J Comput Virol (2010) 6:277–287
DOI 10.1007/s11416-009-0136-2

ORIGINAL PAPER

From the design of a generic metamorphic engine to a black-box classification of antivirus detection techniques

Jean-Marie Borello · Éric Filiol · Ludovic Mé

Received: 22 December 2008 / Accepted: 7 September 2009 / Published online: 25 September 2009

Abstract In this paper, we propose an original black-box approach concerning antivirus products evaluation. Contrary to classical tests focusing on detection rates concerning a specific malware sample, we use a generic metamorphic engine to observe the detection products behaviors. We believe that this point of view presents a double interest: First, it offers an original way of evaluating current antivirus products focusing on the observed detection technique. More precisely, the use of metamorphic malware guarantees the difficulty of static signature based detection techniques to focus only on heuristic and behavioral detection approaches. Second, by pointing out current detection capabilities, we practically evaluate the danger that complex metamorphic malware could represent. To achieve this goal, we start with the description of a generic metamorphic engine acting in two steps: obfuscation and modeling. Then, we apply this engine to a real mass-mailing worm and propose the resulting metamorphic malware samples to current antivirus products. The observed results lead to a classification of detection techniques in two main categories: the first one, relying on static detection techniques, presents low detection rates obtained by heuristic analysis. The second one, composed of behavioral detection programs, mainly focuses on elementary suspicious actions. In all cases, no product was able to detect a global malware behavior. Consequently, we consider that metamorphic malware detection still represents a real challenge for antivirus products. Through this study, we hope to help defenders understand and defend against the threat represented by this class of malware.

J.-M. Borello (✉)
CELAR, BP 7419, 35 174 Bruz Cedex, France
e-mail: jean-marie.borello@dga.defense.gouv.fr;
borello.jeanmarie@gmail.com

É. Filiol
ESIEA, Operational Virology and Cryptology Lab.,
53 000 Laval, France

L. Mé
SUPELEC, SSIR (EA 4039), Rennes, France

1 Introduction

Malware is a generic term used to describe all kinds of software presenting malicious behavior. In terms of security of computer users, malicious software is considered as major threat. Many detection programs are based on form detection relying on byte signature to identify a specific malware. To circumvent these detection tools, attackers have developed specific counter-measures, giving birth to more and more advanced code mutation techniques. Among all these techniques such as encryption and polymorphism, metamorphism is certainly the most advanced one. Metamorphism consists in canceling as much as possible any fixed element that would represent a potential detection pattern according to byte signature matching. Here, we consider metamorphism as a special class of self-replicating programs.

From a theoretical point of view, few results exist concerning the detection complexity of code mutation techniques, even if these notions have already been evoked in F. Cohen's seminal works [10]. Recently, D. Spinellis has proved [23] that the detection of bounded-length polymorphic viruses is an NP-complete problem. Then, E. Filiol has formalized metamorphism by the means of formal grammars and languages [16] to extract three classes of detection complexity according to the corresponding grammar class identified by N. Chomsky [7,8]: polynomial complexity for class 3 grammars, NP-complexity for class 1 and 2 grammars, and undecidability for class 0 grammars.

From a practical point of view, very few metamorphic malware are known to exist thanks to the difficulty of writing such complex programs. The most advanced metamorphic virus known is MetaPHOR [13], for which 14 000 lines of assembly instructions (90% of the code) are dedicated to the metamorphic engine. To change its form, this virus uses simple instructions rewriting rules which allows its detection [16].

In this paper, we focus on metamorphic malware detection capabilities. More precisely, it was suggested in [16] that a metamorphic malware presents few limitations from an execution context point of view, whereas any antivirus tool is bounded by severe time constraints. To take advantage of this time constraint, a new class of obfuscator denoted τ-obfuscator was introduced in [3] to delay code analysis for a predefined time τ. Our work aims at evaluating current antivirus products confronted by possible future threats that metamorphic malware could represent by taking advantage of the time-answer constraint from a detection point of view. This approach allows an original black-box evaluation of current malware detectors through their response to new mutated forms of known malware.

The contributions of this paper are the following:

- We propose a design of a metamorphic engine corresponding to a future possible threat that antivirus tools must deal with.
- We present an original way of classifying antivirus detection techniques based on their observable behavior towards mutated form of known malware.
- We give a precise example of our approach through the application of our metamorphic engine on the well-known email-worm MyDoom [15].

This paper is organized as follows. In Sect. 2 we introduce metamorphic malware detection and its link with code obfuscation. In Sect. 3 we present the description of our metamorphic engine. In Sect. 4 we classify detection tools according to their response to metamorphic malware obtained by the application of the previous metamorphic engine on a real email-worm.

2 Metamorphism and obfuscation

Metamorphic malware and code obfuscation techniques are narrow linked subjects. Indeed, as mentioned in [9], a metamorphic code can be viewed as an obfuscated program. So, detecting such a program leads to the ability to de-obfuscate it. Before describing our metamorphic engine approach, we briefly introduce these two fundamental notions of obfuscation and metamorphism.

Barak et al. informally defined an obfuscator $\mathcal{O}$ as a probabilistic compiler taking in input a program P, and producing a new program $\mathcal{O}(P)$ with the same functionality as P but being unintelligible [2]. Starting with this informal definition, they proposed a formal one based on oracle access to program. Then, they proved that no obfuscator exists according to this definition. Recently, another formalization of obfuscation, based on the notion of oracle programs led to the same impossibility result [3]. Despite these theoretical results, practical obfuscation has been intensively investigated to protect intellectual property and especially concerning high level languages such as JAVA and the .NET framework [11,12]. Indeed, with such languages, the resulting binary contains all the information allowing us to easily retrieve the original program structure such as names, structures, data types, etc.

