Computer Virus Enhancement

John P Lynch
Division of Computer Science
University of Minnesota, Morris
Morris, Minnesota, USA 56267
lynch446@morris.umn.edu

ABSTRACT

This paper gives insight into computer virus implementation and the applications of evolutionary computation and advanced algorithms. Evolutionary algorithms are created to genetically modify viruses and advanced forms of search. These search algorithms are for finding specific items in a system such as a folder in a file system or an exploited port number on a switch to connect to another computer in a network. The parts and phases of the computer virus will be covered to give the reader a better understanding of how and where the virus may be improved by some form of evolutionary computation. We also discuss advances in anti-malware made to combat the increased strength of evolving malware.

1. INTRODUCTION

Malware can be found worldwide corrupting computer systems. Anti-malware detects, halts, and destroys malware. One form of malware is a computer virus. A computer virus can be something as simple as ad-ware, a type of software used to install pop-ups that don't leave your computer. There are viruses that are more advanced and dangerous to computers and entire systems. These viruses are meant to massively disrupt businesses, schools, government offices, and other major forms of infrastructure.[6] A growing trend in the development of the computer virus is the implementation of evolutionary algorithms and computation which allow the virus to evolve in different ways, becoming stronger and more dangerous. The necessity to research the power of viruses becomes more pressing due to the growth of technological advancement and development in computers.

A virus typically recreates itself in a system over and over again, much like a biological virus, to spread and infect the system. As each new virus is made, the new virus may change its name or activating code to continue to be undetected. A virus may evolve by changing its function to accomplish different tasks much faster. These changes to a virus make it more difficult for anti-malware to combat the infection. Anti-malware will have to change and adjust to keep up with and detect malware that it was previously able to fight.[4]

The rest of the paper is structured as follows: Section 2 provides the background in evolutionary computation and gives basic definitions of terminology related to viruses. This section will explain how malware and anti-malware act on computer systems. Section 3 provides instances of how viruses can evolve and what they are capable of doing. Section 4 demonstrates how a virus may search through a computer system to find specific files and act on them. We include an example of a virus training, which programs the virus to search a computer with knowledge of the system it is infecting. In Section 5 we show how a virus may spread through systems and become harder to detect with evolutionary traits. In Section 6 we describe how anti-malware works on a common virus in a computer system. Results are given from another team's work on capturing evolving and non-evolving viruses. We demonstrate anti-malware's capture rate of evolving viruses to see how a virus can be fought when it evolves and what changes anti-malware must go through to fight evolving viruses. We conclude with ideas of how evolutionary computation may become a cornerstone in research against malware.

2. BACKGROUND

In order to better understand the content, definitions will be given to terms applied throughout the paper.

2.1 Viruses

A virus is just one form of *malware*, software created to act maliciously on a computer or system of computers. A *computer virus* is a malicious software program that replicates itself to perform disruptive or destructive tasks on a computer system in related space on the system. Computer viruses have different phases and parts that are active throughout their lifespan.

Computer viruses infect a system in four phases. The *dormant phase* is when the virus may be in a system but is taking no actions or functions yet. It is waiting for the *trigger phase* of the virus, which is the initializing code, to perform its functions. The trigger can be a timer or an execution of detected events. These events can be something as simple as a changing condition such as a time on the internal clock, or a user double clicking an application holding the virus. The *propagation phase* of the virus is when the virus is replicating itself and triggering the new virus as it is created. The *execution phase* is when the virus' intended functions are performed. The virus may delete many local

UMM CSci Senior Seminar Conference, December 2016 Morris, MN.

files, aside from itself, or try to continue to propagate and slow a system down by using disk space and power.

