Towards Automated Malware Creation: 
Code Generation and Code Integration

A. Cani, M. Gaudesi, E. Sanchez, G. Squillero,* A. Tonda*

January 25, 2014

Abstract

The analogies between computer malware and biological viruses are more 
than obvious. The very idea of an artificial ecosystem where malicious soft-
ware can evolve and autonomously find new, more effective ways of attacking 
legitimate programs and damaging sensitive information is both terrifying 
and fascinating. The paper proposes two different ways for exploiting an evo-
lutionary algorithm to devise malware: the former targeting heuristic-based 
anti-virus scanner; the latter optimizing a Trojan attack. Testing the stability 
of a system against attacks, or checking the reliability of the heuristic scan 
of anti-virus software could be interesting for the research community and 
advantageous to the IT industry. Experimental results shows the feasibility 
of the proposed approaches on simple real-world test cases. A short paper 
on the same subject appeared at the 29th Symposium On Applied Computing 
(SAC’14).

1 Introduction

Malware is a collective noun denoting programs that have a malicious intent – the 
neologism standing for mal-icious soft-ware [11]. Specifically, malware usually de-
notes hostile, intrusive, or simply annoying software programmed to gather sensitive 
information, gain access to private systems, or disrupt the legitimate computer op-
erations in any other way. Since computer technology has nowadays emerged as 
a necessity in various aspects of our day to day life, including education, banking, 
communication, and entertainment, the threat posed by malware can’t be over-
looked.

The most popular form of malware is represented by computer viruses, a term 
coined by Fred Cohen in 1983 [5]. Viruses are programs able to replicate themselves 
and infect various system’s files. As many other in computer science, the idea of self-
replicating software can be traced back to John von Neumann in the late 50s [22], 
yet the first working computer viruses are much more recent. Creeper, developed

*This document is available at http://www.cad.polito.it/downloads/

*Contact authors: giovanni.squillero@polito.it, alberto.tonda@grignon.inra.fr
in 1971 by Bob Thomas, is generally accepted as the first working self-replicating computer program, but it was not designed with the intent to create damage. On the other hand, the virus *Brain*, written by two Pakistani brothers and released in January 1986, is widely considered the first real malware [4].

Since the late 90s, computer malware creation has emerged as a commercial industry with revenues skyrocketing to several million dollars a year [1]. Programs that fight malware are generally called anti-virus. However, nowadays, the majority of threats are not posed by viruses themselves, but by worms and trojans. The former are self-replicating software able to send themselves to other computers on the Internet or a network; the latter are software that emulate the behavior of an authentic, legitimate program but also perform some fraudulent, hidden action.

Several terms may be used to describe specific malware, denoting their purpose, replication strategy or specific behaviors. These terms are clearly non-orthogonal, and the same program may be described by several of them. A spyware is a software which is installed on a computer system to hijacks his personal and confidential information. A keylogger is a particular type of spyware that records user interactions, trying to steal passwords or credit-card data. A rootkit is designed to take control of infected machine by gaining administrator access of the system – the name comes from the term root under UNIX. A dialer is a program that connect the telephone line to a fraudulent provider. A botnet is a remotely controlled software, and the machine under control is sometimes called a zombie. Adware is generic advertising-supported software whose functionality is to displays or downloads the advertisements to a computer.

Another common classification distinguishes five different generations of computer programs [18]: first generation malwares cause infection by simply replicating their code into other software; second generation have additional functionalities, such as the ability to identify files already infected (self-recognition); the third generation marks the appearance of stealth techniques to avoid detection by anti-virus software; the fourth-generation malware possess armoring procedures specifically designed against removal and analysis (anti-anti-virus techniques); finally, malware of fifth generation applies algorithms to obfuscate its code with every replication, altering the program structure itself.

While the anti-virus industry is able to counter most of the menaces few days after they appear, creating tools for protecting users against 0-day malware, that is, unknown threats as soon as they appear, is a taxing problem [2].