Concerning malware, obfuscation schemes were used to change the syntactic structure of the code to escape simple form detection techniques such as pattern matching. Metamorphic malware traditionally used basic obfuscation transformations modifying either data flow (rewriting rules, registers exchange) or control flow (branch insertions) to avoid pattern detection [5]. The choice of such basic obfuscation transformations was clearly evoked in [20] as follows: *"... a metamorphic virus must be able to disassemble and reverse itself. Thus a metamorphic virus cannot utilize [...] techniques that make it harder or impossible for its code to be disassembled or reverse engineered by itself."* In agreement with this point of view, many static detection approaches based on de-obfuscation techniques (such as data flow analysis [1] and slicing [25]) were developed [6,22,28]. However, more complex obfuscation schemes based on control flow modifications such as [5], could thwart these static detection techniques. Being aware of static detection limitations, an increasing number of antivirus products consider behavioral detection, which can be divided in two classes [17]. The first one is represented by dynamic detectors relying on sequences of observable events such as system call traces. The second one is composed of static verifiers relying on instruction meta-structures (graphs, temporal logic formula). In [18], the coverage of behavioral detection engines was assessed with the introduction of functional polymorphic engines. Briefly, a functional polymorphic engine was defined as a malware embedding a non deterministic compiler to dynamically produce functional variants from a high level malware description. In this paper, we focus on the temporal constraint aspect by investigating the new threat that τ-obfuscation-based metamorphic malware could represent on antivirus products.

3 Metamorphic engine description

From a high level point of view, a metamorphic engine offers a self-replicating property which has to produce a syntactically different but semantically equivalent mutated form. A generic description of metamorphic binary-transformation is given in [27]. Here, we present our metamorphic engine self-replication process, which acts in two steps:

1. In the first step, known as the obfuscation step, the engine changes its form to escape detection algorithms. The main purpose of this step is to avoid static detection approaches such as [4,6,9]
2. In the second step, the already obfuscated engine reverses its own obfuscation transformations to come back to its original form. This step, known as the modeling step, allows the engine to re-obfuscate itself. It is worth mentioning here that the reverse engine in charge of the engine modeling is itself obfuscated otherwise it could be easily detected by pattern matching.

This section presents the design of our metamorphic engine. More precisely, in Sect. 3.1, we focus on the obfuscation step. In Sect. 3.2, we describe the engine information needed to ensure its modeling. In Sect. 3.3, we describe the whole replication process. Finally, in Sect. 3.4, we explain how to produce a metamorphic binary starting from the sources of an original program.

3.1 Obfuscation step

This section presents the obfuscation step in the self-replication process of our metamorphic engine. The obfuscation process has to work on both the code and the data in a program at the same time.

Code obfuscation: The general code obfuscation scheme detailed hereafter is inspired from [5]. Let P be a program composed of n consecutive instructions $(I_1, \ldots, I_n)$. This program P is split in k consecutive blocks $P = (P_1, \ldots, P_k)$. Each of these blocks contains a random number of instructions. Let σ be a random permutation over the set $[1, n]$ used to randomize P_i blocks. For each P_i block, we define a new block $P'_{\sigma(i)}$ with its transition. This approach is illustrated in Fig. 1. On the left hand, we have an original program P composed of ten instructions whose control flow is represented with arrows. The boxes illustrate the random splitting of P in five blocks. On the right hand, the new program P' is obtained by permutating P_i blocks according to σ.

It is easy to see that whatever the *Control Flow Graph* (CFG) of program P looks like, the execution remains the same if after executing the last instruction of block $P'_{\sigma(i)}$, the first instruction of $P'_{\sigma(i+1)}$ is reached. These transitions, represented with dashed arrows in Fig. 1, are the key points of the obfuscation scheme. For instance, considering the block containing instructions I_1, I_2 and I_3, the execution of instruction I_3 must lead to I_4 as illustrated in P and P'.

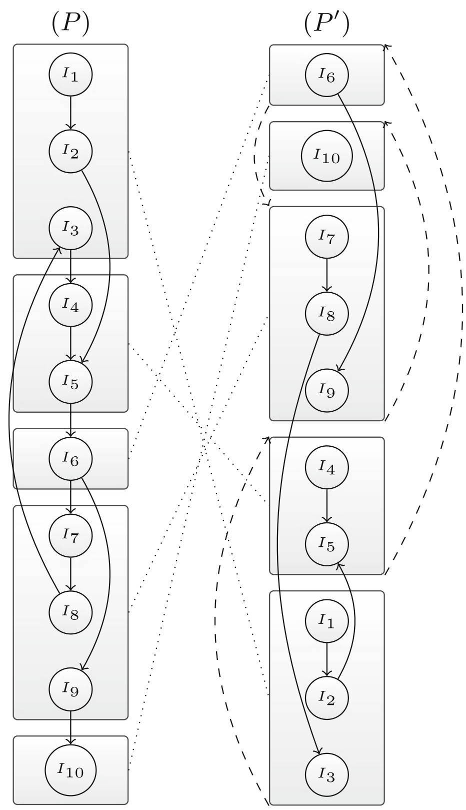

Fig. 1 Illustration of the obfuscation scheme. Original program P on the left and the obfuscated program P' on the right

As the splitting is randomly generated, no syntactic pattern can be directly extracted, according to this approach. Moreover, it was proved in [5] that the static detection of metamorphic malware employing such a technique in a multi path assumption, is an NP-complete problem. In static analysis, the multi path assumption translates the difficulty of branch target evaluation. Indeed, considering a branch instruction, represented as follows, "`if` (*condition*) {*branch1*} `else` {*branch2*}", the *condition* evaluation can be highly complicated by the use of opaque predicates as detailed in [11]. Informally, a predicate is said to be opaque if it has a property which is known to the obfuscator, but which is difficult for the deobfuscator to deduce. Thus, if a program cannot determine the *condition* value, then it has to consider the two branches as possibly executable paths. However, the creation of opaque predicates which are difficult to resolve is a hard task [11]. It is also the case from a metamorphic malware point of view. Instead of focusing on opaque predicate creation, we deliberately choose to take advantage of the malware time detection constraint evoked in [3,16].

