here is just the size of the file. Fitness is mentioned in chapter 5.6. So, each node was connected to the network with probability equal to its $fitness$. In fact it is not the real probability, because the $fitness$ value could be way higher than 1, but the bigger the $fitness$, the bigger the probability. This type of pick can be described by following algorithm:

```
public static TType ProbabilitySelect<TType>(this IEnumerable<Tuple<TType,
   double>> items)
{
   double r = Random.NextDouble() * items.Sum(i => i.Item2);
   double wheelPosition = 0;
   foreach (var item in items)
   {
      wheelPosition += item.Item2;
      if (wheelPosition >= r) return item.Item1;
   }
   return default(T);
}
```

Listing 5: Roulette pick mechanism

This is an extension method for any sort of collection as far as it is collection of Tuples containing an object to pick, and its pseudoprobability ($fitness$).

The Bianconi-Barabási model has some parameters to set, but here we have only two. A probability to pick a neighbour p and number of links to create m. In practice the p is set to a number near 1. That is because the model shall create a network with community structure. Now we do not care about communities but we do care about collecting swarm intelligence. In this case the intelligence should be in the network structure. The better the node is, the higher should be its centrality. For this purpose, the parameter p is set at least to 0.5. That means probability of connection with a neighbour is equal to the probability of connection with a random node. The reason why the Bianconi-Barabási was chosen is that nearby files in a file system should be connected together, so they could create groups or they should be connected to some important nodes in the network. The more connections important nodes have, the bigger their centrality. The collective knowledge is in the network structure. Another parameter to set is the m parameter. The m tells us, how many connections it will initially have. Because parameter p was optimised to create connections with the important nodes, the m parameter just multiplies this optimisation. Thus higher values are recommended. Numbers higher than 4 performed well. The lower the number, the lower the amount of information stored in the structure of the network. If both numbers are set to low number, the resulting network contains a small amount of information. Networks tend to create centralities even from not so important files, and important files are not registered as centralities. Recommended setting of all parameters is shown in the table 1.

Table 1: Recommended setting of Swarm virus parameters

Parameter	Value
number of ants n	$5 - \infty$
TabooList length	$20 - \infty$
p	$0.4 - 0.9$
m	$4 - 10$
τ_{min}	$1 - 5$
τ_{max}	$\tau_{min} - \infty$
α	$1 - 5$
β	$5 - 10$
ρ	$0.001 - 0.05$
normalization factor q	depends on $fitness$

6 Results

The goal was to create a network with certain properties from a file system. The main property was to have centralities in the network representing more important files in operating system. This property is helpful when figuring out which file to infect. Following graphs show dependence between $fitness$ and some centralities. Then again, the $fitness$ is a value representing importance of a file. The fitness function is set by a virus developer and depends on his needs. The bigger the $fitness$ value, the better the file to infect. In this case the fitness function is just size of the file. We used 10 ants and each ant has moved 10 times.

In graphs, there were shown dependencies between $fitness$ and PageRank. The reason is that the ant is a random walker. The higher the PageRank of a node, the higher the probability of visiting the node. The goal is to visit the most important nodes in the network. Among other centralities the PageRank fits into this scenario the best. Other centralities are useful for further analysis of the network. This could be performed by the virus itself, or the virus can send the network to a hacker. We could see that files with the highest $fitness$ also have higher centrality numbers. They are distinguishable among others. Unimportant files have low centrality numbers. This means a random virus instance walking the network has higher probability to infect a file with the higher $fitness$.

Important factor is, that files with higher $fitness$ should be distinguishable. In graphs it is shown, that files with higher $fitness$ also have higher centrality, so they are more in the right. Files with lower $fitness$ should be at the bottom leftmost corner. $fitness$ and centralities should somehow correlate with each other. This is true in all graphs but the first one. The first one is showing unsuitable parameters setting. We can see files with higher $fitness$ are also in the left (they have low centrality value) and files with low $fitness$ are also in the right (they have high centrality value). So no dependency is showing. On the other hand, all graphs but the first one are showing dependency between $fitness$ and centralities. Not important files, the ones with low $fitness$, tend to be at the bottom leftmost corner, and files with higher $fitness$ are more to the right. The best file, the one with highest $fitness$, is at the up rightmost corner. So it is the most important node in the network also.