This paper proposes to exploit an Evolutionary Algorithm (EA) to create, or rather, optimize, malware. The EA is used with two different goals: to make malware undetectable by existing anti-virus program; to optimize the injection of the code inside a given host, creating a Trojan horse. Producing new malware with negligible human intervention could be extremely advantageous to anti-virus producers to test and enhance their products. Moreover, the creation of trojans could also be used, for example, to test the security of computer infrastructures.

The paper is organized as follows: Section 1 introduces the paper; Section 2 sketches the essential background information; Section 3 illustrates the goal of the research; Section 4 reports an experimental evaluation; and Section 5 concludes the paper, outlining future works.

2 Background: Stealth and Armoring Techniques

Anti-virus programs are designed to detect and remove all malicious applications, assuring the continued integrity of a system. Nowadays, these applications are perceived as needed in Windows-based operating systems (OSs): new versions of this OS include anti-virus software directly in their basic distributions. Despite Mac owners sense of security, recently, all main anti-virus companies also launched products targeting specifically Mac OS. Still to a lesser extent, the same phenomenon is visible for Linux.

The most simple, and widely used, technique for detecting malware is to recognize specific fragments of code. This approach is called **signature-based detection**, because the comparison is performed calculating and checking specific signatures. While moderately efficient, this approach suffers from important drawbacks. The more obvious is that to be effective, the specific malware must be already known, its signature analyzed, recorded, and provided to end-users. However, this can be hardly achieved because tens of thousands of new malware appear every single day\(^1\).

The traditional viruses are able to infect executable programs by appending their code to the existing one [9]. This kind of attacks may be easily detected because the size of the compromised program is modified. In order not to increase the host size, the **cavity viruses** such as CIH or Chernoby1, both appeared in 1998, infect a program by overwriting its code. This was possible due to the many empty gaps in the file format for executables and object code called **Portable Executable** (PE). All these malwares are easily detected resorting to signature scanning, and was able to spread only thanks to the relative slowness in the process of getting virus samples and delivering the new signatures to end users.

From the 3\(^{rd}\) generation, malicious programs start hiding their own code from scanners through encryption [23]. The virus called **Cascade**, appeared in 1986, adopted a symmetrical encryption/decryption routine based on a XOR cipher. Despite its simplicity, the ploy was demonstrated quite effective: the decryption routine was so small that caused several false-positive, and even when it was correctly detected, the anti-virus programs were unable to discriminate between the different strains of the virus.

Concurrently to improvement in signature-based detection, new stealth strategies were devised by malware authors, namely: **oligomorphism**, **polymorphism** and **metamorphism**. Oligomorphic malwares mutate their own decrypt routines. An example of such technique is a virus called **Whale**, first detected in late 1990. Whale, as the early oligomorphic malwares, could generate at most a few hundreds of different encryption schemes, and thus could still be detectable using signatures. Newer ones created decrypt routines dynamically, making harder for the anti-virus vendors to write comprehensive signatures able to catch all variations. Indeed, history showed that it was practically infeasible to catch every new strain of oligomorphic malwares using simple signature.

---
\(^1\)Panda Security claims to have detected 27 million new forms of malware in 2012, an average of 73,000 per day. According to Kaspersky Lab, nearly 200,000 new malware samples appear around the world each day, while, probably due to a different classification, McAfee reports the slightly smaller figure of 46,000.
Polymorphic malwares are even more able to escape signature-based detection. A polymorphic engine is able to create many distinct encryption schemes using obfuscation methods, such as dead-code insertion, unused register manipulation, or register reassignment. The *Tequila* and *Maltese Amoeba* viruses caused the first widespread polymorphic infections in 1991 [20]. In both oligomorphic and polymorphic malwares, the code may change itself each time, but the function, that is, the semantic, is never modified. Thus, anti-virus programs may simulate the execution of a potential malicious application to recognize the pattern of operation.