In other words, each block P' transition is τ-obfuscated by dynamically computing the target destination. Several approaches were detailed in [3] to τ-obfuscate programs. In order to facilitate time measurement, we decided to use an obfuscated loop which computes the destination address.

So, for the rest of the article, the τ delaying time is measured thanks to the number of iterations in the transitions loops. Here, we only present a sketch of our τ-obfuscation design for two reasons. Firstly, from an ethical point of view, giving a complete description of the implementation could lead an attacker to directly write such a metamorphic engine, which is a non affordable risk. Secondly, according to the experiment result, τ-obfuscation seems to have no impact on current detection tools. So, τ-obfuscation does not appear to be the key component of the metamorphic engine. To achieve τ-obfuscation, the idea consists in choosing a random function f for each transition. Then the target address is determined by the number of compositions of this function f. Of course, this simple loop is obfuscated using classical techniques such as rewriting rules to avoid any pattern.

Data obfuscation: A simple way to protect data is encryption as used in polymorphic malware, for example. In this case, the malware execution begins with a (polymorphic) decryption routine acting on the rest of the code and data. After this decryption, all the code and data represent a possible detection pattern. Thus, a practical detection [24, pp. 451–458] consists in emulating the decryption routine to come back to classical pattern matching detection techniques on the decrypted program.

To avoid such a detection, a better approach consists in decrypting data just before they are used and re-encrypting them just after. By data we mean a block of data which cannot be divided without a loss of semantics (for instance, a string, a switch table, a structure, etc). This technique known as on-the-fly encryption is commonly used in malware protection (DarkParanoid [19] and W32/Elkern [14]). More precisely, let f be a function taking as parameter a data block, denoted D. Our original program P computes the function $f(D)$. Let E be a symmetric encryption scheme. We modify the original program P to get the program P' defined as follows: P' contains (in its binary representation) an encryption key K and the encrypted data $C = E_K(D)$. Then, during its execution, P' starts with the decryption of the encrypted data C. After that, P' computes f with the previous decrypted data D. Finally, P' re-encrypts this data D with the same key K. So, without the knowledge of the key K, the protection of D is guaranteed by the robustness of the encryption scheme.

The data obfuscation process consists in randomizing the key value and its position in such a way that only the piece of code which previously had access to this key has access to the new one. The new program contains the new encryption K' and the new encrypted data $C' = E_{K'}(D)$. In this case, the decryption key can only be discovered by disassembling the code. Thus, the robustness of data obfuscation directly relies on the robustness of the previous code obfuscation guaranteed by τ-obfuscation.

3.2 Modeling step: the necessity of extra information

This section presents the modeling step in the self-replication process of the metamorphic engine. From now on, we consider that the metamorphic engine M is already obfuscated, as presented in Fig. 1. The obfuscated metamorphic engine is denoted M' in the rest of the section.

From its entry point, M' must be able to extract its structure in memory in order to re-obfuscate itself. Without any particular information on the way it was produced, M' would have to disassemble itself as any other external program would have to. In this case, the engine would be confronted with the difficulty of reversing its own obfuscation scheme. So, to easily reverse its own code, M' must embed extra information allowing its de-obfuscation without simplifying the detection.

According to the obfuscation scheme presented in Sect. 3.1, coming back to M means recovering the original sequence of code blocks $(M_1, \ldots, M_n)$ and the original data blocks. More precisely, the extra information to be embedded is composed of:

1. the description of the original sequence of code blocks $(P_1, \ldots, P_n)$;
2. the description of data blocks with their corresponding encryption key;
3. the description of memory references.

With these three elements, the metamorphic engine M' is able to come back to its exact original (de-obfuscated) form M. We shall explain the necessity of references. At binary level, each logical element in a program (a block of data, an instruction, an import table entry, etc.) is represented by its address. As these addresses change during each mutation according to the previous obfuscation scheme, the metamorphic engine must be able to find and update these references according to the new position of the corresponding element. Unfortunately, the exact determination of references in a binary program is difficult.

To illustrate this problem, let us consider the following assembly instruction: `cmp eax, 0402000h`. This instruction compares the value contained in the `eax` register with the hexadecimal value `402000`. Considering the metamorphic engine (or any disassembler), the problem consists in determining the semantics of this value. In other words, is it an address or not? Now, let us consider the two following programs described in C language: both of them declared a constant value `MY_FLAG` in line 1 and a global string `Global1` in line 2. The `main` function only declares a variable in line 6, whose value is supposed to be defined later in the `main` function. The key point is the `if` statement line 8 which compares `var1` with `MY_FLAG` in the first source and with `Global1` in the second source.

Fig. 2 Illustration of the difficulty of precise references evaluation

```
1 #define MY_FLAG 0x402000
2 char     Global1[]="string";
3
4 main()
5 {
6      int var1;
7      ...
8      if (var1==MY_FLAG) {...}
9 }
```

```
1 #define MY_FLAG 0x402000
2 char     Global1[]="string";
3
4 main()
5 {
6      char* var1;
7      ...
8      if (var1==Global1) {...}
9 }
```

Considering the particular case where the compilation process places the `Global1` string at address `402000` in the two resulting binaries, line 8 corresponds to the previous assembly instruction. It is worth mentioning that this extreme academic case is not very probable, but clearly illustrates the problem of the references. Concerning our metamorphic approach, code and data are randomly dispersed throughout the program during the replication. So, considering the previous example, the address of `Global1` will be different after replication. And then, to be correct, in the statement `cmp eax, 402000h`, the hexadecimal value must be updated by the new address of `Global1` only in the second program's binary (Fig. 2).