The parts of the computer virus include portions of code and methods to complete the virus' intended functions. The *infection mechanism* of the virus is the function of reproducing itself into system areas. This is usually a simple search algorithm that attempts to find all other folders on a disk and copy itself into the folders before activating its new copy. The *payload* of the virus is considered the malicious portion of the virus. The payload may be a copying method to steal information or a destructive bit of code to modify a system. The *trigger* is the initializing code or feature that delivers the payload portion or starts the execution phase of the virus.[6]

2.2 Anti-malware

Anti-malware is software to detect, prevent, or 'cure' malware. This is a catch-all term for software on a computer that is meant to stop malware from disabling or damaging a computer. A firewall is a form of anti-malware that is a preventive security system. A firewall monitors and controls what goes into a computer system or multiple systems. The most common types of anti-malware are anti-virus scanners which are general applications that protect against all sorts of malware, but primarily scan for threats already on the system.[6]

2.3 Evolution

A computer virus evolves through several methods. *Genetic algorithms* are a type of evolutionary algorithm based on natural selection relying on mutators to change and reshape the formations of each new population. Taking a biological analogy, it is simple to understand evolution in computer science. A *population* is a particular section or group of individual code. We make a distinction between viruses by generations. One group of viruses are different from the next groups that they create like in generations of families. When a subsequent virus is made, that created virus may be more efficient or faster at accomplishing a set task than its predecessor. *Fitness* is the measurement that determines the increased speed or efficiency of a virus compared to its previous generation. *Mutators/genetic modifications* are operators creating genetic diversity. They are used in genetic algorithms to change selected portions of code the algorithm recreates.

There is more than one way to evolve viruses. You may mutate the code that controls their ability to search a computer system with training. *Advanced search (with heuristics)* are simple rules applied to some versions of search for finding data in structures or systems. This is done by training a virus to search through systems over and over until they see a pattern that they consistently rely on for better speed. For the biological analogy imagine a species that relies on how fast they may catch their prey. If the previous populations of the species evolve future populations to use specific techniques to catch prey faster, the new population becomes the generation that survives and continues to repopulate with those techniques. To apply this to a search algorithm we take a virus and its ability to find a program file. We determine how many folders a virus searches through to find the file and we create a new virus. If the virus learns which folders to search through first, that virus searches through fewer folders to find the target file. We determine the search speed of the two viruses and whichever is faster continues to make more "children".[1]

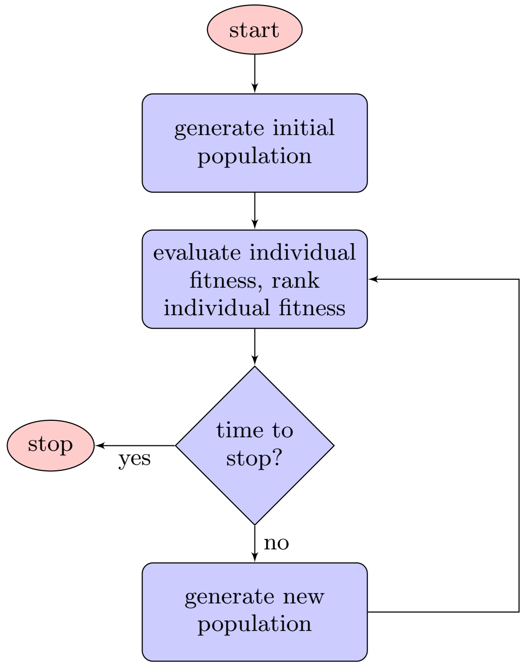
start
generate initial population
evaluate individual fitness, rank individual fitness
time to stop?
yes
stop
no
generate new population

Figure 1: evolutionary algorithm basic structure, based on [1]

2.4 The Example Virus

For a constructed example, throughout the paper we will be using a '.bat' file named "supahVirus.bat". This '.bat' file will be built to infect some average older operating system of Windows such as Windows Vista Home. A '.bat' file is a batch file that works to change a system using code written directly to a command script. To illustrate what the virus is aiming to accomplish, consider some network of computers where there is a folder named 'secretFolder' that we want to collect in each system on the network. Without access to these computer's inner files, we create this virus and attempt to infect these systems. Throughout the paper there will be examples of pseudocode for this virus. The pseudocode is built from basic functions taken from no particular source and is for batch files that can directly act on a system through terminals. The pseudo-code is based on the language python for clarity.