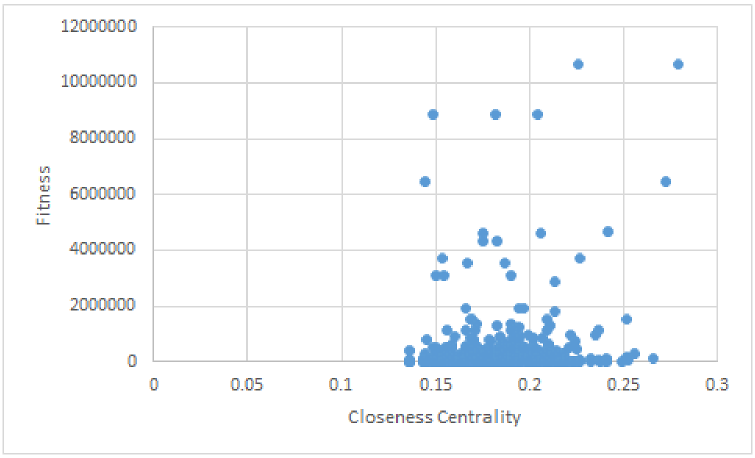


Figure 10: Fitness Closeness Centrality dependency, $p = 0.3$, $m = 2$

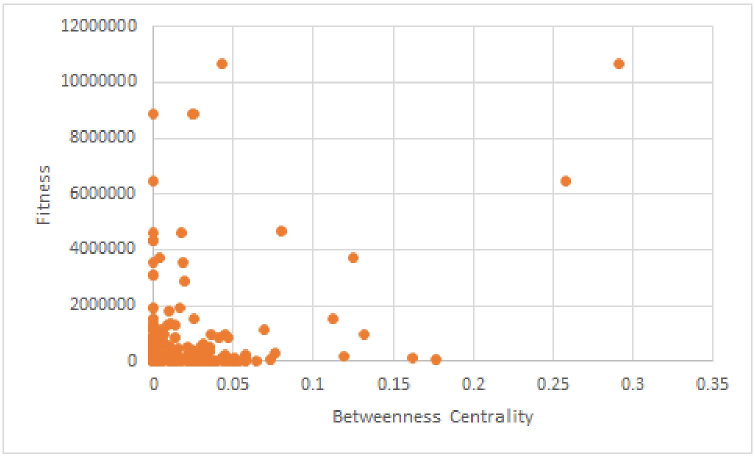


Figure 11: Fitness Betweenness Centrality dependency, $p = 0.3$, $m = 2$

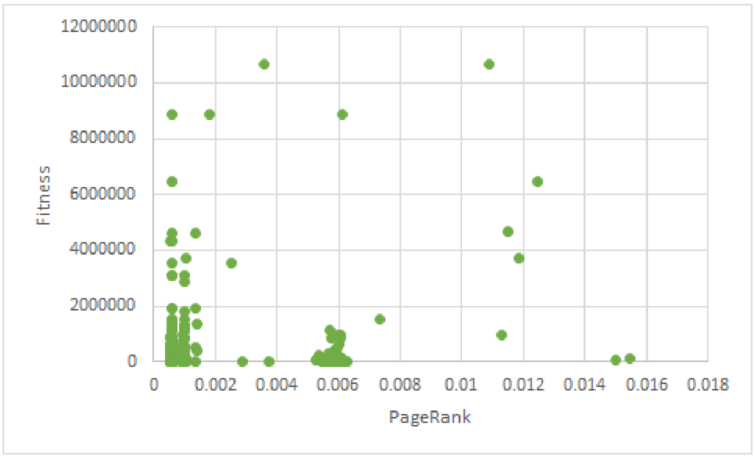


Figure 12: Fitness PageRank dependency, $p = 0.3$, $m = 2$

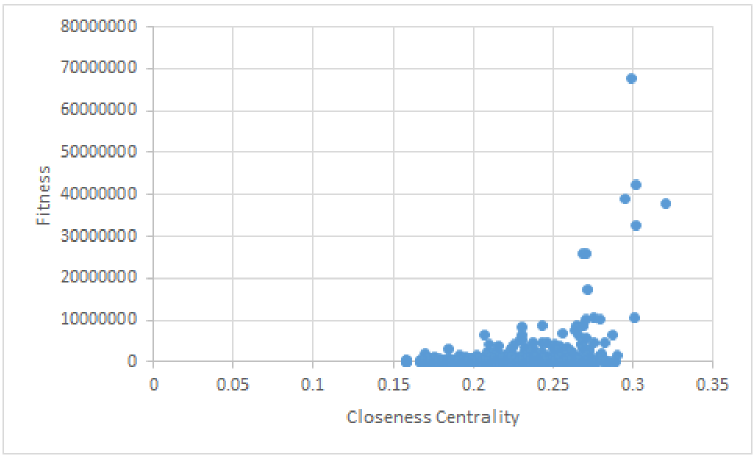


Figure 13: Fitness Closeness Centrality dependency, $p = 0.3$, $m = 7$

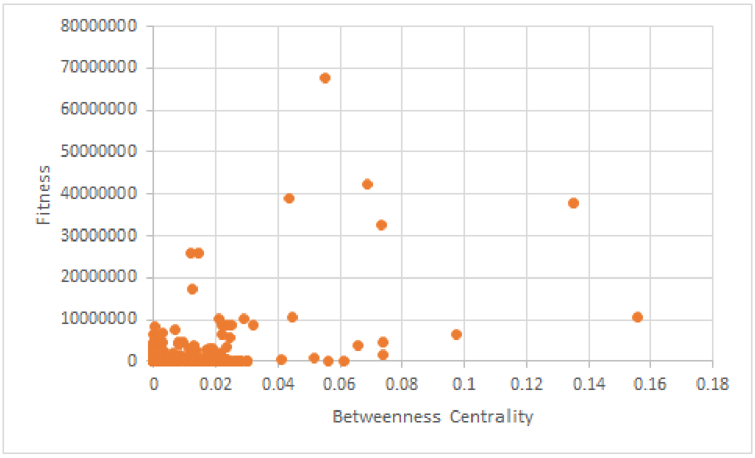


Figure 14: Fitness Betweenness Centrality dependency, $p = 0.3$, $m = 7$

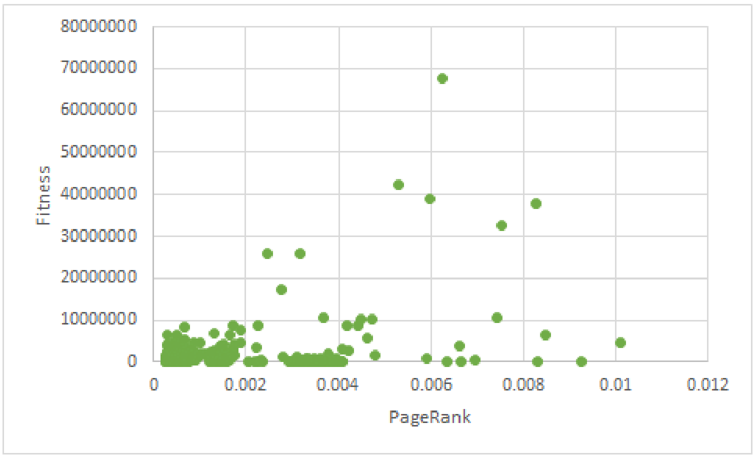


Figure 15: Fitness PageRank dependency, $p = 0.3$, $m = 7$

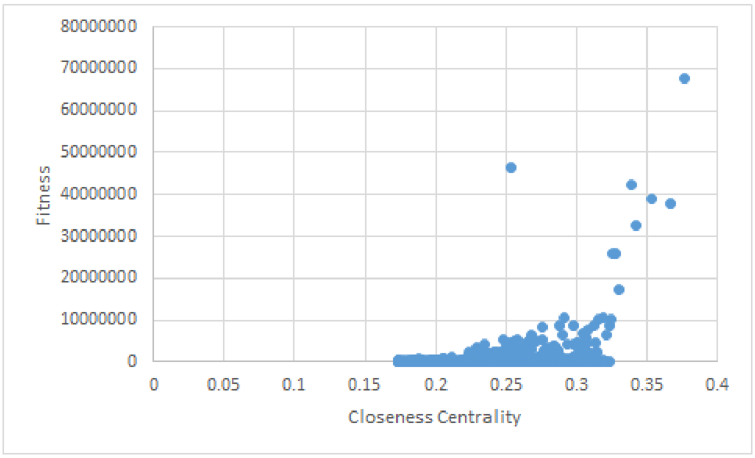


Figure 16: Fitness Closeness Centrality dependency, $p = 0.5$, $m = 7$

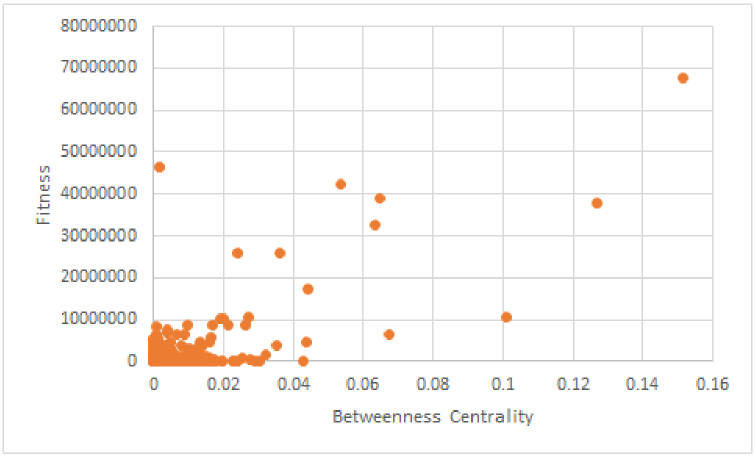


Figure 17: Fitness Betweenness Centrality dependency, $p = 0.5$, $m = 7$

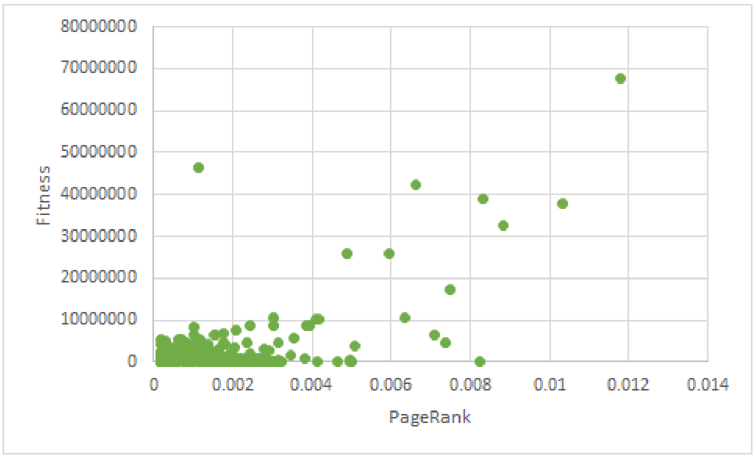