To thwart such analyses, metamorphic malware denotes polymorphic programs able to rewrite their own polymorphic engine. This is usually performed translating their binary code into a temporary representation, editing such representation of themselves, and eventually translating the edited form back to machine code [13]. A well-known example is the virus *Simile*, also known as *Etap* or *MetaPHOR*, appeared in 2002. Even more advanced stealth techniques are permutating malware, that, instead of generating new instructions like polymorphic programs, modifies existing ones, and thus does not alter their size. The term is used by the Russian virus writer *Zombie* in a set of articles appeared under the title *Total Zombification*.

In 2002, the same Zombie created the *Mistfall engine* using code integration, a technique similar to metamorphism applied to cavity viruses. The Mistfall engine decompiles PE files; moves code blocks for creating space; inserts itself; regenerates code and data references, including relocation information; and eventually rebuilds the executable. The virus *Zmist*, or *Zombie.Mistfall*, was the first one to exploit the technique, and scholars defined it “one of the most complex binary viruses ever written” [21].

Most anti-virus programs cope with camouflage resorting to dynamic and static heuristics analysis. In the former ones, a potential malicious application is executed on a virtualized sandbox system, then the modifications brought about by the program are checked relying on heuristic measures or specific triggers. On the contrary, the latter are based on the mere analysis of the code.

While dynamic heuristics may look promising, they require a significant amount of CPU-time and malware authors devised specific counter measures, called *arming*, against them. Fore instance, some malwares adopt stealth strategies to prevent an infected system to report their presence, and made necessary to run the anti-virus tools from a clean environment like by booting a live operating system from a portable device. Other anti-anti-virus techniques prevents emulation using undocumented CPU instructions, CPU intensive routines and other tricks, or detects whether a virtualization is in progress and do not execute on emulated systems at all [16]. As a result, effective static heuristics are an essential step in protecting from threats.

### 3 Automated Malware Creation

The idea of creating, or optimizing, malware in an automatic way is fascinating. Moreover, starting from the choice of the term “virus”, the connections between computer software trying to penetrate legitimate systems and biological infectious
agents that replicate inside the living cells of other organisms are more than evident [18].

Since early 90s, hundreds of malware creation toolkits were introduced, enabling individuals with little programming expertise to create their own customized malware. The Virus Creation Laboratory (VCL) is one of the earliest, it was released in 1992 by the NuKE hacker group, and featured a nice user interface and documentation. The same year, a felon under the name of Dark Avenger distributed a polymorphic toolkit: The Mutation Engine (MtE). MtE enabled neophyte programmers to automatically extend their malicious code into a highly polymorphic one. In the following years similar toolkit appeared, like Dark Angel’s Multiple Encryptor (DAME) written by the felon named Dark Angel, and the TridenT Polymorphic Engine (TPE) by Masud Khafir.

An early approach to create malware through an EA was proposed, with noble intents, in [12]. The approach was based on rearranging existing blocks of code, and it was unable to create a virus at the level of a single instruction. Despite its limitation, the very idea of autonomously evolved malware is still frightening [7]. This paper proposes to exploit an EA to optimize malware, tackling two of the main problems faced by anti-virus companies: 0-day detection and trojans’ infection. Both approaches exploit the open-source EA called $\mu$GP [15], which is able of both optimizing numeric values and handling assembly programs, adding, subtracting or changing single instructions.

Tackling 0-day detection, the EA is used to automatically create a new strains of attackers. The new malware is optimized in order not to be detected by existing using static heuristic analyses. The result might provide some insights about the weakness of adopted approaches. Conversely, tackling trojan generation, the EA is given the objective to find weak spots inside a target host software, where malware code can be seamlessly inserted without altering the target’s behavior. Such evolved malware could be used to assess anti-intrusion mechanisms of secure environments.

Determining that the generated software is an effective malware and that the full functionalities of an existing program have been preserved are two intractable problems. However, the use of EA enable to find potentially acceptable solutions in a limited amount of time and with limited computational resources.