3. GENETIC CHANGES TO A VIRUS

Genetic modifications are used to create different formations of code via genetic algorithms. These algorithms define and refine different strains of viruses to create new strains that can infect and propagate themselves over and over. One major factor in the measured strength of computer viruses is their ability to self replicate using the infected system. Viruses become stronger with genetic modifications, and evolve to execute tasks that would otherwise be too difficult for one model systems to accomplish. An example would be "supahVirus.bat" infecting systems that are incapable of having a user interface due to some broken system functions or just by making a program run a black screen.

3.1 Structure of evolutionary algorithms

Evolutionary algorithms are based on the idea of evolving data similar to biological evolution. Observing Figure 1, we create an initial population of data for accomplishing a task we desire using that algorithm. For our example we design an algorithm meant to create a virus with a new name. Next, we evaluate the "fitness" of each set of data, defined as the effectiveness to accomplish the task. To evaluate fitness we create scenarios or tasks we want our population to accomplish. For the changed name we may have some comparative algorithm to determine how closely related the two names are. The more varied the name from the starter name, the stronger "fitness" of the virus. Then, we take measurements of how well the individuals in the population accomplished those tasks. After ranking the population we take the strongest of the population and attempt to mutate the data using changeable genes. The selection of genes is based on code of the virus that can be changed to accomplish a task in a different way. This could be parts of the code of a virus such as a 'for' loop or some data structure the virus holds for searching. Because our code creates populations with different names, the characters of the name in each generation will vary. Each new generation will have more variation in the names than the previous generation until they no longer resemble the initial population's names. Figure 1 shows the basis of an evolutionary algorithm.

3.2 Propagation phase of the virus

The propagation phase of the virus as previously explained is the phase in which the virus self-replicates and spreads. The virus will possess some action, or method, that it activates to begin replicating and spreading. This means that our "supahVirus.bat" will have a method called "swarmMethod". This method will repeatedly recreate the same computer virus in different parts of the system which may prove problematic for the system. Most anti-virus software that attempts to prevent a virus from "reproducing" looks for repeating programs that seem malicious. A possible solution for the virus to avoid being found by anti-malware is to change the virus each time so that it is not recreating the same code, names, and files that the anti-malware would find. The anti-malware would only see that files and code are being written, and not that the same files and code are being recreated. This modified viral propagation could use code that makes the virus harder to be observed by anti-malware and more powerful each time it is created.

3.3 Genetic algorithms and mutators for malware propagation

To create a genetic algorithm that will mutate our virus each time it is created, we begin with the foundation of the formula that will go into "supahVirus.bat". A "true" genetic algorithm is supposed to more accurately portray a Darwinian scenario utilizing parent variants of the stronger versions of modifications. A "child" is then created by the two parents' combinations which would share overlap, and randomly choose between the two parents' differences. For the sake of simplicity this paper will be using a simplified version which creates very minimal changes to genes while still accomplishing the designated task of our "supahVirus.bat". This pseudo-code for mutation is based on the structure of evolution.

```
def swarmMethod():
  initialize(thisVirus)
  powerOfThisVirus = evaluate(thisVirus)
  while true:
    mutatedVirus = mutate(thisVirus)
    powerOfMutation = evaluate(mutatedVirus)
    if powerOfMutation >= powerOfThisVirus:
      initialize(mutatedVirus)
```

To measure the power of this computer virus, we only need to take into account the speed of successfully creating subsequent viruses and the speed of running the code of our virus. To check for this speed we created a method "evaluate" in our code that simply runs and checks if the virus created would work properly and how fast the virus works in initialization.

