Challenging Anti-virus Through Evolutionary Malware Obfuscation

Marco Gaudesi[1], Andrea Marcelli[1(✉)], Ernesto Sanchez[1], Giovanni Squillero[1], and Alberto Tonda[2]

[1] DAUIN, Politecnico di Torino, Corso Duca Degli Abruzzi 24, 10129 Turin, Italy
{marco.gaudesi,andrea.marcelli,ernesto.sanchez,
giovanni.squillero}@polito.it
[2] INRA, UMR 782 GMPA, 1 Avenue Lucien Brétignières,
78850 Thiverval-Grignon, France
alberto.tonda@grignon.inra.fr

Abstract. The use of anti-virus software has become something of an act of faith. A recent study showed that more than 80 % of all personal computers have anti-virus software installed. However, the protection mechanisms in place are far less effective than users would expect. Malware analysis is a classical example of cat-and-mouse game: as new anti-virus techniques are developed, malware authors respond with new ones to thwart analysis. Every day, anti-virus companies analyze thousands of malware that has been collected through honeypots, hence they restrict the research to only already existing viruses. This article describes a novel method for malware obfuscation based an evolutionary opcode generator and a special ad-hoc packer. The results can be used by the security industry to test the ability of their system to react to malware mutations.

Keywords: Security · Malware · Packer · Computational-intelligence · Evolutionary algorithms

1 Introduction

Malicious software, *malware* for short, plays a part in most computer intrusion and security incidents. The name is used to indicate any software that causes *intentional harm* to a user, computer, or network [1] and may be found in different variants: *worm*, *rootkit*, *trojan*, however *viruses* are the most common.

The term *computer virus* was coined by Fred Cohen in 1983 [2] to indicate those programs that are able to replicate themselves and infect various system 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 [3], yet the first working computer viruses are much more recent.

Creeper, developed 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. A decade later, *Elk Cloner* was the first one known to

G. Squillero and P. Burelli (Eds.): EvoApplications 2016, Part II, LNCS 9598, pp. 149–162, 2016.
DOI: 10.1007/978-3-319-31153-1_11

infect different computers spreading by floppy disks. On the other hand, *Brain*, written by two Pakistani brothers and released in January 1986, is widely considered the first real malware [4]. Since then, malware grows in complexity and dangerousness. As of 2015, malware analysis and detection is still an important field in computer research and computer virus still represents a heavy risk in computer security. Malware analysis is like a cat-and-mouse game. As new analysis techniques are developed, malware authors respond with new ones to thwart analysis.

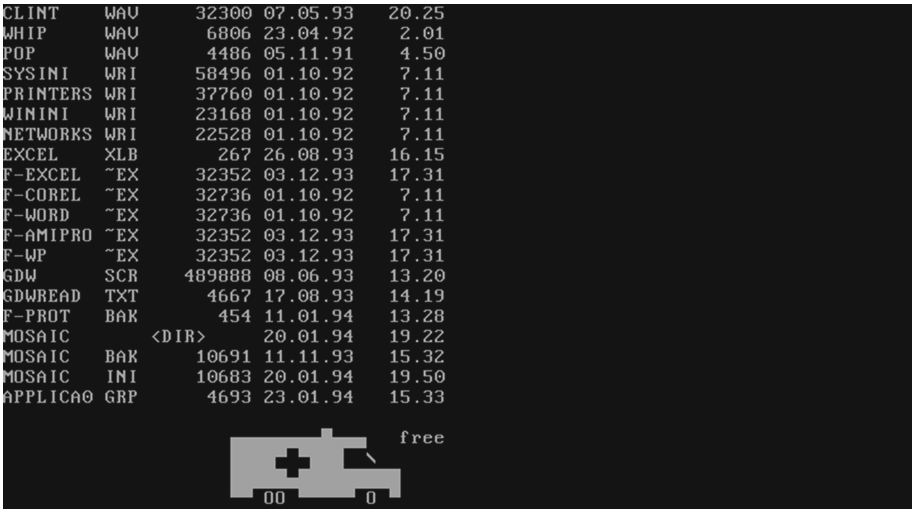

Fig. 1. Screen shot of malware *Ambulance.com* after infection.