3.3 Metamorphic engine replication with no constant kernel

At this stage we have illustrated :

1. how to obfuscate a program to guarantee that it cannot be disassembled under a predefined time τ in Sect. 3.1;
2. which information is mandatory to create a program able to reverse this previous obfuscation scheme in Sect. 3.2;

We now have to describe how the metamorphic engine can link these two steps to achieve its self-replication. Figure 3 illustrates this replication process. For the purpose of simplicity, we only present how the description of the original code blocks sequence is used in the replication process.

Let M' be an already-obfuscated version of the metamorphic engine M, as described in Sect. 3.1. M' embeds its own rebuilding information, as presented in Sect. 3.2. More precisely, M' is here composed of 20 instructions $(I_1, \ldots, I_{20})$ distributed in 7 blocks $(M'_1, \ldots, M'_7)$ as represented in (1). Each instruction index represents its execution order, I_1 stands for the first executed one whereas I_{20} is the last instruction. Each block M'_i contains a random number of instructions and a random position in the program M'. At the end of each block, another one denoted τ'_i represents the τ-obfuscated branch whose destination is highlighted by pointed arrows. As previously mentioned, the destination of this block cannot be determined before the τ'_i duration.

Without loss of generality, let us assume that the rebuilding information is used by instruction I_4 to start the modeling step. Then, this instruction has a reference to the first block description represented in (2). This description gives the position and the size of each M'_i block. So, M' can disassemble M'_1, then M'_2 until the last block M'_7. From now, M' has its own instructions sequence $(I_1, \ldots, I_{20})$ in memory as illustrated in (3). The re-obfuscation starts here, as described in Sect. 3.1 whose results is illustrated in (4): new code blocks are randomly generated $(M''_1, \ldots, M''_6)$ with their corresponding τ-obfuscated transitions $(\tau''_1, \ldots, \tau''_6)$. The original code block sequence $(M''_1, \ldots, M''_6)$ is inserted in a new data block represented in (5). The key point consists in updating the reference to this rebuilding information in instruction I_4, to be sure that this instruction will use the new code blocks description. Finally, the entry point of M'' is defined in its header to point to the position of I_1 instruction in M''.

From a detection point of view, rebuilding information presents no constant part nor constant position between the two mutated programs. Thus, we assume that reaching rebuilding information means to be able to disassemble any obfuscated program until identifying the part of the program using this information.[1] In this case, any disassembler would be confronted with the robustness of the code obfuscation scheme. And then detection is delayed during the amount of time defined by τ-obfuscation.

3.4 Embedding the metamorphic engine in another program

We have illustrated how the metamorphic engine can reproduce itself according to its rebuilding information. However, the remaining question is the origin of this information. In other words, how can we get the first obfuscated metamorphic binary? First, our metamorphic engine works at binary level taking advantage of the dissembling difficulty in x86 architecture. Second, the purpose of the engine is to be generic, in order to transform high level language programs to make them metamorphic. Here we only focus on programs written in C language. Thus, we have to modify the compilation process to build the first metamorphic binary in the same way the metamorphic engine does. This is achieved by inserting an obfuscator in the compilation process as illustrated in Fig. 4 step (2).

The compilation process starts normally by taking two inputs programs, the metamorphic engine and the to-be-obfuscated program. First, the compiler produces the

[1] The question of heuristic detection of the permuted code is not mentioned here.