One advantage of stealth for a virus in a computer system is that it is relatively easy to trick basic anti-malware. Something as simple as a method to change names will allow new generations of viruses to gain mild variations on names such as "supahVirus" changing to "superbVirus". These changes in name allow a virus to successfully hide from most anti-malware that simply detects repeating file and program names.[4] The team that created evolvable malware started with a simple "bagle" worm that was meant to represent a virus. From this virus they showed that splicing in variant genes created different viruses that became harder to detect by their test module. The test module was made up of different anti-malware programs that had progressively increasing efforts dealing with variations of the virus. These variations included how a virus may execute applications to conceal themselves from user suspicion or additional functionalities like killing anti-virus software or changing names to hide from anti-virus software.[4]

This method should bring about multiple changes from each new mutator to create multiple viruses. These new viruses are created to function similarly to the original virus and are as fast or faster. These viruses are selected to be initialized throughout the computer system in different areas. We create a simplistic check that allows us to determine the speed of the virus. The problem with this check is that it is typically based on the size of the virus created in code. The check would simply be whether the size of the code remains the same and wouldn't be larger than the initially tested against population. One virus may have a mutator which changes the function of copying various files in the folder it was initialized in, and it would instead start copying and removing those same files before sending them to other areas in the system. Another possible mutation that anti-malware may not detect is the different ways that the virus may use the computer's systems to perform its actions. This means that one version of "supahVirus.bat" may produce a terminal and begin searching through the file systems, while another version could hijack the Windows Explorer and directly act like a user or administrator to the file systems.

For the sake of simplicity, "supahVirus.bat" will only produce one mutation based on the characters in their own name. This will simplify our check, which will verify characters are actually different and do not match the characters from the virus that created it. The "swarmMethod" of our virus would not have a check for power of mutation, but rather would perform a check for characters in its own name.

4. INFECTION AND SEARCH IN A COMPUTER VIRUS

A very important operation of the virus is to search a computer system for different files, folders, or disks to infect and continue propagation. To properly propagate it is required that the virus spreads so that it does not simply stay at the point of infection. To search for the “secretFolder” we require the operation of the “supahVirus.bat” to recreate itself in each new folder. Eventually the virus finds and starts copying the contents of “secretFolder” after it propagates into it.

4.1 Infection mechanism/vector

In all computer viruses there is a mechanism that typically uses a search routine to find new files or disks to infect. This portion of the virus doesn’t just seek to copy itself, although that is the intention of the infection mechanism’s other method, but also to spread, so that it can properly disperse copies into other parts of the disk. To accomplish dispersal to propagate the virus, “supahVirus.bat” will require another method that we will call “mechanismMethod”.

```
def mechanismMethod():
  while true:
    for f in folders:
      if(f.name == "secretFolder"):
        copyAndTransfer(f)
        break
      else:
        mutatedVirus = swarmMethod()
        into(initialize(mutatedVirus), f)
```

This infection mechanism will search through top level “folders” and attempt to go into each file system and initialize the newly created mutated viruses in each. To properly do this we need to access the top-level of the file system. This is much more easily accomplished by a “.bat” file, given that it can open and write to a terminal, which can write and run code to the disk and the highest priority folders that can be infected. This method was improved by Sadia Noreen using a worm malware called “bagle”. The method of infection is one of the parameters that they tested to see how a virus may be caught on different levels. [4]

4.2 Heuristic search

Our heuristic search prioritizes systems that we can infect and ignores systems that don’t match our preferred system. For “supahVirus.bat”, when the virus is in the dormant phase, before actually beginning to infect systems, we want to seek out the highest system or folder that has the most vital information that we want to affect. This is done by our “.bat” file writing the actual payload of the virus delivered to the highest writable folders of the disk. The virus initializes and starts in top level folders before propagating into lower level folders in its file system. Each new virus in a folder then infinitely recreates a virus into each folder it can find recursively.

To speed up the process the heuristic that we include in our infection mechanism method “mechanismMethod()” will be one that prioritizes the list of folders most likely holding the “secretFolder”. To do this we create a data structure which would be a map of folders to search through in our ‘for’ loop that automatically prioritizes those folders for having a higher chance of containing the target folder based on the past experience of the virus on similar systems. To change the code we simply change ‘folders’ in the ‘for’ loop to the data structure with priority map values applied to each folder which will most likely contain the target.