It is interesting to note that from year 2003 the focus has changed from "writing for fun" to "writing for profit". In the early days, viruses were written by hobbyists mainly for joke or for challenge. They usually played with the user or print funny messages or graphics on the screen. For instance, Fig. 1 displays a screen shot of an infection by *Ambulance.com*, a virus written for DOS. When the user got infected, an ambulance crossed the screen and a siren started to sound. Today, when someone is infected by malware does not even know to be infected. The victim does not see anymore funny images on the screen, nor the cd rom trail rom will open and close all the time. The malware runs silently in the background, without crashing the system. In effect, viruses are well tested and debugged in order to not slowing down the system [5]. Indeed, the hiding capability of a malware is a crucial aspect. For this reason the vast majority employs some kind of *obfuscation*, ranging from simple XOR encryption, to more sophisticated anti-analysis tricks. However the most common is *packing* [6,7].

The research presented in this paper aims at developing a new evolutionary obfuscation mechanism. The results could be used by the security industry to evaluate the efficiency and effectiveness of their analysis methodologies, as well as to test the ability of their system to react to new malware mutations. In order to mimic real malware mutations, the research focused on the development of an *evolutionary opcode generator*, which is the foundation of a packer. A mechanism that is able to automatically generate hard to detect programs, can help the security research into developing a new proper countermeasure.

The analysis depicted in this paper represents the third chapter of a work focused on evaluating and testing anti-virus products: in a first paper [8] it was experimented the application of the μGP toolkit to evolve the infection mechanisms of a simple DOS virus. Then, a second work tested commercial products responsiveness to a runtime obfuscated shellcode [9].

The usage of evolutionary computation techniques in the field of malware mutation is not new. Previous works include [10], where an evolutionary framework that exploits genetic algorithms was proposed to evolve a well-known virus family, called Bagle. Furthermore, in [11], the authors proposed an artificial arms race between an automated white-hat attacker and various anomaly detectors for the purpose of identifying intrusion detection system weaknesses. Eventually, in [12], it is provided a theoretical proof behind malware implementation that closely models Darwinian evolution. The malware autonomously incorporates behaviours by including numerous discoverable APIs. The new behaviour profiles would is constantly screened by security software in the same way natural selection acts on biological organisms.

The rest of the paper is organized as follows: Sect. 2 introduces viruses and anti-viruses; Sect. 3 illustrates the goal of the research; Sect. 4 reports experimental evaluation; finally, Sect. 5 concludes the paper, outlining future works.

2 Virus and Anti-virus

Since the first days of malware, the security industry developed anti-virus programs. Their efficiency is evaluated on the base of the *detection ratio* (that should be maximum) and *false-positive ratio* (that should be minimum). However, there is a common misconception about anti-virus scanners: people often get the impression that faster is better; hence all the techniques need to target speed scanning. Commonly used techniques in commercial anti-virus applications include Signature Scanning, Geometric Detection, Disassembler combined with a State Machine and Emulators. *Signature scanning* was the first to be developed and today it is the most common technique of viruses detection. It looks for a specific sequence of bytes in the inspected executables. Instead, *geometric detection* analyses the file structure and identifies a virus if an alteration has occurred. While a *disassembler* is used to separate the byte stream into individual instructions, when it is combined with a state machine, it can be used

to record the order in which "interesting instruction are encountered". However, the most advanced and complex solution is the *emulator* which simulates the behaviour of a CPU. The code that runs in an emulator is executed within a controlled environment from which it cannot escape. Moreover, it can be examined periodically or when particular instructions are executed [13].

In the anti-virus field it is often used the term *heuristic* to indicate all those technologies that are employed to detect malware, without requiring a specific detection routine or signature for each known malware. Moreover the usage of heuristic is has the potential to find out new version of malware, by exploiting similarities with known samples. Thus employing heuristics, is a systems that can only reduce the problem against masses of viruses and of course, the perfect detection is more difficult to be done by using pure heuristics without paying some attention to virus-specific details. Heuristic may target both static and dynamic analysis. A proposed system to heuristically detect zero-day malware based on static analysis is PE-Miner, which uses around two hundred features of the Portable Executable file structure [14]. Plenty of research has been done using n-gram distribution or opcode frequency of byte sequences to heuristically detect zero-day malware [15]. However, since these methods depend only on the byte sequence, they can be easily bypassed by obfuscation, code transformation and packing [16]. In the dynamic analysis the program operations are tracked while a heuristic mechanism tries to recognize behavioural patterns typical of malware. In any case it is unknown which is the heuristic technique implemented in anti-virus commercial products, since it represents the value of the software itself.

