J Comput Virol (2013) 9:49–58
DOI 10.1007/s11416-012-0174-z

ORIGINAL PAPER

Metamorphic worm that carries its own morphing engine

Sudarshan Madenur Sridhara · Mark Stamp

Received: 14 June 2012 / Accepted: 5 October 2012 / Published online: 25 October 2012

Abstract Metamorphic malware changes its internal structure across generations, but its functionality remains unchanged. Well-designed metamorphic malware will evade signature detection. Recent research has revealed techniques based on hidden Markov models (HMMs) for detecting many types of metamorphic malware, as well as techniques for evading such detection. A worm is a type of malware that actively spreads across a network to other host systems. In this project we design and implement a prototype metamorphic worm that carries its own morphing engine. This is challenging, since the morphing engine itself must be morphed across replications, which imposes restrictions on the structure of the worm. Our design employs previously developed techniques to evade detection. We provide test results to confirm that this worm effectively evades signature and HMM-based detection, and we consider possible detection strategies. This worm provides a concrete example that should prove useful for additional metamorphic detection research.

1 Introduction

Metamorphism is the process of transforming a piece of software into unique instances [22]. In metamorphic software, copies of the software are functionally equivalent but their internal structure differs. Metamorphism is used by virus writers to avoid detection by antivirus software, which primarily use signature-based techniques [3].

Metamorphism provides virus writers the opportunity to develop malware that is undetectable, with respect to signature detection [7,11]. Therefore, it is natural to expect an increase in volume as well as complexity of metamorphic viruses in the future.

Although metamorphic viruses have been extensively studied [2,9,13,20,25,26,28], a metamorphic worm presents significant challenges. Metamorphic viruses do not need to carry their own morphing engine. For nearly all highly metamorphic viruses, such as NGVCK [21], the metamorphic generator is separate from the virus body.

Examples of hacker-produced metamorphic malware that carry their own morphing engine do exist, but most are trivial, and none has proven difficult to detect in practice. Relatively sophisticated examples of the genre include Lexotan32 [19] and MetaPHOR (also known as W32.Simile and W32.Etap) [15]. MetaPHOR, for example, is large and complex, consisting of more than 14,000 lines of assembly code, with more than 90 % of its code dedicated solely to morphing. The author of MetaPHOR has described its development and workings in [15], which is a rarity for hacker-produced malware, but it still extremely opaque, poorly documented, and very difficult to work with. In fact, our attempts to work with MetaPHOR convinced us that an alternative was needed for our planned research in this field.

Unlike viruses, worms are self-propagating [3], and therefore a metamorphic worm would, almost certainly, need to carry its own morphing engine. Since the morphing engine itself can act as a signature, such a worm must morph its morphing engine—as well as the actual worm body—across replications. This presents a significant complication and imposes some restrictions on the structure of the morphing engine.

In this paper, we develop and analyze a worm that carries its own morphing engine. That is, the morphing engine morphs itself and the worm body across replications. The resulting metamorphic worms are evaluated based on the

S. Madenur Sridhara · M. Stamp (✉)
Department of Computer Science,
San Jose State University, San Jose, CA, USA
e-mail: stamp@cs.sjsu.edu

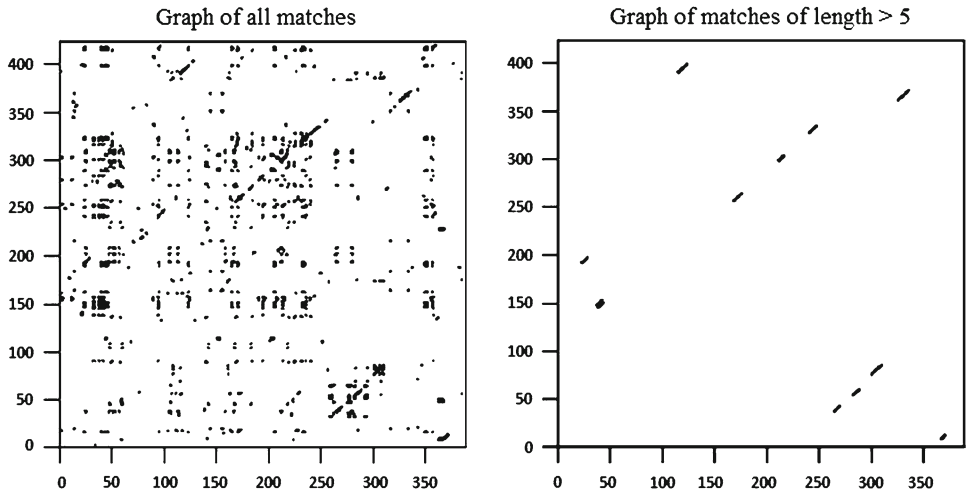

Fig. 1 *n*-gram similarity for two NGVCK viruses [28]