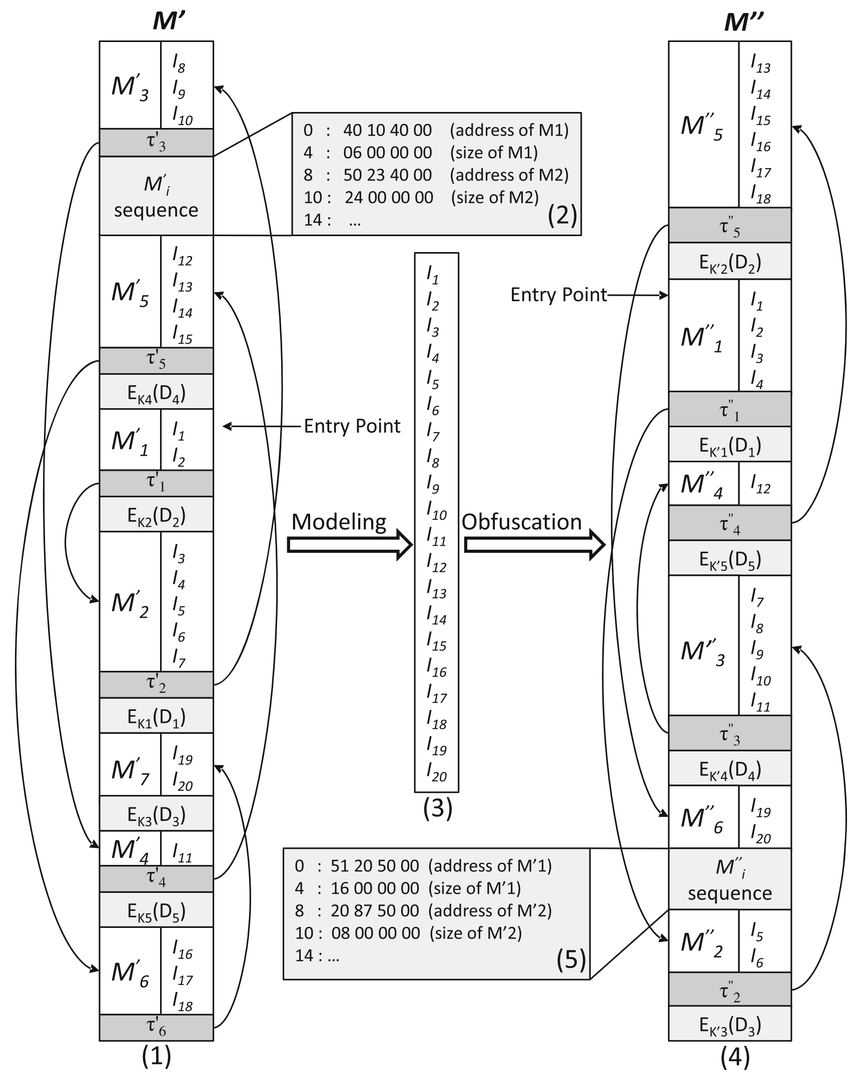

Fig. 3 Illustration of the replication process of the metamorphic engine

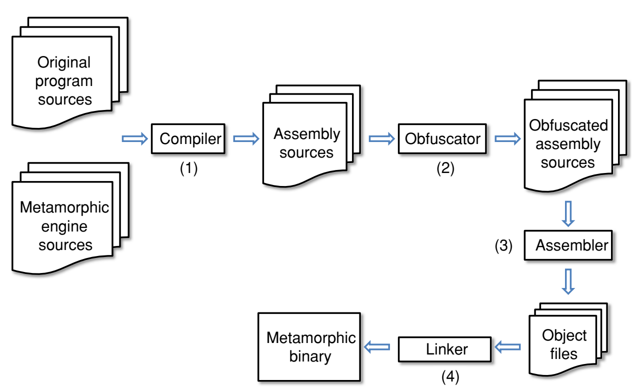

Fig. 4 Illustration of the production of the first metamorphic binary from the metamorphic engine and an original program

corresponding assembly sources. Second, the obfuscator transforms these sources, as presented in Sect. 3.1. The obfuscated assembly sources now contain all the rebuilding information for the whole program. Then, the assembler produces object files which will be linked with additional libraries to obtain the final metamorphic binary just like any classical assembling process.

4 Malware detectors classification

This section aims at empirically evaluating the impact of our metamorphic engine approach on the state of the art detection tools. In Sect. 4.1, we present the way we transform the well-known email-worm MyDoom into a metamorphic one. In Sect. 4.2, we describe the evaluation platform. In Sect. 4.3, we present the results of our experiments.

4.1 Building a metamorphic version of MyDoom

All our experiments are based on the mass-mailing worm MyDoom.A [15] discovered in January 2004. The choice of this malware was motivated by two major reasons:

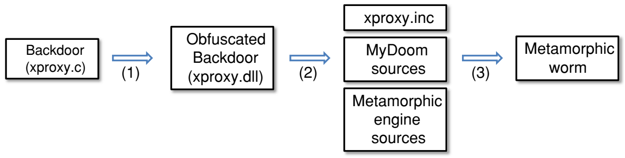

Fig. 5 Illustration of the incorporation of the obfuscated backdoor (xproxy) in the metamorphic email-worm (MyDoom)

1. the worm sources [26] are available in C language, which allow us to directly use our metamorphic engine;
2. according to its virulence (number of emails generated), MyDoom is considered as the most serious email-worm attack ever known [24].

Briefly, MyDoom is a worm propagating through a peer-to-peer client and by emails. Its payload is composed of two parts: first, this worm tries a *Denial Of Service* (DOS) on a specific web site. Second, MyDoom embeds an encrypted *Dynamic Link Library* (DLL) which represents a backdoor listening on a TCP port ranging between 3127 and 3198. This DLL can be viewed as a standalone malware, loaded by the Windows `Explorer.exe` process, and waiting for malicious commands. Thus, we have two malware candidates for detection purpose: MyDoom, and its backdoor. In the original sources of MyDoom, the `CopyFile` function is in charge of the worm replication. To use our metamorphic engine, we have modified the sources of MyDoom to replace all the `CopyFile` calls to the replication entry point of the metamorphic engine. Concerning the backdoor, its detection could make MyDoom suspicious according to heuristics detection techniques. As a non-replicating program the backdoor does not use the metamorphic engine but has to be obfuscated as the worm is.

The generation of the metamorphic email-worm is illustrated in Fig. 5: first the backdoor is obfuscated as explained in Sect. 3.1 to obtain the obfuscated backdoor (`xproxy.dll`) in step (1). Then, the backdoor binary has to be encrypted as the worm normally does. This encrypted binary is then translated as a table of hexadecimal values in a source file (`xproxy.inc`). This step is denoted in (2). Finally, the metamorphic email-worm is produced as illustrated in Fig. 4 from the previous `xproxy.inc` file, the metamorphic engine and MyDoom sources in step (3).

4.2 Evaluation platform

To observe the malware's behavior in a safe and protected environment, a target platform was installed. The adopted solution consisted in using virtual machines for two reasons: first, to prevent any infection of the real operating system from the malware. On this subject, we verified beforehand that MyDoom did not try to detect any virtualization environment. Indeed, malwares are used to changing their behavior in case of virtualization. Second, virtual machines allowed us to easily come back to a clean state independently of detection success by restoring the safe machine image. The evaluation platform was composed of two components, namely the guest and the host machine.