Figure 18: Fitness PageRank dependency, p=0.5, m=7

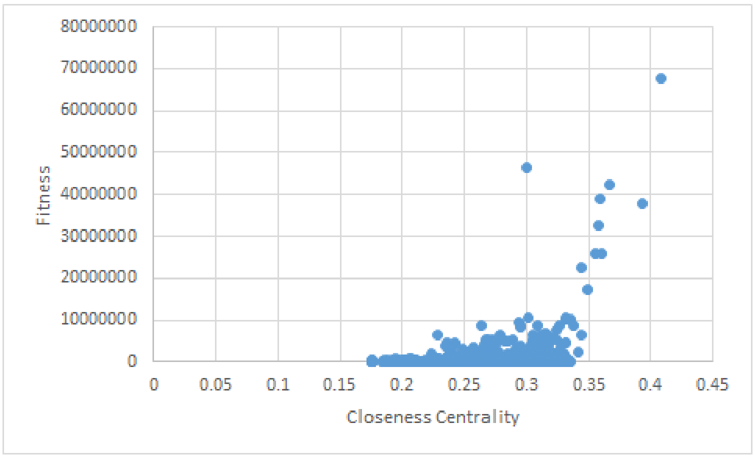


Figure 19: Fitness Closeness Centrality dependency, $p = 0.9$, $m = 7$

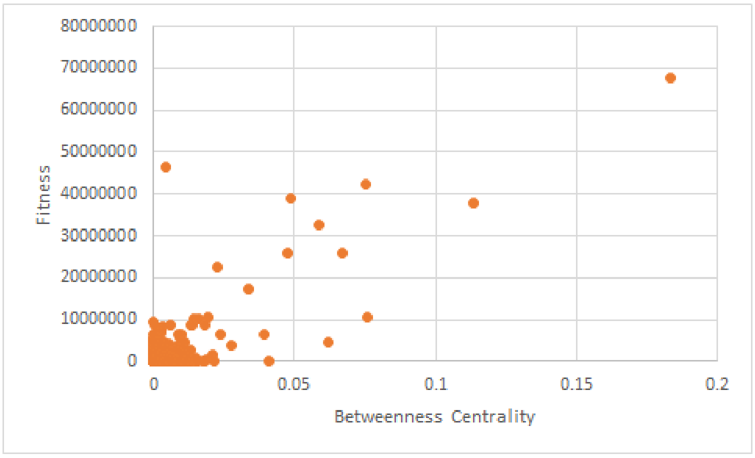


Figure 20: Fitness Betweenness Centrality dependency, $p = 0.9$, $m = 7$

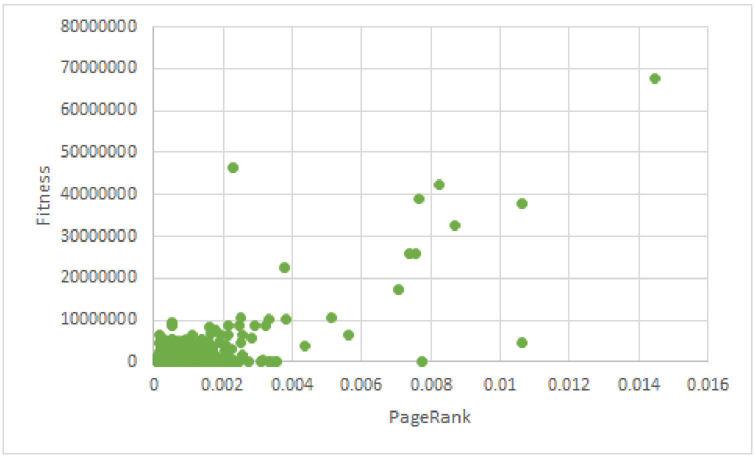


Figure 21: Fitness PageRank dependency, $p = 0.9$, $m = 7$

There are a few nodes in the network, that have more neighbours then others. These nodes are more important and have greater $fitness$ value.

The networks have some interesting properties:

- There are cycles in the network
- Files close to each other in the file system are also close to each other in the network
- Files with higher $fitness$ tend to have more neighbours (their PageRank is higher)

This swarm virus could utilise much more features than it already has at this point. One of them could be adding more subspecies of ants. The virus already contains those that explore file system and transform it to network with some information. But a structure of the code enables the possibility to have more than that. It is not a bad thing to have interface upon each module. It enables us to create move version of them, so there can be more infectors, payloads, triggers and most importantly, more swarming behaviour. Now we can create a different ant species to perform a different role. Some ants could explore, some of them can execute only a part of the payload, some could do a different part of the payload. Some could even explore with different behaviour. It is enough only to add some other signals to specify, which type of module to instantiate or how to set some parameters. Different approach is to mutate a virus instance to achieve the desired result. Embrace polymorphism or metamorphism or just create a different program. It is enough to share information about location of external sources.