The usage of anti-virus software has become something of an act of faith [17]. People seem to feel more safe not with a more secure operating system, or with the latest patch, but with some anti-virus software installed in their systems. A recent study [18] showed that over 80 per cent of all personal computers have anti-virus software installed on their computers. Quite clearly, this is a must-have for most users. However, this research outcome showed that viruses are a serious threat and the protection mechanisms in place are less effective than we would expect.

By analysing the evolution of the malware in the last thirty years, it is clear that there has been a growth in the complexity of the hiding mechanisms. Anti-virus software industry historically classifies obfuscating malware in *encrypted*, *olilgomorphic*, *polymorphic* and *metamorphic* viruses. The first ones are the simplest and they are characterised by a constant decryptor that is followed by the encrypted virus body. On the other hand, metamorphics are the most advanced: they are capable of completely disassemble, regenerate the code and re-compile it at run time [19].

The analogies between computer malware and biological viruses are more than obvious. The very idea of an artificial ecosystem where malicious software can evolve and autonomously find new, more effective ways of attacking legitimate programs and damaging sensitive information is both terrifying and fascinating [8]. In the biological world, every time a new species is discovered, it's encountered an adaptation that is not expect. The same is true for malware:

virus analyst needs to train themselves to think like exploratory biologists and be prepared for things they have never seen.

The research, supported by security experts, identifies the *evolutionary malware* as the next possible malware threat. The new category is suggested for those viruses that will exploit the full power of evolutionary algorithms (EA) and computational-intelligence techniques in general. This kind of virus will be able to learn from the surrounding environment, to be trained and to create false positives.

3 Proposed Evolutionary Malware Obfuscation

The section describes the new malware obfuscation mechanism. The developed *ad-hoc packer* is responsible of creating a new working variant of the malicious executable file. The *opcode generator* uses an evolutionary algorithm to create two assembly routines: one is used to *obfuscate* and the other to *de-obfuscate* arbitrary data. Such hiding mechanism is then adopted by the packer to encode the file content, by obfuscating the malicious code and data embedded in the file.

3.1 Evolutionary Opcode Generator

An *Opcode* is the binary representation of an assembly instruction. The opcode generator creates both an encoding and a decoding function starting from randomly-generated, variable-length sequence of IA-32 assembly instructions. Those are directly handled as binary opcodes, so there is no need of a compilation and linking phase.

The generation process exploits a simple evolutionary algorithm with a strategy similar to $(1, \lambda)$, that is, not using any recombination operators. The encoding and the decoding functions are created at the same time in a process that requires to find either reversible assembly instructions (or small blocks of code) and their complementary one. Since even few bytes may represent a signature, it is also necessary to partially shuffle the instructions, although this has the drawback of potentially disrupting the encoding/decoding routines.

In order to assess candidate fitness values both the *reversibility* of the generated routines and the *Jaccard Similarity* is evaluated. In favour of efficiently evaluate a candidate obfuscating routine, the encoding and the decoding routines are applied subsequently to randomly generated sequence of bytes: if the final result is different from the original sequence, the candidate is simply discarded. Then, the *encoder* is used to obfuscate the malicious code and eventually the *Jaccard Similarity*, a statistic coefficient used for comparing the similarity and diversity of sample sets, is appraised. The Jaccard Index has been chosen for keeping the evolution mechanism lightweight while gaining effective results.

Aiming to achieve invisibility through diversity, the process is iterated until a timeout expires, or until the Jaccard coefficient is lower than an experimentally-defined threshold.