lack of similarity between successive generations [17] and their ability to evade detection [28].

This paper is organized as follows. In Sect. 2, we briefly cover relevant background information on malware, metamorphism, similarity measures, and hidden Markov models. Then in Sect. 3 we outline the design of our metamorphic worm. Section 4 contains experimental results that measure the degree of metamorphism achieved by our worm, as well as its success at evading detection. Finally, Sect. 5 gives our conclusions and suggestions for future work.

2 Background

In this section, we briefly discuss metamorphic techniques, methods for measuring binary similarity, and hidden Markov models. These topics are relevant to the work presented in the remainder of the paper.

2.1 Metamorphic techniques

The metamorphic worm described in this paper makes use of several morphing techniques. Here, we briefly discuss metamorphic techniques; for more details see [3] and [4].

Register swapping is one of the simplest metamorphic techniques. Swapping operand registers can change signatures. However, opcode sequences remain unchanged when using this technique.

Another simple morphing technique is reordering subroutines. More generally, we can transpose any instructions that do not have any inter-dependency.

Garbage code instructions can be of two types, namely, instructions that are not executed and instructions that are executed, but have no effect. When we want to distinguish between these two cases, we refer to instructions that are not executed as "dead code," while code that has no effect is "do-nothing" code.

In contrast to garbage code insertion, instruction substitution consists of replacing existing code with different—but equivalent—instructions. For example, `MOV R1, R2` is equivalent to `PUSH R2` followed by `POP R1`.

Formal grammar mutation can be viewed as a generalized morphing technique [4,10,29]. A morphing engine can be viewed as a non-deterministic automata, where transitions are possible from instructions to other instructions [29]. We can then apply formal grammar rules to easily create viral copies with great variation; see [29] for some elementary examples.

2.2 Binary similarity

To evade signature detection, it is necessary that our metamorphic worm produce variants that are significantly different from each other. Ideally, two morphed viruses should be about as different as two randomly selected executable files. Consequently, to evaluate a metamorphic generator, we need a way to measure the degree of metamorphism. That is, we need a measure of "distance" between executable files.

Many binary similarity measures have been studied. In Sect. 4, we employ two such measures, namely, *n*-gram similarity [17] and opcode graph similarity [20]. Here, we briefly summarize these two techniques.

The *n*-gram similarity measure developed in [17] compares subsequences of instructions in two assembly program files and yields a score that measures the percentage of similarity between the two files. This measure has been used in subsequent research to measure the effectiveness of metamorphic generators [13,28]. For example, a comparison between two NGVCK metamorphic viruses [21], is presented in [28] and reproduced here in Fig. 1. The left part of the graph

shows matching subsequences prior to noise removal, while the right half shows matching subsequences retained after eliminating noise. The final similarity score in this particular instance is 21 %.

A method for measuring similarity between executable files using opcode graphs is developed and applied to metamorphic viruses in [20]. This method involves creating weighted directed graphs based on opcodes extracted from executable files. More precisely, each extracted opcode corresponds to a node in the directed graph. A directed edge is inserted from a node to every successor node that occurs in the file, that is, each edge represents a pair of consecutive opcodes. Edge weights represent transition probabilities to successor nodes. A score is obtained by comparing the resulting directed graphs. Other graph-based similarity measures have appeared in the literature; see, for example [1] and [5,6].

2.3 Hidden Markov models and metamorphic detection

Recently, there has been research focused on the use of machine learning techniques for malware detection. In particular, hidden Markov models (HMMs) have been used for analysis and detection of metamorphic viruses [2,9,13,20,25,26,28]. A method is presented in [28] in which an HMM is trained to detect metamorphic malware. We use this HMM-based technique in Sect. 4 to measure the effectiveness of the detection-evading strategy employed by our metamorphic worm. In the remainder of this section, we briefly discuss HMMs and how they are applied to the metamorphic detection problem.