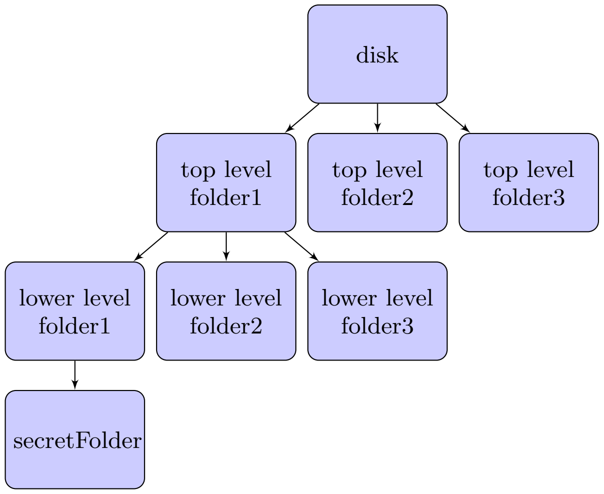

Figure 2: computer search

```
def mechanismMethod():
  priorityfolders = priorityMap(heuristic)
  for f in priorityfolders:
    if (f.name == "secretFolder"):
      copyAndTransfer(f)
    else:
      swarmMethod()
      into(swarmMethod.mutatedVirus, f)
      initialize(mutatedVirus)
```

For the priority map to search for more desired folders we create the heuristic based on testing done using computers that would have a “secretFolder” on some level of the computer. Suppose we know what folders generally contain the “secretFolder”, those folders would be the folders that appear in the heuristic to be at the front of the loop when the mechanism attempts to propagate. Testing done to these computers would have our virus learn the most common folders to attack and be the starting heuristic to create the priority map that would target which folders each virus should propagate first.

This method increases the average search speed through possible folders by using knowledge of the most common locations for the target folder. The upper bound remains the same as all possible ‘for’ loops can be executed before finding the target folder. The lower bound decreases to the shortest possible path to the target folder from the disk. The average speed increases to become closer to the lower bound. Each new virus prioritizes which folders may hold the “secretFolder” and may reduce the search time to become near linear.

Observing Figure 2 we prioritize the top level folder to propagate into folder 1 and continue to lower level folders. When that newly created virus is pushed into “top level folder 1”, it initializes and creates another priority map based on its knowledge, until it finds the lower level that holds the “secretFolder”.

SPLIT.EXE	
offset interval	(0,43000)
Evaluations	300
Type I(zones found)	1
Type I(largest)	334
Type II(zones found)	32
Type II(largest)	1,511

TESTDISK.EXE			
offset interval	(0,43000)	(0,10000)	(0,2000)
Evaluations	15,000	2,000	300
Type I(zones found)	-	1	1
Type I(largest)	-	33	25
Type II(zones found)	3	4	3
Type II(largest)	179	167	183

Figure 3: Capture results taken directly from [2]

5. PARALLEL SECURITY TO MALWARE GROWTH

Through various forms of evolutionary computation and training, viruses may advance in ways uninhibited by anti-malware programs. Supposing the owners of the network want to eradicate the new "supahVirus.bat" that is evolving on the system, they may have to change strategies on how to deal with the evolving virus.

Much like in viruses, there are advanced forms of computation that, when applied to anti-malware, can substantially improve the anti-malware's performance. By using detection of signatures of code, and not just the forms of the code, we gain insight into how to find evolving malware. In adaptive systems, the advanced anti-malware methods applied by the researchers Xu and team[7] showed that there are ways to control computer virus propagation across a network. This anti-malware methodology can also be applied to personal systems for detecting machine generated viruses.