3.1 Evolutionary Algorithms and $\mu$GP

EAs are stochastic search techniques that mimic the metaphor of natural biological evolution to solve optimization problems [6]. Initially conceived in the 1960s, the term EAs now embraces different paradigms like genetic algorithms, evolutionary strategies, evolutionary programming, and genetic programming. All EAs operate on a population of individuals; underlying, each individual encodes a possible solution for the given problem. The goodness of every solution is expressed by a numeric value called fitness, obtained through an evaluator able to estimate how well the solution performs when applied to the problem. An evolutionary step, called generation, always consists of two phases: a stochastic one where some of the best individuals are chosen at random to generate new solutions; and a deterministic one,
where solutions are ranked by their fitness and the worst ones are removed from the population. The process is then repeated until a user-defined stop condition is met.

Over the past decade, EAs have been successfully employed as optimization tools in many real-world applications [15]. EAs provide an effective methodology for tackling difficult problems, when no preconceived idea about the optimal solution is available. While it is not usually possible to mathematically guarantee that the optimal solution will be found in a finite amount of time, EAs have been demonstrated able to perform much better than traditional optimization techniques in several practical NP-hard problems.

µGP is a general-purpose EA toolkit developed by the CAD Group of Politecnico di Torino in 2002 and now available under GPL [17]. While the first version was developed specifically to generate assembly language [19], the latest release can be used to tackle a wide range of problems, including numerical optimization.

In the µGP toolkit, the candidate solutions of a problem are represented as graphs, while the problem itself is indirectly modeled as an external script that evaluates each candidate solution and supplies the tool with a measure of its goodness. Due to this loose coupling, µGP can be used on different problems with no modifications needed.

Configuration files in eXtensible Markup Language (XML) describe individuals’ structure and all necessary parameters such as population size, stop conditions, number of genetic operators activated at each step. Since in the specific problem individuals map sequences of keys, the related graphs are linear genomes.

### 3.2 Code Generation

In code generation, the EA is used to create a new malware, with the precise intention not to be detected by existing scanners. While it would be theoretically possible to make the EA discover the patterns of malware autonomously, it is far more advantageous to feed the initial population with examples of working software to obtain successful individuals in a far more reasonable amount of time. Thus, the code of several malicious applications can be converted into the EA’s internal representation of individuals. The evolution is then started, and the EA rearranges freely freely materials from the individuals provided in the initial population in order to create new malware. Figure 1 shows the structure of the proposed framework.

The final goal of the evolution is to create malware not detected by anti-virus applications. But during the evolutionary process it is possible to obtain non-valid programs, unable to compile or not being executed correctly. Moreover, programs that are compiled and executed successfully could lose the characteristics proper of malware, becoming harmless software applications. To drive the evolution towards the creation of malicious applications hard to detect, an individual is awarded a progressively higher fitness value if it satisfies a series of requisites.

Firstly, the assembly code must compile without errors. Then, the obtained executable must run without raising exceptions or fall into infinite loops. Since the system is able to insert any kind of instruction into the code, infinite loops are of course possible: this occurrence is taken into account by forcibly killing candidate programs that do not terminate in a time several orders of magnitude superior to
that of the original code. Provided that the program runs correctly, the results of its execution are then checked, to verify that its behavior is still compatible with that of malware. Finally, the candidate malware is analyzed by the scan of a group of anti-virus software. Since the experience is focused on testing the static heuristics only, the chosen anti-virus programs perform the scan without relying on their database. The final fitness value of each individual is proportional to the number of anti-virus application it is able to deceive.

Even more than in other applications of bio-inspired methodologies, the evaluation mechanism closely resembles a synthetic environment. Candidate malware programs represent individuals of a species hunted by anti-virus software. Like living animals, individuals in the EA may evolve features that help them escape the predation. Emerging positive characteristics of this kind are passed on to individuals in the successive generations, thus creating a natural defense strategies against anti-virus predators.

It is interesting to notice that deciding whether an individual still retains the characteristics of malware is a major issue of the evaluator: determining the behavior of a program is in fact a Turing-complete problem [3], thus not approachable in an automated way. As an approximation, a set of heuristics is here used to conclude if a specific program can still be called “malware” with a reasonable probability.