Guest machine: VMWare workstation was chosen as the emulating environment. Windows XP Pro SP3 was installed with up-to-date hot fixes to represent a personal user configuration. To observe the worm propagation, a mail client and a peer-to-peer one were configured. An ISP account was also defined with different parameters and especially the SMTP server address. This guest machine configuration was cloned according to the number of antivirus to be tested. An antivirus program was installed on each configuration.

Host machine: A bridge was installed between the two machines to establish a network communication between them. A fake SMTP server listening on TCP port 25 was in charge of collecting the worm's mail. All the guest traffic was oriented toward the bridge to reach the host machine.

In order to validate the metamorphic engine replication, and to bring representative results, each sample of malware used in the followings experiments was produced as follows:

1. a metamorphic email-worm obtained, as illustrated in Fig. 5, was installed on a guest machine containing a detection product. The parameters of the metamorphic engine were configured according to the experiments (code block sizes and τ iteration values).
2. this worm was then executed on the guest machine until two mutated worms were obtained if the worm was not detected. These two worms were collected from the peer-to-peer client and from emails by the host machine.
3. the virtual environment was finally restored to a clean state and this process was renewed (step 2) with the previously collected malware until the desired sample of worms was obtained.

4.3 Experiment results

To be as general as possible, we started with 16 of the most used antivirus software regardless of their detection techniques. In terms of license several products present ambiguities concerning black-box evaluations: *"You shall not use this Software in automatic, semi-automatic or manual tools designed to create virus signatures, virus detection routines, any other data or code for detecting malicious code or data."* To be as neutral as possible, all the results are given anonymously denoted by AV1 to AV16. All detection software were configured according to their best detection capacities.

Concerning the metamorphic engine parameters, τ-obfuscation was initialized to a single iteration and the code blocks size was set to contain from 1 to 5 instructions. For detection purpose, each worm was installed on a guest machine and submitted to on demand detection. Then, non detected malware were executed until detection, or mail and peer-to-peer propagations.

Two samples of malware differing only in their replications were submitted to antivirus products: the first one used direct replication API calls (`CopyFile`), whereas the second one used the metamorphic engine replication functionality. The interest of such a distinction lies in the difficulty of determining the self-replication of the metamorphic engine whereas it is quite simple to identify a direct copy. The detection results concerning the two submitted samples of malware are presented in Table 1 with their corresponding observed detection technique.

According to Table 1, which presents the observed results obtained from the two submitted worms samples, four classes of detection techniques can be extracted:

1. behavioral monitoring software represented by AV1 and AV2;
2. behavioral blocker products represented by AV3 to AV8;
3. heuristic-based detection tools represented by AV9;
4. form-based detection software unable to detect any obfuscated worm or metamorphic ones (AV10 to AV16);

It is worth mentioning that the backdoor action (listening on a specific TCP port) is detected in all cases by the system firewall. The first three detection classes are detailed hereafter.

Behavioral monitoring results: This class of detectors includes two software (AV1 and AV2) able to detect all the obfuscated email-worms but no metamorphic one. This result tends to illustrate that AV1 and AV2 considered self-copying as a key component for detection purposes. However, the self-replication problem is known to be a difficult one [10]. Our results show that direct replication by calling the `CopyFile` function was detected but not the metamorphic engine replication process illustrated in Fig. 3. AV1 gave no more information to help understand the precise detection technique used. It seems that sensible events (files creations, file and registry modifications, self-copying, etc.) were correlated to identifying a generic class of malware (here trojans). AV2 detected all the obfuscated and metamorphic email-worms during the installation of the backdoor. As illustrated by the results, if the backdoor is not embedded in the tested worms, then no metamorphic worms is detected.

Behavioral blocker results: All these software (AV3 to AV8) required a user decision concerning each detected suspicious action. AV3 blocked each file containing an executable program disguised by harmless file extension. This happened during mail creation with a probability of 40% set in the source code. More precisely, in this case, the email-worm packed itself in a temporary directory with a `.tmp` extension before encoding this copy as a mail attachment. Consequently, all of these temporary files were detected as suspicious by AV3. AV4 to AV6 blocked all the malware attempts to become resident by registry modifications. AV7 blocked all write accesses to the system directory. Finally, AV8 monitored several behavior with different level of risks and gave the following warnings for all the metamorphic email-worms:

1. modifying your computer so that another computer can access it;
2. copying an "executable" file to a sensitive area of your system;
3. registering itself in your "Windows System Startup" list;
4. copying another program to an area of your computer that shares files with other computers;
5. connecting to the Internet in a suspicious manner to send out emails.

Here, it is worth mentioning that this product was not able to detect the self-replication of the metamorphic engine. Indeed expressions "an executable" in 2 and "another program" in 4 confirmed the self-replication detection difficulty as for AV1 and AV2. Moreover, it was verified that AV8 could detect self-replication on the obfuscated versions of MyDoom. However, in all cases a warning was generated for any program copy as well as a self-copying. Moreover, these different warnings were not correlated to identify a specific malware behavior. Behavioral blocking is a proactive detection technique preventing any malicious action before execution. Each of these action relies on a single system call. So, τ-obfuscation is useless on this class of detectors.

Heuristic-based detection results: AV9 detected all the malware according to their binary files and not during their executions. More precisely, all the malware were detected under the label "Heur_PE virus" which suggests that heuristics were used for detection purpose. To validate this heuristics-based detection assumption, we created 3 samples of malware with different τ values (1,500, and 1 000 000 iterations). Each of these samples was composed of 4 groups of 100 malware with different code block sizes. Figure 6 gives the corresponding detections rates.