5.1 Detecting machine generated malware

To properly demonstrate machine generated malware detection, the testing done by Andrea Cani and team [2] began with the "timid" trojan horse having several generations that all evolved in mass quantities throughout a simulated system. To find the trojans in the simulation, the anti-malware used a form of signature detection. *Signature detection* is when anti-malware collects samples of what it believes to be malware and looks for consistent repeating patterns in code. This approach takes a sample of the detected virus and begins testing against the different files in the system until it finds matches. The matched programs are halted and counted towards the number of detected malware. To ensure that the code wouldn't copy the signature of something insignificant, the code sample used as a signature to test against the other files and viruses would have to be some payload portion of the virus. The payload is the 'logic bomb' of the virus that actually is the code to activate the changing or 'malicious' method or mechanism in the virus. For our "supahVirus.bat" this would be the previously created method "swarmMethod" that copies the virus and initializes it in whatever folder it is called into. For the team, a simulation proved successful after each generation was taken and computed for each new form of the detectable signature. New signatures were added to the anti-malware's pool of signatures to detect a virus.

A trojan horse is not actually a virus but acts similar to one in its malware content. Rather than searching for a possible exploit or breach, it relies on user error which would allow the user to trigger the malware. This can be done with a simple prompt that the trojan shows on screen with the user clicking or opening the content of the file to activate it. A virus continues its existence via spreading from any active point constantly. This is how a trojan differs from the virus. For the sake of simplicity a trojan horse can become a virus by simply recreating itself in a system that it infects.

According to this testing done by Andrea Cani in the paper on automated malware creation [2], using the trojan horse called "timid" and genetic algorithms created different generations of the modified trojan horse. These generations were activated in a computer system with several files to have code directly injected into by "timid". According to Figure 3, there are two main programs injected by the trojan that anti-malare attempted to detect the injected trojan horse code. "Split.EXE" was an executable program that split files of any kind into smaller parts to rebuild the original and analyze it. "TestDisk.EXE" was another executable program that recovers data and attempts to detect whenever a program is editing data recovered or backed-up. The team ran several tests that each ran for 30 minutes each. The trojan horse attempts to inject its code into '.EXE' and '.COM' files on a computer system. After 6 generations of the virus, 4 anti-malware programs attempt to clean all files of unwanted code.

Figure 3 shows the results of how many zones and how much code was injected into the programs and remained hidden from anti-malware. These viruses would attempt to infect and write themselves into two different types of areas. 'Type I' areas are areas that an execution would usually skip over. These areas would typically be spaces in between functions and code that would be forgotten after an execution. 'Type II' areas are uncalled functions and space in code that are almost never executed at all. Zones are how many areas there is a clear injection of foreign code from timid. Offset Interval is a determined space of how much the "timid" virus blocks code in bytes. The smaller the second number in offset interval, the more it observes smaller spaces in code as individual code blocks to judge as potential areas to infect. Evaluations are the number of times "timid" reran to create generations and look for areas to inject itself into the files.

For "Split.EXE", there was 1 zone of 'Type I' that had the largest block of injected code take up 334 bytes of space. There were found to be 32 zones of 'Type II' where the largest block of code held 1,511 bytes. "TestDisk.EXE" appeared more resilient to attacks by only having 3 "Type II" areas be exploitable in a larger offset. More tests were run on "TestDisk.EXE" as "timid" was forced to observe in more small areas at a time. This would allow more observation in smaller chunks for potentially vulnerable areas to inject the virus' code into. After shrinking the offset interval, "timid" was capable of infecting more areas, including 'Type I' zones. The largest area for injected code was still only 183 bytes in a "Type II" zone. This testing still proved that machine generated evolving malware was useful in determining areas at risk of attack. [2]

5.2 Modeling timing parameters for virus generation

For creating anti-malware that seeks to slow, if not halt, a virus in a network, there are two unique parameters to consider when adjusting for large networks to control virus populations. According to a research paper written by Yang and Chenxi Wang [5], the two most powerful methods for halting virus growth in a system or network are as follows; infection delay based on systems throttling computational power when a virus is detected, and the vigilance of a user of a system or systems in a network. Throttling the CPU of a system slows the virus and its ability to propagate. After slowing the system down the anti-malware can begin searching the system for the virus and all copies of it. This requires the anti-malware be outside of common system use to find disk use and change it. The anti-malware may gain access to the processes of the computer and halt processes that it detects to be malicious. The effectiveness of user vigilance is usually based on how fast a user can detect their system's infection and their course of action. The action can range from opening an anti-virus scanner to switching off the computer and running it outside of local or global networks to stop the virus from spreading to other systems.