A hidden Markov model (HMM) is a statistical model of a Markov process where the states are unknown [23]. A trained HMM can be viewed as a representation of the data on which it was trained.

The use of HMMs for metamorphic virus detection is explained in detail in [27,28]. The basic objective is to train an HMM using opcodes extracted from viruses belonging to a particular metamorphic family. The trained HMM will, in effect, represent the statistical properties of the virus family. Using this trained HMM, we can compute a score for any given program—based on its extracted opcode sequence—to determine how "close" the file is to the virus family that the HMM represents. We can then classify the file based on a predetermined threshold.

There has been some research on metamorphic code that can evade HMM-based detection. The method presented in [13] relies on dead code insertion from benign files.

The results in [12] show that, with an increase in the amount of carefully selected dead code from benign files, the average scores for viruses tends toward the scores for benign files. Furthermore, the results in [12] indicate that inserting long sequences of opcodes, (e.g., entire subroutines) is more effective at defeating HMM detection than inserting an equivalent amount of dead code in the form of multiple small fragments. Thus, the worm presented in this paper uses a detection-evading strategy based on dead code insertion, where the dead code is inserted in blocks.

3 Design and implementation

In this section, we consider the design of our metamorphic worm and provide some relevant implementation details. As discussed above, the fact that the worm carries its own morphing engine presents a significant challenge since both the worm body and the morphing engine itself must be morphed. In addition, we discuss the defenses that our metamorphic worm incorporates, and we point out how these make static detection more difficult.

3.1 Worm structure

The worm includes the following components.

1. Body—This is the central component that controls the life cycle of the worm. The worm body controls and coordinates the activities of all the other active components of the worm.
2. Disassembler—This component disassembles the worm binary and extracts instructions from it.
3. Morphing engine—The morphing engine operates on the set of disassembled instructions by removing old dead code instructions, adding new dead code, and it also employs equivalent instruction substitution.
4. Reassembler—The reassembler restructures the control flow of the morphed worm body and converts the code to binary.
5. Payload—This is the code that will execute on every infected computer. For a malicious worm, the payload would, of course, be malicious, but for our implementation, the payload is completely benign—we simply append a line of text to a temporary file.
6. Padding blocks—These are blocks of dead code that are replaced from generation to generation. The purpose of this code is to make the worm statistically similar to normal files and thereby evade HMM detection [13]. These blocks also help us to avoid relocating sections and simplify other book-keeping operations.

Figure 2 illustrates the structure of the various components of the worm. In addition, the layout of the components within the address space of the worm is illustrated in Fig. 3, where Pad_block_1 and Pad_block_2 are the padding blocks.

Fig. 2 Metamorphic worm components

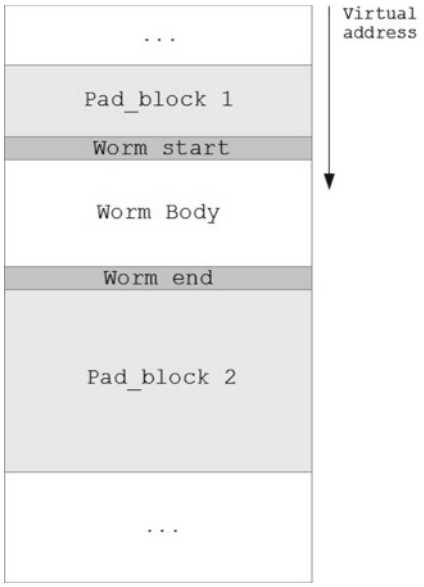

Fig. 3 Metamorphic worm memory layout

3.2 Worm implementation

The worm uses two of the metamorphic techniques discussed in Sect. 2.1. Specifically, the worm employs equivalent instruction substitution and garbage code insertion.

For equivalent code substitution, we concentrate on the `MOV` and the `XOR` instructions. We chose the `MOV` opcode since it is abundant in binaries, and `XOR` was selected since it is often used in our substitutions for `MOV`; by substituting `MOV` for `XOR`, we tend to maintain the first order opcode statistics.