According to these results, the detections rates seem proportional to the code block sizes. Moreover, τ-obfuscation appears to have no impact on detection. This confirms that AV9 used heuristics-based detection approach to recognize

Table 1 Detection results obtained on 16 antivirus products with 100 obfuscated worms (first column) and 100 metamorphic ones (second column)

Software	Obfuscated worms	Metamorphic worms	Observed detection technique
AV1	100/100 detected as generic Trojan	0/100	behavior monitoring
AV2	100/100 detected for suspicious file actions (self copies)[a]	0/100[a]	behavior monitoring
AV3	40/100 blocked for suspicious files actions	40/100 blocked for suspicious files actions	file blocker
AV4 AV5 AV6	100/100 blocked for registry modifications (residency)	100/100 blocked for registry modifications (residency)	registry blocker
AV7	100/100 blocked for system directory write access	100/100 blocked for system directory write access	file blocker
AV8	100/100 blocked for suspicious actions	100/100 blocked for suspicious actions	actions blocker
AV9	10/100 detected as “Heur_PE virus”	10/100 detected as “Heur_PE virus”	heuristic form-analysis
AV10	0/100	0/100	no detection
⋮	⋮	⋮	⋮
AV16	0/100	0/100	no detection

[a] All the malware (obfuscated and metamorphic) were blocked for suspicious DLL installation. For this antivirus, the presented results correspond to worms without any payload (backdoor)

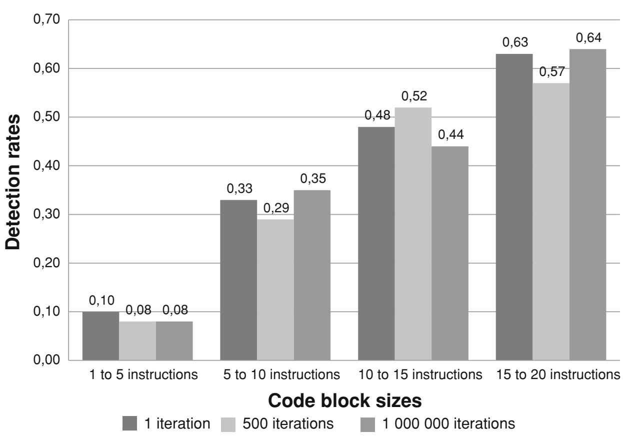

Fig. 6 Detection rates of AV9 according to code block sizes and τ values

these metamorphic malware. No information was given by the product on these specific heuristic detection techniques.

4.4 Discussion

Detection results: the previous experiments emphasize the two main representative behavioral detection techniques used in current industrial malware detectors: behavioral monitoring and behavioral blocking. AV1 and AV8 produce two interesting results, each one representing a different class of detection techniques. Indeed, AV1 is able to correlate some suspicious actions to identify generic malware behavior. Unfortunately, complex self-replications such as metamorphic ones are not detected, leading to a 100% false negatives results when confronted with our experimental metamorphic worms. AV8 can detect all elementary suspicious actions which could describe the behavior of the email-worm MyDoom. Unfortunately, these events are not correlated to identify the generic email-worm behavior. In all cases, it appears to be too immature to evaluate the impact of τ-obfuscation on the current state of the art detection products. Finally, the detection of metamorphic malware appears to be a real challenge for current detection products.

Why metamorphic malware? Syntactic modifications are required to observe the different detection techniques used in antivirus products and especially heuristics and behavioral ones. Currently, these form modifications are generally obtained by packers actions. In this study, we focus on metamorphism as we believe it could represent a more worrying threat than packers. Indeed, starting with a worm binary, some packers can produce several packed variants of this original worm. However, each variant will keep the same form during replication. For instance, considering MyDoom, the `CopyFile` API used for replication purpose

Table 2 Detection results obtained on the packed version of MyDoom.A

AV1	Type_Win32
AV2	Sus/UnkPacker
AV3	TR/Crypt.CFI.Gen Trojan
AV4	*registry modifications detected*
AV5	behaves like Win32.P2P-worm
AV6	*temporary files detected as* Generic Malware.a!zip
AV7	Type Win32
AV8	*many suspicious actions detected* (see Sect. 4.3)
AV9	Win32 MyDoom
AV10	∅
AV11	W32/Malware
AV12	W32/Atak!Generic
AV13	suspicious
AV14	∅
AV15	∅
AV16	Win32:trojan-gen

acts by duplicating the image file of the email-worm. So, a packed form of MyDoom will produce several identical packed worms. On the contrary, a metamorphic malware will produce different forms after replication as illustrated in Sect. 3.3.

To practically evaluate the difference between packers and metamorphic malware, we have implemented a simple packer based on the LZ77 [29] algorithm. The original binaries (worm and backdoor) were packed before being submitted to the previous antivirus programs. Detection results are given in Table 2 which presents better detection results than with metamorphic email-worms.

Concerning false positive results, our approach consists in a metamorphic self-replication technique which is not present in legitimate programs. In this sense, the issue of false positive results is out of the scope of this article. However, a more interesting point is the detection results of benign programs after application of our metamorphic engine. So, we have applied our metamorphic engine to benign programs before submitting them to detection. In all cases, no antivirus program appears able to detect the self-replication. In other words, the obtained metamorphic programs are considered as "suspicious" as they were before application of the metamorphic engine.

5 Conclusion and future works

In this paper, we have proposed an approach of a generic metamorphic engine based on advanced code transformation techniques. Describing the process of the metamorphic engine self-replication, we have illustrated the difficulty of detecting it. From a static point of view, the obfuscation scheme was designed to avoid any syntactic signature which could represent a possible detection pattern. Moreover, classical static analysis techniques based on data flow propagation or slicing are limited by the robustness of code obfuscation.