The entire process is shown in the following pseudo-code and each phase will be analysed in details in the following paragraphs.

```
Initialise population

while (!stopping condition) {
        create new lambda individuals by choosing one of the
            following methods:
                        1. append instructions to both
                            encoder and decoder
                        2. shuffle decoder instructions
        evaluate new individuals
        keep the best
}

evaluation:
  test reversibility
  if (success)
       evaluate Jaccard
```

IA-32 Instruction Set. The *assembly* language for Intel x86 is a collection of hundreds of instructions. Each of them is translated in several binary representation, called opcodes, according to its *usage* in combination with register or memory and its *version*: 8, 16 or 32 bit.

According to Intel IA-32 manual, instruction can be classified in *Data Transfer*, *Binary Arithmetic*, *Decimal Arithmetic*, *Logic*, *Shift and Rotate*, *Bit and Byte*, *Control Transfer*, *String*, *Flag Control*, *Segment Register*, *Miscellaneous*, *MMX Technology*, *Floating Point* and *System* instructions.

Only a small subset of the IA-32 instruction set have been employed in the realisation of the opcode generator, but more can be added in the future: in the prototype described in this paper no *Floating Point*, nor *MMX Technology* instructions are used. Also *Segment Register*, *Miscellaneous*, *Bit and Byte*, *String* and *System* instructions have been discarded, because their application was not straightforward.

The other categories represent the most used opcodes, hence the most interesting. Some of them have a direct reversible instruction, which is very useful when it comes to create two equal, but reverse sequence of operations. Others, like *mov* or *push and pop*, insert some approximation in the reversibility of the solution. Those who remain are strongly related to *carry management* or to the *evaluation of a condition*.

Among all of them, the most common *twenty* have been chosen, including XOR, INC, DEC, NEG, ROL and ROR. Instructions are stored in a small database of opcode ranges, where each one is associated with a data structure with all the necessary information for the code generation: the opcode length and range, a pointer to the opposite opcode, the need of a parameter and eventually the length and range of this one.

Instruction Generation. When the opcode generator is called, the evolutionary algorithm starts. An individual is made of a pair of encoding and decoding routines built from randomly chosen sequence of instructions. Since each opcode has a pointer to its directly opposite instruction in the database, both the functions are constructed in parallel. Usually instructions are characterised by a base number, that identifies the specific version of the assembly instruction in use and an offset, that identifies which register or memory address is being adopted. If a parameter is required too, the algorithm generates one in the proper range.

In order to assess candidate fitness values both the *reversibility* of the generated routines and the *Jaccard Similarity* of the obfuscated code is evaluated. In order to increase the strength of the obfuscation, the mutation mechanism partially shuffle the instruction order, although this has the drawback of potentially disrupting the encoding/decoding routines. Hence, to test the reversibility of the two just created assembly routines, a randomly sequence of byte is generated and copied in the processor registers. If the final register values are different from the original ones, the candidate is simply discarded and the generation restarts from scratch.

The evolutionary process is iterated until a timeout expires, or until the Jaccard coefficient is lower than an experimentally-defined threshold.

Evaluating Similarity. The *Jaccard index*, also known as the *Jaccard similarity coefficient*, is evaluated to assess candidate fitness values. It measures the similarity between finite sample sets and it is defined as the size of the intersection divided by the size of the union of the sample sets:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Starting from two variable length byte array of assembly opcodes, the program cyclically calculates the Jaccard Coefficient for each pair of bytes.

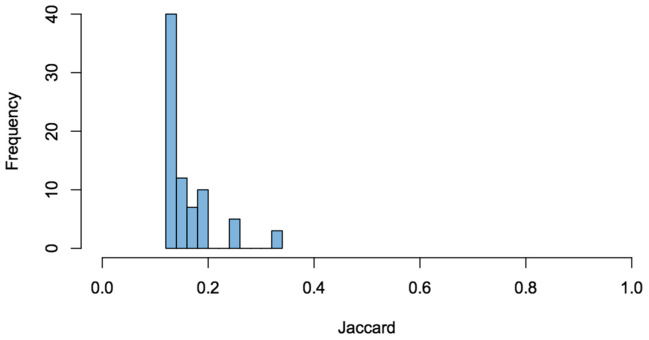

Fig. 2. Histogram of the Jaccard coefficient distribution of a Malware Sample.