Table 1 Equivalent instruction table

Instruction	Equivalent	Action
`0x48 0x89 0xc3`	`0x48 0x31 0xdb` `0x48 0x01 0xc3`	NULL
`0x48 0x89 0xc1`	`0x48 0x31 0xc9` `0x48 0x01 0xc1`	NULL
⋮	⋮	⋮

After the disassembler has disassembled the worm-containing portion of the executable image, the morphing engine scans for possible equivalent instruction substitution. Substitutions are made for each candidate instruction with a fixed probability. This is achieved using a substitution table—the initial part of the code substitution table appears in Table 1; the complete substitution table used by our morphing engine can be found in [14].

The first column in Table 1 correspond to the instructions

`MOV RBX, RAX` and `MOV RCX, RAX`

respectively. The second column correspond to the equivalent instructions

```
XOR RBX, RBX;
ADD RBX, RAX;
```
and
```
XOR RCX, RCX;
ADD RCX, RAX;
```

respectively. In these examples, one instruction has been replaced by two instructions.

The "action" field in Table 1 is the address of a label inside the morphing function which performs actions specific to the instructions that were substituted, such as modifying addresses. If no additional action is necessary, the action field is set to `NULL`.

Our metamorphic worm also employs garbage code insertion. Garbage code of the "do-nothing" variety (i.e., the code is executed, but it has no effect) is inserted throughout the worm. Garbage code of the "dead code" variety (i.e., the code is never executed) is inserted in the padding blocks.

Some care must be taken when inserting do-nothing code—the effect of such instructions on the `RFLAGS` register must be carefully considered since there is the potential to inadvertently alter the control flow. Control flow instructions use bits in the `RFLAGS` register to decide which code path to take. If a dead code instruction which manipulates `RFLAGS` is inserted before a control flow instruction, it can change the path taken by the next control flow instruction. Consequently, we are careful not to insert do-nothing code in positions where it could change the control flow.

Examples for do-nothing code employed by our metamorphic worm include

`ADD RAX, 0` and `SUB RBX, 0` and `XOR RAX, 0`.

Table 2 Worm function

```
run "payload" // actual intent of the worm
open own binary image by reading /proc/self/exe
read data and book-keeping info from own ".data" section
disassemble worm body excluding padding blocks:
   for each disassembled instruction:
      add a unique label and store virtual address
build symbol table:
   for each disassembled instruction INS:
      if control flow instruction
         add (INS.label, INS.target.label) to symbol table
morph:
   // see Table 3
reassembly:
   fix control flow in list of morphed instructions
patch new binary:
   replace padding blocks from benign binaries
   create new binary image
   write binary to disk
propogate:
   // for completeness only, real worm relies on exploits
   "rcp" or "scp" the binary to surrounding IP addresses
```

A complete list of do-nothing code used by the worm is given in [14].

The overall operation of the worm is summarized in Table 2. Note that the `Morph` function that appears in Table 2 is summarized in Table 3.

The worm has been implemented on Linux in the Intel x86_64 architecture. The programming language used to implement the worm is C. The compiler used to build the worm is GCC, version 4.6.2 build 20111027, and the resulting format of the executable image of the worm is ELF64.

The worm only links directly to libc and libdl. The libraries dynamically loaded during run-time are libbfd and libopdis. Libraries libc and libdl are part of the core of any Linux distribution, while libbfd is part of the GNU Binutils [16] package, and is found on most Linux distributions. Libopdis is an independent library licensed under GNU LGPL as of version 1.0.4 [18]. Libopdis extends the libopcodes library [16] by offering algorithms for linear and control-flow disassembly, instruction and operand objects that are suitable for analysis.

4 Experiments

In this section, we evaluate the effectiveness of our metamorphic worm in terms of its morphing ability and its ability to evade statistical detection. To measure the degree of morphing, we use the n-gram similarity and the opcode graph similarity techniques discussed in Sect. 2.2. To measure the worm's ability to evade statistical detection, we test it using the HMM-based detection technique discussed in Sect. 2.3.

For the worm to be effective at evading signature based detection, the worm bodies in different generations cannot be too similar to one another. At the same time, the worm files must be sufficiently similar to benign files so that they are not easily distinguishable based on a similarity threshold [20].