To evaluate the threat represented by self-replicating metamorphic malware, we applied our metamorphic engine to a representative email-worm to assess current industrial antivirus products detection capabilities. According to the obtained results, we can consider that no tested detection tool is able to reliably detect this class of malware. Concerning static detection products, only one seems able to detect samples of malware according to some heuristics. Concerning behavioral detection tools, two techniques seem to be used: behavioral monitoring and behavioral blocking. Unfortunately, our experiments found some worrying limitations in these detection techniques. Indeed, behavioral monitoring fails to identify the replication process of the proposed metamorphic engine, leading to false positive results. Behavioral blocking, which consists in suspending some suspicious actions, relies on the user decision to achieve system security and appears unable to detect a global malicious behavior. Consequently, behavioral detection seems an early detection strategy, for which current implementation is not yet effective against metamorphism.

These experiments results allows us a black-box classification of antivirus products based on their observable detection techniques through their response to metamorphic malware. This approach presents an original way concerning the evaluation and the classification of current detection tools as a supplement of existing evaluation tests.

Finally, this work aimed at focusing on the threats that metamorphic malware could represent. By considering the practical case where no user can decide on the malicious aspect of an action, the question is about the automatic detection of such metamorphic threats. As underlined in [21], we believe that alert correlation would offer interesting perspectives in malware detection, as has already been done concerning intrusion detection. From now, our work will be aimed at dynamically detecting this type of malware.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
2. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (Im)possibility of Obfuscating Programs. Crypto '01, Lecture Notes in Computer Science, vol. 2139, pp. 1–18 (2001)
3. Beaucamps, P., Filiol, É.: On the possibility of practically obfuscating programs—towards a unifed perspective of code protection.

In: Bonfante, G., Marion, J.-Y. (eds.) WTCV'06 Special Issue. J. Comput. Virol. **2**(4) (2006)
4. Bonfante, G., Kaczmarek, M., Marion, J.Y.: Architecture of morphological malware detector. J. Comput. Virol. **5**(3), 263–270 (2008)
5. Borello, J.M., Mé, L.: Code obfuscation techniques for metamorphic viruses. J. Comput. Virol. **4**(3), 211–220 (2008)
6. Bruschi, D., Martignoni, L., Monga, M.: Detecting self-mutating malware using control-flow graph matching. In: Detection of Intrusions and Malware & Vulnerability Assessment. Lecture Notes in Computer Science, vol. 4064, pp. 129–143. Springer, Berlin (2006)
7. Chomsky, N.: Three models for the description of languages. IRE Trans. Inform. Theory **2**, 113–124 (1956)
8. Chomsky, N.: On certain formal properties of grammars. Inform. Control **2**, 137–167 (1959)
9. Christorodescu, M., Jha, S.: Static analysis of executable to detect malicious patterns. In: Proceedings of the 12th USENIX Security Symposium, pp. 169–186 (2003)
10. Cohen, F.: Computer Viruses. PhD thesis, University of Southern California (1986)
11. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Principles of Programming Languages POPL98, pp. 184–196 (1998)
12. Collberg, C., Thomborson, C., Low, D.: A Taxonomy of Obfuscating Transformations. Technical Report 148, University of Auckland, New Zealand (1997)
13. Driller, T.M.: Metamorphism in Practice. 29A E-zine, vol. 6 (2002)
14. Ferrie, P.: Un combate con el kernado. Virus Bulletin, pp. 8–9 (2002)
15. Ferrie, P., Lee, T.: W32.mydoom.a@ mm. http://www.symantec.com/security_response/writeup.jsp?docid=2004-012612-5422-99&tabid=2 (2004)
16. Filiol, É.: Metamorphism, formal grammars and undecidable code mutation. Int. J. Comput. Sci. **2**(1), 70–75 (2007)
17. Jacob, G., Debar, H., Filiol, É.: Behavioral detection of malware: from a survey towards an established taxonomy. J. Comput. Virol. **4**(3), 251–266 (2008)
18. Jacob, G., Debar, H., Filiol, É.: Functional polymorphic engines: formalisation, implementation and use cases. J. Comput. Virol. **5**(3), 247–261 (2008)
19. Kaspersky, E.: Darkparanoid—Who me? Virus Bulletin, pp. 8–9, January (1998)
20. Lakhotia, A., Kapoor, A., Kumar, E.U.: Are metamorphic viruses really invincible? Virus Bulletin, pp. 5–7 (2004)
21. Morin, B., Mé, L.: Intrusion detection and virology: an analysis of differences, similarities and complementariness. J. Comput. Virol. **3**, 39–49 (2007)
22. Preda, M.D., Christorodescu, M., Jha, S., Debray, S.: A Semantic-based approach to malware detection. In: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2007)
23. Spinellis, D.: Reliable identification of bounded-length viruses is NP-complete. IEEE Trans. Inform. Theory **49**(1), 280–284 (2003)
24. Szor, P.: The Art of Computer Virus Research and Defense. Addison-Wesley, Reading (2005)
25. Tip, F.: A survey of program slicing techniques. J. Program. Lang. **3**, 121–189 (1995)
26. VX Heavens. http://vx.netlux.org
27. Walenstein, A., Mathur, R., Chouchane, M., Lakhotia, A.: The design space of metamorphic malware. In: Proceedings of the 2nd International Conference on i-Warfare & Security (ICIW), pp. 241–248 (2007)
28. Walenstein, A., Mathur, R., Chouchane, M.R., Lakhotia, A.: Normalizing metamorphic malware using term rewriting. In: SCAM 2006: The 6th IEEE Workshop Source Code Analysis and Manipulation, pp. 75–84 (2006)
29. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inform. Theory **23**(3), 337–343 (1977)