The implementation is straightforward: the union is implemented using the binary operation "AND" and the intersection using "OR". Then, the sum of the "1" in each resulting byte is calculated and the result is saved in a temporary array. In the end each coefficient is normalised on a scale of ten values and it could be graphically displayed on an histogram. The algorithm has been tested with different block size and experimental results showed best behaviour considering a block of one byte. Figure 2 shows the *Histogram* result from the calculation between two malware variants that are dissimilar among them. The entire process is repeated untile the Jaccard coefficient is lower than an experimentally defined threshold or when the timeout expires.

3.2 Development of an Ad-Hoc Packer

In the proposed approach, the developed *packer* is responsible of creating new hiding mechanism strong enough to ensure the survival of future generations of malware. The suggested hiding mechanism is based on evolutionary computation and the *Evolutionary Opcode Generator* is embedded directly in the packer. The goal is to develop a packer able to thwart analysis, creating each time a new packing routine exploiting Evolutionary Algorithms. As the decoding routine is embedded in the executable file, once the new malware is run, it will restore each part of the program in memory ready for execution. Differently from a previously published research [9], the packer tackles the entire executable and not only portions of code.

In detail, the developed packer consists of a stand-alone program, the *packer*, that is responsible of creating a new variant of the input program. The research targets the *Portable Executable* file, the format of Windows executable [20]. The file format is logically organised into several sections and the Section Header keeps track of all of them. The packer encode the code and data sections with a custom obfuscating routine and the *unpacker* is directly injected in a new crafted section.

Packing. In order to inject the packing routine, the file size is increased. For this purpose, Windows offers two useful API: *CreateFileMapping* and *MapViewOfFile* which allow to map a file in memory and to work on that by simply using memory addresses. Then the *packing* function is called.

Firstly the *NumberOfSection* is increased by one. Then, the *Characteristic* of each section is set to *read* and *write*. Later, the address of the *RelocationTable* in the *DataDirectory* is set to zero to avoid the Windows Loader to perform "relocation fixups". Finally, a cycle loops on each section encoding it using a previously created obfuscating routine.

A new ".unpack" section is appended at the end of the file, where the unpacker code, directly written in assembly language, is copied and all the necessary informations are written in the *SectionTable*. The *AddressOfEntryPoint* is updated to point to the corresponding decoding function in the new section and

the *SizeOfImage* is enlarged to include the new portion of file. Finally the Relative Virtual Address (RVA) of the "Relocation Table" and the *Original Entry Point* are updated in the unpacker code.

Unpacking. The code of the unpacker is the first to be executed when the program is run. It is mainly responsible of restoring the original sections in memory, performing the relocation fixups and loading the required libraries.

ImageBaseAddress: The *Windows Loader* loads the Portable Executable file starting from an arbitrary *ImageBaseAddress*[1]. In the 32bit version of Windows, the FS register points to the *Thread Environment Block*[2], a small data structure for each thread, which contains information about exception handling, thread-local variables and other per-thread state. Among the various fields, one is a pointer to the *Process Environment Block*[3], that contains per process information. Those data structure are not well documented by Microsoft, but over the Internet, it is possible to find a lot of information from independent researchers. An interesting blog post[4] illustrates that the pointer to the *ImageBaseAddress* is constant among all the Windows version, hence the following unpacker is multi version compatible.

PE Parsing and Decoding: The Portable Executable header is parsed, looking for the address of the *SectionTable*, where it is possible to retrieve the starting addresses of the sections to be decoded. Then a loop iterates over each of them and the unpacker restores the original data in memory.

Relocation Fixups: One of the main drawbacks of encoding the code section is that it prevents the Windows Loader to perform the correct relocation adjustment. In summary, the complexity of the unpacker is to mimic the operations typically done by the Windows Loader. The process of relocation is organized in three phases. Firstly it is necessary to calculate the *delta_reloc*: the difference between where the executable is actually loaded, from the one the compiler assumed it would have been loaded. Secondly, it obtains the address of the RelocationTable and thirdly, a loop iterates over each entry of the *RelocationTable* to perform all the necessary fixups. At the end, the program execution is reverted to the *OriginalEntryPoint* and the original program is run.