### 3.3 Code Integration

In code integration, the EA is used to determine the optimal position for hiding malicious code inside an existing executable. The goal is to perform the injection preserving both malware’s and host’s functionalities, and with no a-priori information about neither of them.

The EA is used to efficiently explore the search space of possible blocks to replace, probing the target’s code. The potential attacker is interested in finding vulnerable parts as large as possible, and it is important to notice that the
search space for blocks of variable size inside a program quickly explodes, even for a relatively small target. Furthermore, finding potential vulnerabilities in compiled software is a task that would involve an intelligent analysis of the target program’s behavior.

In order to empirically evaluate the vulnerability of a certain part of the code, the part is overwritten with other code and the program is run, trying to verify changes in the most common behavior. The injected code is quite simple, few assembly instructions for displaying characters on the screen, nevertheless it is sufficient to discriminate whether and to what extent it is executed.

Every individual in the EA represents a part of the program to be probed, and it is encoded as two integers: the first one (called offset) is the offset from the beginning of the compiled code, in bytes; the second (called size) is the size of the part, again in bytes (Figure 2).

The tool distinguishes between two types of areas of potential interest from an attacker’s point of view: Type I areas represent blocks of code that almost always skipped during a regular execution, like branches after a flow control instructions that are rarely activated; Type II areas are usually not processed by the normal flow, and often appear after the end of the main function of the program, like functions that are infrequently called.

The rationale is to use a Type I area to inject a vehicle, overwriting the branch with a call – as few as 22 bytes are sufficient to save all registry values, call the malware code, and restore the original values. Then, store the actual malware into one or more Type II areas. The tool does not need any hint, and it is is able to autonomously discriminate between the two types of area observing the behavior of the program after integration.

The EA generates individuals representing blocks, encoded as a vector of integers. The first integer is the starting point of the block, the second is the size. During evaluation, the original block indicated by the individual is overwritten with other code, and the behavior of the program is tested. The evaluator also tries to distinguish between areas of Type I and Type II, with different runs of the code.
4 Experimental Results

In order to attest their efficacy, the two proposed approaches are experimentally tested. The code generation method is assessed on a real-world malware against different commercial anti-virus software; while the evolutionary code integration is run on two Windows executables.

The two main parameters controlling evolution in µGP are µ and λ. The former is the size of the population: the number of active solutions in each step of the optimization. The latter is the offspring size: the number of new solutions generated in each generation. Almost all other parameters in µGP are self-adapted, that is, the tool tweak the value internally and no user intervention is required.

4.1 Code Generation

For the experimental evaluation, the code of the virus Timid is inserted into the initial population. Timid is a relatively simple malware, a file infector virus that does not become memory resident, firstly discovered in 1991 and rumored to be an escaped research virus [10]. Each time a file infected with Timid is executed, it copies its own code into other uninfected .COM files in the same directory; if there are no more .COM uninfected files, a system hang occurs. Timid is chosen for the experiment because of several desirable characteristics: despite its age, it still works on the Windows XP operative system; its code is available in the public domain; and its behavior is very predictable and controllable. Thus, checking if the modifications of Timid created by the evolutionary framework still behave as the original malware becomes a relatively straightforward process.

The A86 [8] assembly compiler is chosen for its efficiency in the compiling process. A DOS script kills the individual’s process if it takes more than 5 s to complete, thus preventing programs with infinite loops from blocking the evolution. A set of 5 .COM files, taken from those available in the directory DRIVE:\Windows\system32, are used as a test for the infection capability of an individual, their integrity being checked with a md5 cryptographic hash function [14]. An ensemble of 4 different freeware anti-virus applications is selected to verify the ability to escape detection. The chosen anti-virus software shares some desired features: possibility of excluding the database-driven malware detection, performing scans with the heuristic detection only; configurable heuristics, ranging from permissive to distrustful; and a relatively fast scan process. Excluding database-driven detection is a time-saving procedure, since the new malware created by the framework is obviously not included in any database, even if the original code is. The possibility of configuring the heuristics’ level of severeness is important to smoothen the fitness landscape, helping the EA to direct the evolution toward the aim. Finally, a fast scan process makes it possible to carry out experiments within a reasonable time limit.