5.3 Adaptive network thresholds and control

Despite the increased speed of the new viruses, there are ways of controlling a network infected by computer viruses. According to research done by Shouhuai Xu and team [7], there are strategies that would allow the conditions of servers susceptible to virus attacks to be better protected. Adaptive network control uses two strategies: semi-adaptive and fully-adaptive defenses. The semi-adaptive scenario is based on the assumption that there is a virus in the network that is spreading and must be mapped to predict where it will spread next. The fully adaptive defenses are predicated on no input parameters including the threshold of when and where a virus is likely to spread. The research concluded that there were fully and semi adaptive defense scenarios that allowed the control or destruction of the virus in a network.

To gain some perspective on this we will use our virus, "supahVirus.bat", in our example network. The virus is in a portion of computers on the servers. For some amount of servers/systems 'X' there are X_i infected systems connected to another list of systems X_v that are vulnerable. Using semi-adaptive defenses, an anti-malware program can tell what servers are infected by "supahVirus.bat" and can add them to X_i thus strengthening the vulnerable servers X_v to detect when a new unwanted program is initialized. This semi-adaptive scenario continues on each accessible system in 'X' to clean each system in X_v of detected unwanted programs. The anti-malware would detect the programs using the CPU excessively and then clear running programs it deems unnecessary or malicious. From the scenarios presented, there would eventually be a state where all X_i are cut off from from each X_v space, creating defensive-like battle lines against a virus. This process is explored more deeply in the research paper by the Xu team. [7]

6. CONCLUSIONS ON ADVANCED MALWARE AND ANTI-MALWARE

Evolutionary algorithms and advanced computation in different viruses and anti-malware create new directions for both types of software. From given examples and dissected processes we have demonstrated that there are applications for advanced and evolutionary computation, and those applications can have benefits to both computer viruses and anti-malware. Despite our virus not thoroughly demonstrated to optimally increase the speed of the malware, a virus still benefits in its ability to hide from anti-malware.

7. REFERENCES

[1] T. Bäck. Evolution strategies: Basic introduction. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '13 Companion, pages 265–292, New York, NY, USA, 2013. ACM.

[2] A. Cani, M. Gaudesi, E. Sanchez, G. Squillero, and A. Tonda. Towards automated malware creation: Code generation and code integration. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 157–160, New York, NY, USA, 2014. ACM.

[3] Y. Kandissounon and R. Chouchane. A method for detecting machine-generated malware. In *Proceedings of the 49th Annual Southeast Regional Conference*, ACM-SE '11, pages 332–333, New York, NY, USA, 2011. ACM.

[4] S. Noreen, S. Murtaza, M. Z. Shafiq, and M. Farooq. Evolvable malware. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 1569–1576, New York, NY, USA, 2009. ACM.

[5] Y. Wang and C. Wang. Modeling the effects of timing parameters on virus propagation. In *Proceedings of the 2003 ACM Workshop on Rapid Malcode*, WORM '03, pages 61–66, New York, NY, USA, 2003. ACM.

[6] Wikipedia. Computer virus — Wikipedia, the free encyclopedia, 2016. [Online; accessed 10-October-2016].

[7] S. Xu, W. Lu, L. Xu, and Z. Zhan. Adaptive epidemic dynamics in networks: Thresholds and control. *ACM Trans. Auton. Adapt. Syst.*, 8(4):19:1–19:19, Jan. 2014.