4 Experimental Evaluation

Experiments have been performed on Windows 7, 32 bit version, with an Intel IA-32 architecture. The choice is dictated by the huge availability of anti-virus

[1] Note: test have been executed on Windows 7, by default ASLR is enabled.
[2] https://msdn.microsoft.com/en-us/library/windows/desktop/ms686708(v=vs.85).aspx.
[3] https://msdn.microsoft.com/en-us/library/windows/desktop/aa813706(v=vs.85).aspx.
[4] http://blog.rewolf.pl/blog/.

software for the platform. As a testbed, three well-known online malware scanner have been used: Metascan Online[5], VirusTotal[6] and Jotty[7]. On the whole they make use of tens of different AV products, allowing a direct and simple comparison of the results. Moreover, in order to confirm the outcome precision, several of the most effective AV software have been installed on different hosts.

In order to test the effectiveness of the *packer*, some virus samples have been chosen among a collection of malware from *VX Heaven* (http://vxheaven.org). All of them are characterised by a very high detection rate. Among the ten chosen samples, the packer successfully created a new variant for eight of them. The other two, *Win32.Apparition* and *Win32.Chiton*, make use of anti-dumping and anti-debugging techniques that suspended the packing process. Each generation required an average of 60 individuals to find a working obfuscating routine, making the packing process almost instantaneous. A prudent resource management must be always taken into consideration when dealing with anti-virus software that carefully monitor processes CPU usage.

Table 1 summarises the test results. The first column *Original* shows the detection percentages of the original malware samples, while column *Packed* illustrates the detection percentages of the packed version of each malware sample. Each column is further subdivided into three ones: *Exact* is associated to the exact detection, *Heuristic* is related to the percentage of heuristic detection. The third one, *Total*, is the sum of the previous two values. Finally, column *Worsening* highlights the detection worsening between the original and the packed version of the malware.

Table 1. Comparison of the detection percentages of the Original and Packed version of the Malware samples.

	Original			Packed			Worsening
	Total	Exact	Heuristic	Total	Exact	Heuristic	
Win32.Bee	79 %	74 %	5 %	26 %	18 %	8 %	54 %
Win32.Benny	74 %	64 %	10 %	31 %	23 %	8 %	44 %
Win32.Blackcat	72 %	64 %	8 %	26 %	15 %	10 %	46 %
Win32.Bolzano	64 %	62 %	3 %	28 %	21 %	7 %	36 %
Win32.Crypto	72 %	64 %	8 %	28 %	21 %	7 %	44 %
Win32.Driller	69 %	62 %	8 %	15 %	10 %	5 %	54 %
Win32.Eva	79 %	74 %	5 %	26 %	18 %	8 %	54 %
Win32.Invictus	79 %	67 %	13 %	15 %	0 %	15 %	64 %
Average	74 %	66 %	7 %	24 %	16 %	9 %	49 %

[5] https://www.metascan-online.com/.
[6] https://www.virustotal.com.
[7] https://virusscan.jotti.org.

The unencoded version of the malware is characterised by a high detection rate. It ranges from 64 % to 79 % and in most of the cases it is related to *exact* detection. This result is mostly due to virus signature scanning, which is a very effective technique when a malware is not obfuscated. Detecting a packed version of a malware, is much more difficult, indeed the detection rate drops down to an average 24 %, furthermore heuristic is responsible of about half of the uncovering. On average, as illustrated in Table 1, the packer caused a *detection worsening* of the 50 %.

A total of *39* anti-virus have been used during the tests. In the following dissertation they are identified by the acronym *AV1..39*. Figure 3 graphically illustrates the results. *Five* products were not able to detect even a malware sample, whereas *nineteen* detected all the eight original viruses. Among them, *eleven* achieved the perfect detection of the original malware version. As previously stated, heuristic detection plays a small role when the malware is in the original form. Only *AV35* relies more on heuristic than on exact matching.