An effective means of evading statistical-based detection is to make the worm variants statistically similar to benign binaries. To accomplish this, we use long sequences of instructions from benign executable files to fill the padding blocks. This is consistent with the HMM evasion technique in [13].

4.1 Test data

For each experiment, 100 generations of the worm were produced and 20 benign files were selected. The list of benign files are given in Table 4. From the 100 worms, 80 worms were chosen to train the HMM. The remaining 20 worms and all benign files were scored using the trained HMM. The

Table 3 Worm `morph` function

```
morph:
   initialize substitution and dead code instruction tables
   for each disassembled instruction INS:
      if INS is a dead instruction:
         if INS is a control flow target
         (determined from symbol table)
            change target to next disassembled instruction
         ignore dead instruction
      // probability to insert dead code instruction = 0.33
      if adding new dead code instruction:
         choose dead code instruction randomly from table
            add new dead code instruction to morphed
            instruction list
      // if possible, morph instruction with probability 0.33
      if morphing:
         check substitution table for a suitable entry
            if valid entry exists:
               add instructions to be substituted to
               morphed instructions list
   recalculate virtual addresses
```

Table 4 Mapping from Benign file ID to actual executable file

Benign file	Executable file	Benign file	Executable file
BEN_0	/usr/bin/as	BEN_10	/bin/mount
BEN_1	/bin/date	BEN_11	/usr/bin/nasm
BEN_2	/bin/dmesg	BEN_12	/usr/bin/nm
BEN_3	/usr/bin/file	BEN_13	/usr/bin/objdump
BEN_4	/usr/bin/gcc	BEN_14	/usr/bin/readelf
BEN_5	/usr/bin/size	BEN_15	/bin/rm
BEN_6	/bin/grep	BEN_16	/bin/sleep
BEN_7	/usr/bin/kill	BEN_17	/usr/bin/strip
BEN_8	/usr/bin/ld	BEN_18	/bin/systemctl
BEN_9	/bin/mknod	BEN_19	/bin/touch

worm files in the test set were named MWOR_0, MWOR_1, . . ., MWOR_19, while the benign files were named BEN_0, BEN_1, . . ., BEN_19 (as indicated in Table 4).

The padding blocks of the MWOR files were randomly chosen blocks of code from one or more of the benign (BEN) files. Replacing the padding block randomly from the chosen benign file set in Table 4 is part of the worm's functionality.

We computed both n-gram and opcode graph similarity between pairs of worms, between worms and benign files, and between pairs of benign files. We also trained an HMM classifier on worm files and the resulting model was used to classify benign files and worm files in the test set. Next, we discuss our results for each of these test cases.

Table 5 n-gram similarity statistics

Case	Mean	Variance	Minimum	Maximum
Worm vs worm	0.190862	0.007521	0.120562	0.526146
Worm vs benign	0.139772	0.001732	0.050677	0.234278
Benign vs benign	0.263479	0.006037	0.176202	0.450179

4.2 n-gram similarity

The n-gram similarity technique outlined in Sect. 2.2 was used to measure similarity between opcode sequences extracted from different generations of the worm and from benign executable files. The objective here is to assess whether common signatures can be extracted from worm executable files, so our worm padding blocks were excluded from the assessment. When comparing a worm body to a benign file, a randomly selected sequence of instructions of length equal to that of the worm body was chosen from the benign files.

Table 5 summarizes the results of 20 n-gram similarity score calculations comparing consecutive generations of the worm, along with analogous results for the similarity scores