Each experiment, run on a Notebook Intel Centrino running Windows XP Service Pack 2, takes 10 to 15 minutes to complete. Starting from the same initial population, containing only one individual modeling the original code of Timid, 100 runs of the framework are executed. For each experiment, µ = 10, λ = 8. 96 runs out of 100 reach the maximum possible fitness value, thus producing individuals
Table 1: Summary of the experiments for code injections. While SPLIT.EXE shows vulnerabilities even after a first run, several attempts are needed to find exploitable areas in TESTDISK.EXE which compile without errors, are executed correctly, behave like the original malware with regards to several monitored .COM files in the same directory and are not detected by all 4 freeware anti-virus applications. 4 experiments out of 100 are stopped because they produce individuals which delete or corrupt files needed to carry on the evolutionary process.

The successful experiments terminated in an average of 6 generations, with a standard deviation of 2.5. It is worth noticing that the best individual in generation 1 is already able to deceive the heuristics of two of the anti-virus applications; nevertheless, the fitness of the best individual progresses steadily during the generations.

### 4.2 Code Integration

The code integration approach is tested on two target executables. Each experiment, run on a Notebook Intel Centrino running Windows XP Service Pack 2, takes about 30 minutes to complete. For each experiment, $\mu = 100, \lambda = 30$. The first executable chosen to test the evolutionary code injection approach is SPLIT.EXE\(^2\), a small program (46.6 kB) able to split files of any kind into smaller parts or rebuild the original. The EA is set to generate the parameter offset in the range (0, 43000), and the parameter size in the interval (12, 1000) byte, factually covering all the original code. The first search was stopped after about 10 generations, for a total of 300 individuals evaluated: it reveals 1 zone of Type I and 32 zones of Type II, ranging from 65 to 1511 bytes, thus showing potentially vulnerable positions for an attack.

The second executable considered was TESTDISK.EXE\(^3\) (315.2 kB), an open-source utility designed to help recover deleted or inaccessible files, partitions, or even whole drives. The experimental setup and parameters are similar to the first experiment, with the offset interval set to (0, 10000) and (0, 2000), respectively. The evaluations were increased to 15,000, 2000, and 300, respectively, for each offset range.

\(^2\)See http://www.iacosoft.com/home/split.txt
\(^3\)See http://www.cgsecurity.org/wiki/TestDisk
source data recovery software available for different platforms. TESTDISK proves to be more resilient to attacks: a first run, with offset = (0, 43000) and size = (12, 1000) stopped after about 500 generations and 15,000 individuals evaluated, only uncovers only 3 small zones of Type II, ranging from 12 to 179 byte. Since the zones appear to be concentrated in the first part of the executable, subsequent attempts are made, progressively restricting the interested area with offset=(0, 10000) and then offset = (0, 2000). For these two settings, a zone of Type I if eventually detected. Thus, even if more resilient to attacks, even TESTDISK.EXE shows exploitable weak points.

For a complete summary of the experiments, see Table 1.

5 Conclusions

The paper shows how EAs can be exploited in anti-malware research. While the research is still at an early stage of development, its potential is apparent. An evolutionary toolkit was used first to devise a new malware, modifying and tweaking existing code to escape detection, then to find holes in an existing executable to hide malicious code in.

Future works will focus on the development of an automated procedure to model malware code in the internal representation of the EA; on the use of even more commercial anti-virus programs to improve the final applications produced by the framework; and on the development of general heuristics able to determine whether the programs produced by the framework still retain the malicious characteristics of the original software source, even when more distinct malware applications are present in the first population.

Finally, the two approaches will be merged in a single one, able to automatically create new code injectors.

Acknowledgments

The authors would like to thank Muddassar Farooq, Ralph Roth, and Waqar Ali for their availability and insightful discussions.

References


REFERENCES