The most interesting outcomes come from the packed version of the malware. In *fifteen* cases, as shown in Fig. 4, the detection rate drop down to an incredible zero and in other *seven* cases the match worsening is greater than 70 %. Only one, *AV5* achieves the 100 % of malware uncovering, but it is all related to heuristic, then it is less trustworthy than exact detection. *AV9* increased its detection ratio of over 130 % when it analysed the obfuscated variant. However, it is safe to be unconfident of this success: it is likely that the anti-virus detected solely the presence of a packer and not the existence of a real threat. On the other hand, *AV6, AV15 and AV26* reached a great achievement, by worsening their result of only 13 % and other *four* of less than 40 %.

As previously mentioned, when it comes to heuristic detection it is hard to judge the success and it is important to evaluate the credibility of the results. For this purpose a further experiment was performed: a simple "Hello World" program was packed using the same *packer* of the previous test. Of course the original program, that quietly prints to the screen the string *Hello World* was not detected as a threat, when examined and only *four* anti-virus identified the

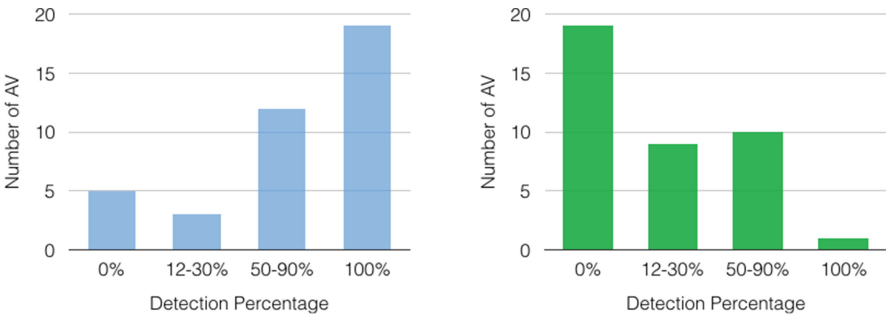

Fig. 3. Comparison of the detection percentage of the original malware samples (on the left) and the packed ones (on the right).

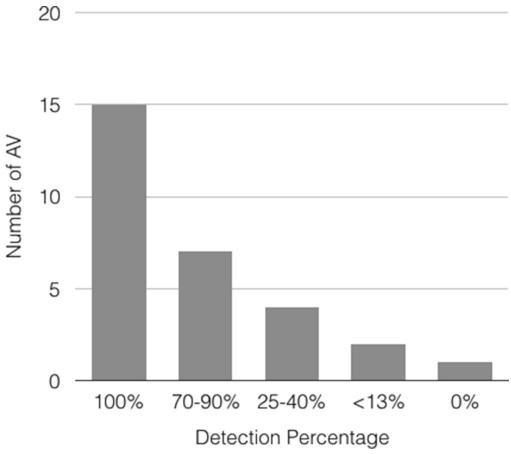

Fig. 4. Detection worsening caused by the packed malware samples.

encoded version as a threat by mean of heuristic detection. Results are quite interesting, as they can be used to further analyse the outcome of the previous test. Simply packing an executable makes *AV9* revealing a threat: this explains the incredible improvement (over 130 %) of the previous experiment. Also *AV6* suffers from false positive detection and its good achievement in detecting packed malware looses value. Finally, despite *AV38* and *AV39* experienced false positive, they were not able to detect none of the packed malware.

5 Conclusion

The basic idea of the research is to use *Evolutionary Computations* techniques to evolve computer viruses to foresee threats of the future. Several proof-of-concept samples have been developed and then tested against most common commercial anti-virus products. Although the relatively simple evolutionary approach, experimental results proved that a packing program that is able to evolve, creating a brand new encoding routine in each infection, may still represent a challenge for the security community. By the way, a mechanism that is able to automatically generate hard to detect programs, can help the security research into developing a new proper countermeasure.