6.1 Usage

Usage of such technology seems obvious. You can use this code for malicious purposes of course, but that is no challenge. The real challenge is using it in a different way. You can directly specify a topology of the network by downloading the topology from web. Creating it from *.exe files is up to ants. The only drawback is that an operating system must contain at least as many files as nodes in desired topology. When you have all of that, you can send ants to solve for example a travelling salesman problem or other network related problems. Ants will go through the network just like in ordinary ACO described in chapter 3. The only difference is that nodes will be executable files arranged into the desired topology. It can be solved in a user's machine without his or her knowledge. The virus can just steal computation time for higher purposes. It also could be used against malware, but this is a topic for further research.

6.2 Network Examples

The bigger the node, the bigger its fitness. The darker the node, the bigger the PageRank. Only figures 22, 23, and 24 have these properties.

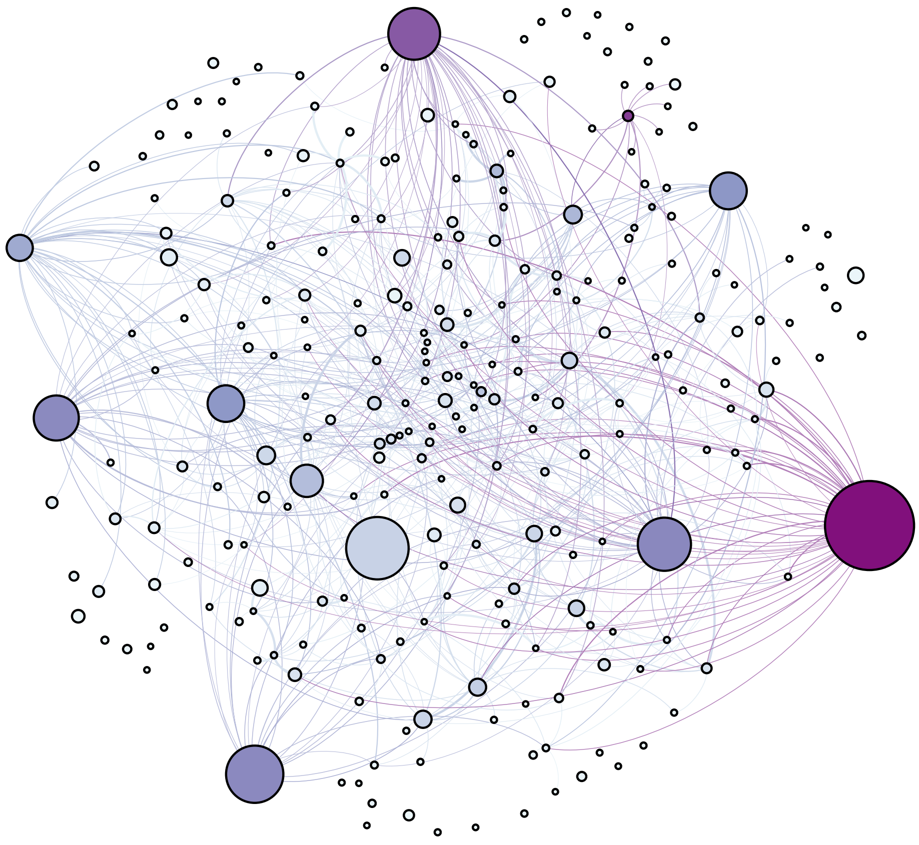


Figure 22: Network example 1, $p = 0.5$, $m = 2$

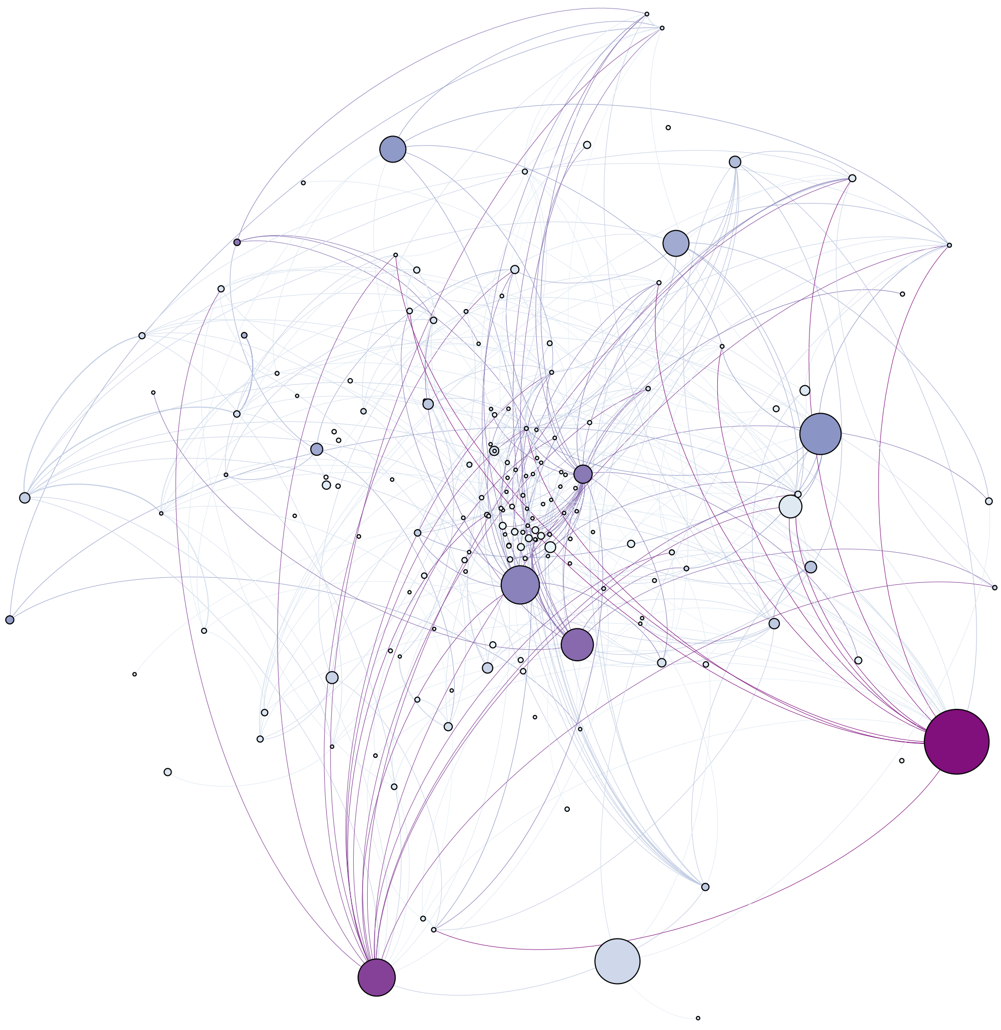


Figure 23: Network example 2, $p = 0.7$, $m = 7$

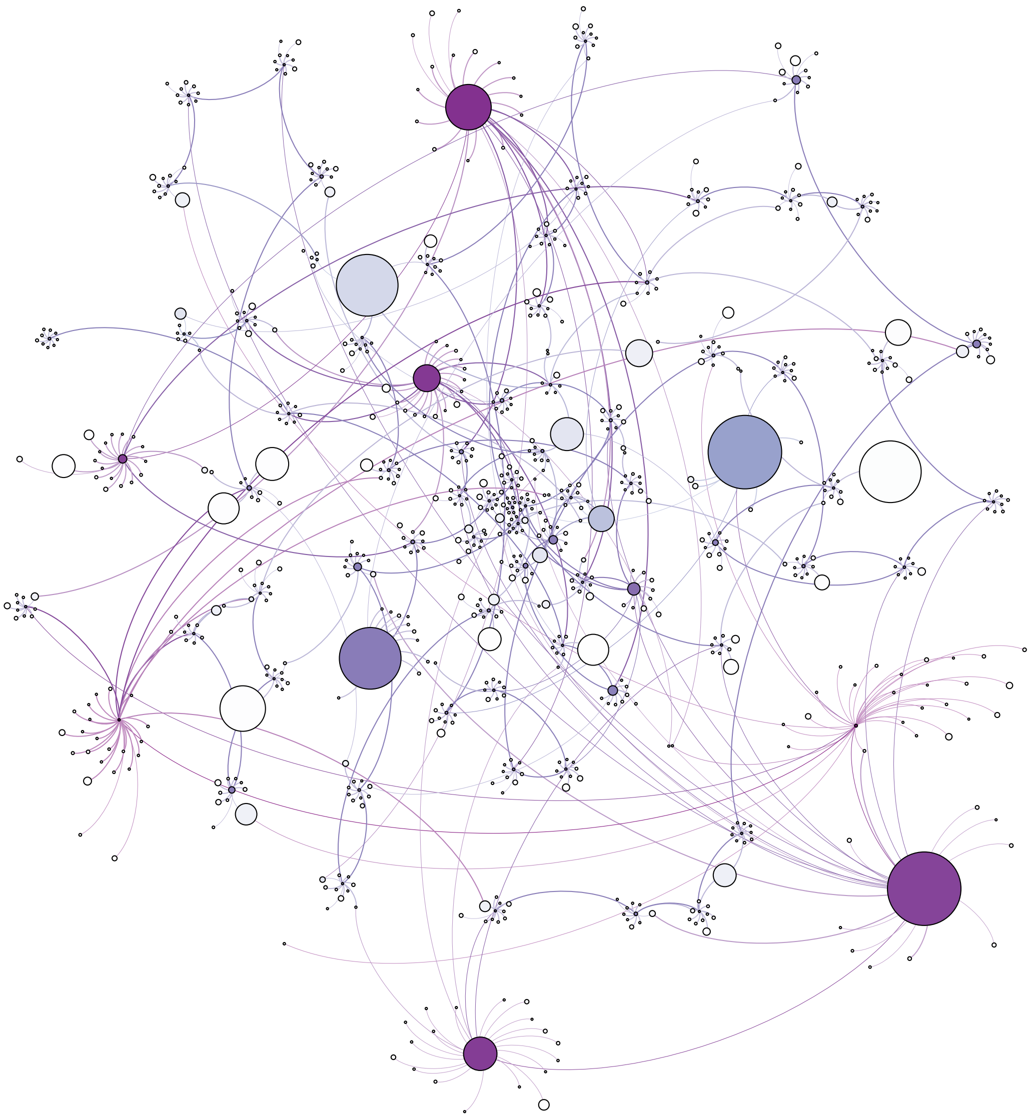


Figure 24: Network example 3, $p = 0.3$, $m = 2$ (unsuitable parameters)

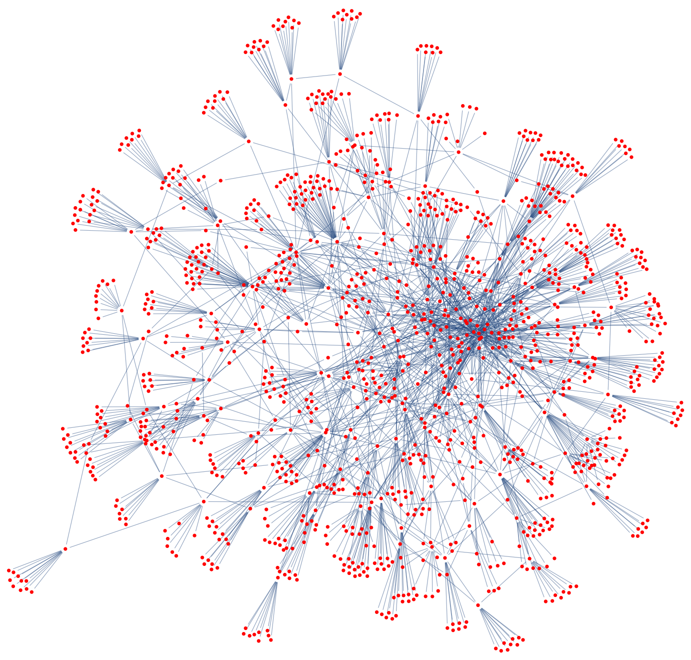


Figure 25: Network example 3, $p = 0.3$, $m = 7$

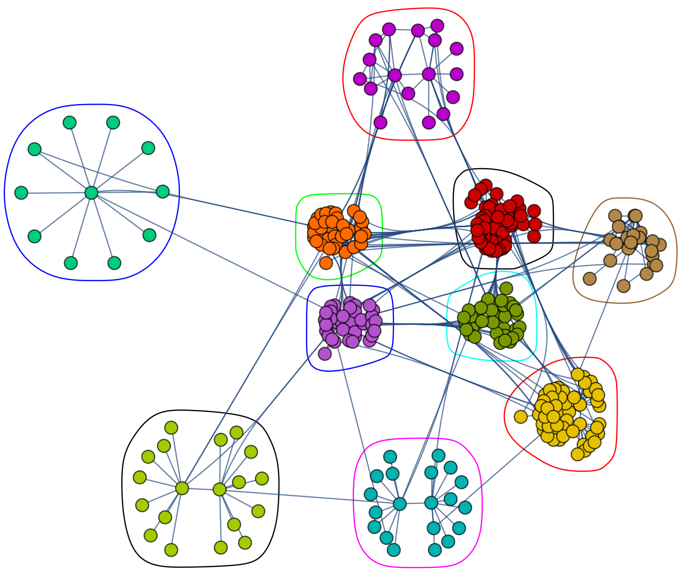


Figure 26: Network example 3, $p = 0.9$, $m = 7$, visualization into communities. (This is just a demonstration of different visualization possibilities of the network)

7 Conclusion

We can use this behaviour pattern to simply explore and map the file system, but for other work like malicious activities we can use different behaviour pattern. If we keep the ants, we can use different commonwealth of them. Let's explain the behaviour of *explorer ants* or just *explorers*. These *explorers* create a network with some properties, meanwhile the *executor ants* or just *executors*, are responsible for a malicious activity. This activity could be crypto-locking, data-manipulating etc. This activity is variable and depends on what evil we as virus developers want to do to the defenceless victim.

When it comes to different behaviour pattern than the ant system, we can use any swarm intelligence behaviour pattern optimised for networks, or devise a new one.

As it was said before malicious activity is not an only activity the swarm "malware" could provide. We can solve any network-like problem at victim's machine without his or her knowledge. Yes, it is stealing computing time, but it could serve the greater good. Another usage of such swarm virus is to observe its behaviour, in order to protect a computer against similarly behaving malware, or such behaviour could be used against a malware in form of antivirus software. But, this is for another study.