Anti-virus software is one of the most complex application. It has to deal with hundreds of file types and formats: executables, documents, compressed archives and media files. However, it has been showed that the user usually overconfidence anti-virus in their immunity capabilities against all files. Results demonstrated that todays most effective solution to detect known malware is *signature-based* scanning. On average 74 % of the original malware was exactly spotted, in effect by only precisely identifying a threat it is possible to correctly repair the system. However, exact identification is a problem, even in human

analysis. How long does it take to be sure if something is really a known variant or a new one? It is believed that when dealing with an *evolutionary malware* it is not feasible to have a custom routine detection for each single sample: the complex infection mechanism coupled with the powerful evolutionary engine make it very difficult to reach full accuracy using only empirical evaluation methods and in depth analysis of the virus code is essential. In the authors' opinion, the anti-virus industry must focus on *heuristic detection*, by increasing the strictness of results and especially targeting the decrease of false positive. Anyway, the need to understand evolutionary code in a quicker way must be the subject of further research.

Malware analysis is like a cat-and-mouse game. As new anti-virus techniques are developed, malware authors respond with new one to thwart analysis. Anyway, as long as old tactics continue to remain effective, we will continue to see them in use: malware needs to propagate, it needs to communicate and it needs to achieve the goals for which it was designed. These are constants that will be seen well into the future.

5.1 Future Developments

While this research is far from being concluded, it has already gained a great deal of insight into the protection provided against viruses. Improvements may concern the developing a more advanced opcode generator and may target analysis of malicious networking interaction.

Future work will focus on investigate *machine learning* techniques for malicious pattern behaviour recognition.

Acknowledgments. A special thank to Peter Ferrie, principal anti-virus researcher at Microsoft, for answering the questions as well as his comments and feedback on latest malware obfuscation technologies.

References

1. Michael, S., Andrew, H.: Practical Malware Analysis - The Hands-On Guide to Dissecting Malicious Software. No Starch Press, San Francisco (2012)
2. Cohen, F.: Computer viruses: theory and experiments. Comput. Secur. **6**(1), 22–35 (1987)
3. Von Neumann, J., Burks, A.W., et al.: Theory of self-reproducing automata. IEEE Trans. Neural Netw. **5**(1), 3–14 (1966)
4. Chen, T.M., Robert, J.-M.: The evolution of viruses, worms. In: Statistical Methods in Computer Security, vol. 1 (2004)
5. Szor, P.: The art of computer virus research and defense. Pearson Education, Indianapolis (2005)
6. Yason, M.V.: The art of unpacking, Chicago (2007). Retrieved 12 February 2008
7. Guo, F., Ferrie, P., Chiueh, T.: A study of the packer problem and its solutions. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 98–115. Springer, Heidelberg (2008)

8. Cani, A., Gaudesi, M., Sanchez, E., Squillero, G., Tonda, A.: Towards automated malware creation: code generation and code integration. In Proceedings of the 29th Annual ACM Symposium on Applied Computing pp. 157–160. ACM, March 2014
9. Gaudesi, M., Marcelli, A., Sanchez, E., Squillero, G., Tonda, A.: Malware obfuscation through evolutionary packers. In: Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference, pp. 757–758. ACM, July 2015
10. Noreen, S., Murtaza, S., Shafiq, M.Z., Farooq, M.: Evolvable malware. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, pp. 1569–1576. ACM, July 2009
11. Kayack, H.G., Zincir-Heywood, A.N., Heywood, M.I.: Can a good offense be a good defense? Vulnerability testing of anomaly detectors through an artificial arms race. Appl. Soft Comput. **11**(7), 4366–4383 (2011)
12. Iliopoulos, D., Adami, C., Szor, P.: malware evolution and the consequences for computer security. arXiv preprint arxiv:1111.2503.Chicago
13. Szr, P., Ferrie, P.: Hunting for metamorphic. In: Virus Bulletin Conference, September 2001
14. Nachenberg, C.: Computer virus-coevolution. Commun. ACM **50**(1), 46–51 (1997)
15. Perriot, F., Ferrie, P., Szor, P.: Striking similarities. Virus Bull., 4–6 (2002)
16. Desai, P.: Towards an undetectable computer virus (Doctoral dissertation, San Jose State University), Chicago (2008)
17. Xue, F.: Attacking antivirus. In: Black Hat Europe Conference (2008)
18. Microsoft Security Intelligence Report, vol. 18, December 2014
19. Ferrie, P., Szor, P.: Zmist opportunities. Virus Bull. **3**(2001), 6–7 (2001)
20. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. https://msdn.microsoft.com/en-us/library/ms809762.aspx