The virus structure was composed from states and modules to achieve certain level of modularity. Every module could be exchanged for another to attain different infection mechanism, different payload, different triggers etc.

When it comes to debugging, it is very handful not to spread actual virus into system, but only simulate its behaviour. It was done by using only the module responsible for moving. I consider the work a success. Swarm malware is definitely possible, but, I hope, that the work is not going to be the startup for a swarm malware era, as the virus Brain was startup for a malware era.

References

[1] Merhaut F., Zelinka I., Úvod do počítačové bezpečnosti [Introduction to Comtuper Security], Fakulta aplikované informatiky, UTB ve Zlíně, Zlín 2009

[2] Peter Szor, Počítačové viry - analýza útoku a obrana [Computer Malware - Analysis of Attack and Defense], Zoner Press

[3] Zelinka I., Oplatková Z., Šeda M., Ošmera P., Včelař F., Evolutionary techniques - principles and applications, BEN, Prague, 2008, 598 p.

[4] Holland, John H. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. Ann Arbor: U of Michigan Press, 1975. Print.

[5] Holland J (1992) Genetic Algorithms, Scientific American, July 44 - 50

[6] Fogel, LJ, Owens, AJ, Walsh, MJ (1966), Artificial Intelligence through Simulated Evolution, John Wiley.

[7] Bull, Larry, and Tim Kovacs. Foundations of learning classifier systems. Berlin: Springer-Verlag, 2005. Print.

[8] Baluja S (1994). Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CMU-CS-94-163, Carnegie Mellon University, USA

[9] Lozano, José A., and Pedro Larrañaga. Estimation of distribution algorithms: a new tool for evolutionary computation. Boston: Kluwer Academic, 2002. Print.

[10] Goldberg, David Edward. Genetic algorithms in search, optimization, and machine learning. Boston: Addison-Wesley, 2012. Print.

[11] Rechenberg I (1971) Evolutionsstrategie – Optimierung technischer Systeme nach Prinzipien der biologischen Evolution (PhD thesis), Printed in Fromman-Holzboog, 1973

[12] Schwefel H (1974) Numerische Optimierung von Computer–Modellen (PhD thesis). Reprinted by Birkhäuser, 1977

[13] Dorigo, Marco, and Thomas Stützle. Ant colony optimization. Cambridge (Mass.): Bradford , 2004. Print.

[14] Onwubolu G, Babu B (2004) New Optimization Techniques in Engineering, Springer-Verlag, New York, 167-218, ISBN 3-540-20167X

[15] Hart, William E., Natalio Krasnogor, and J. E. Smith. Recent advances in memetic algorithms. Berlin: Springer, 2005. Print.

[16] Goh, Chi-Keong, Yew-Soon Ong, and Kay Chen Tan. Multi-objective memetic algorithms. Berlin: Springer, 2009. Print.

[17] Schnöberger, Jrön. Operational Freight Carrier Planning. New York: Springer-Verlag Berlin Heidelberg, 2005. Print.

[18] Dasgupta, Dipankar. Artificial immune systems and their applications. Berlin: Springer, 1999. Print.

[19] Castro, Leandro N. de., and Jonathan Timmis. Artificial immune systems: a new computational intelligence paradigm. London: Springer, 2002. Print.

[20] Laguna, Manuel, and Rafael Martí. Scatter search: Methodology and implementations in C. Dordrecht: Kluwer Academic Publishers, 2003. Print.

[21] Clerc, Maurice. Particle swarm optimization. London: ISTE, 2010. Print.

[22] Price K (1999) An introduction to differential evolution. In: Corne D, Dorigo M, Glover F (eds) New Ideas in Optimisation, McGraw Hill, International (UK), 79-108.

[23] Xiaodong, Li, and Andries Engelbrecht. "Particle Swarm Optimization: An Introduction and Its Recent Developments." Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation Londres (Angleterre), July 7-11, 2007. New York, NY: ACM, 2007. N. pag. Web.

[24] Zelinka I (2004) SOMA - Self Organizing Migrating Algorithm, In: Onwubolu, Godfrey C., and B. V. Babu. New optimization techniques in engineering. Berlin: Springer, 2004. Print.

[25] Davendra, Donald, and Ivan Zelinka. Self-organizing migrating algorithm: methodology and implementation. Switzerland: Springer, 2016. Print.

[26] Wolpert D.H., Macready W.G., No Free Lunch Theorems for Search, Technical Report SFI-TR-95-02-010 (Santa Fe Institute), 1995

[27] Bianconi, G., Darst, R. K., Iacovacci, J., Fortunato, S. (2014). Triadic closure as a basic generating mechanism of communities in complex networks. Physical Review E, 90(4), 042806. [https://arxiv.org/pdf/1407.1664.pdf]

A CD/DVD

CD/DVD Contents:

- Swarm malware project – a folder with the Swarm Virus Project containing the virus and the antivirus
- Swarm Malware.pdf – Digital form of the master's thesis