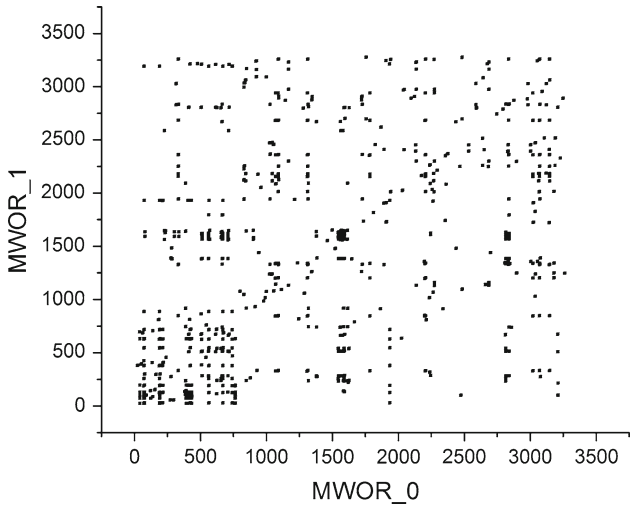

Fig. 4 n-gram similarity—worm vs worm

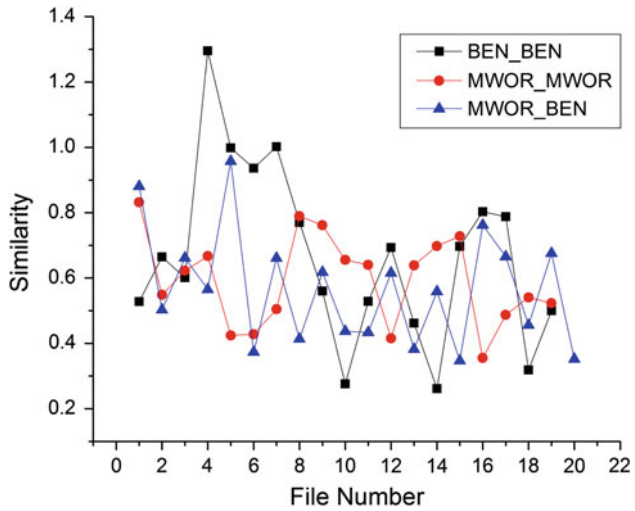

Fig. 5 Opcode graph similarity

Table 6 Opcode graph similarity statistics

Case	Mean	Variance	Minimum	Maximum
Worm vs worm	0.592744	0.017882	0.355481	0.831699
Worm vs benign	0.565945	0.028684	0.347320	0.957393
Benign vs benign	0.667563	0.068312	0.261346	1.295305

between worms and benign files, and, finally, between pairs of worm files. These results show that the morphing applied to the worm body is highly effective.

The n-gram similarity between worm generations can also be visualized graphically as discussed in Sect. 2.2. The similarity between a typical first and second generation worm is illustrated in the graph in Fig. 4. Examples of similarity graphs for other pairs of worms can be found in [14].

The n-gram similarity between worms is somewhat lower than the similarity between benign files. This is due to the fact that only the worm body is considered in our similarity tests, rather than the whole worm. The initial sections of the benign files result in a higher similarity between benign files, since there is considerable commonality in the initial sections of most executables.

4.3 Opcode graph similarity

In this section, we consider the opcode graph similarity technique discussed in Sect. 2.2. We applied this similarity measure to complete worm executable files, including padding blocks. We also computed the similarity of pairs of benign files, and worm files versus benign files. Figure 5 shows the similarity between these various combinations of worm and benign files.

Table 6 summarizes similarity scores between consecutive generations of worms, and the similarity scores between worms and benign files, and, finally, between pairs of benign files. These results indicate that the morphing of the worm—including the padding blocks—is highly effective.

4.4 HMM-based detection

We now analyze the results of testing an HMM detector on our metamorphic worms. Previous research has shown that the number of states in the HMM does not significantly impact its accuracy [13,28]. Consequently, here we only consider HMMs with $N = 2$ hidden states; additional results can be found in [14].

We define the ratio of dead-code to worm-code as the "padding ratio." For example, a worm with twice as much dead code as worm instructions will have a padding ratio of 2. We used an HMM classifier to score worms with padding ratios of 0.5, 1, 1.5, 2, 2.5, 3, and 4. The goal here is to approximately determine the minimum padding ratio at which the HMM detector starts to falter.

Figure 6 shows the results of scoring worms versus benign files, for padding ratios varying between 0.5 and 2.5. The scores are given as log likelihood per opcode (LLPO), that is, the scores are normalized to account for files of different length.

In Fig. 6(a), a padding ratio of 0.5 was used, which implies that the generated worms contain half as much dead code as the number of instructions that constitute its core functionality. We see that for this padding ratio, the detector can effectively distinguish our metamorphic viruses from benign files. However, for the higher padding ratios in Fig. 6, the LLPO scores of the worms overlap with those of benign binaries, to the point that detection would be impractical.

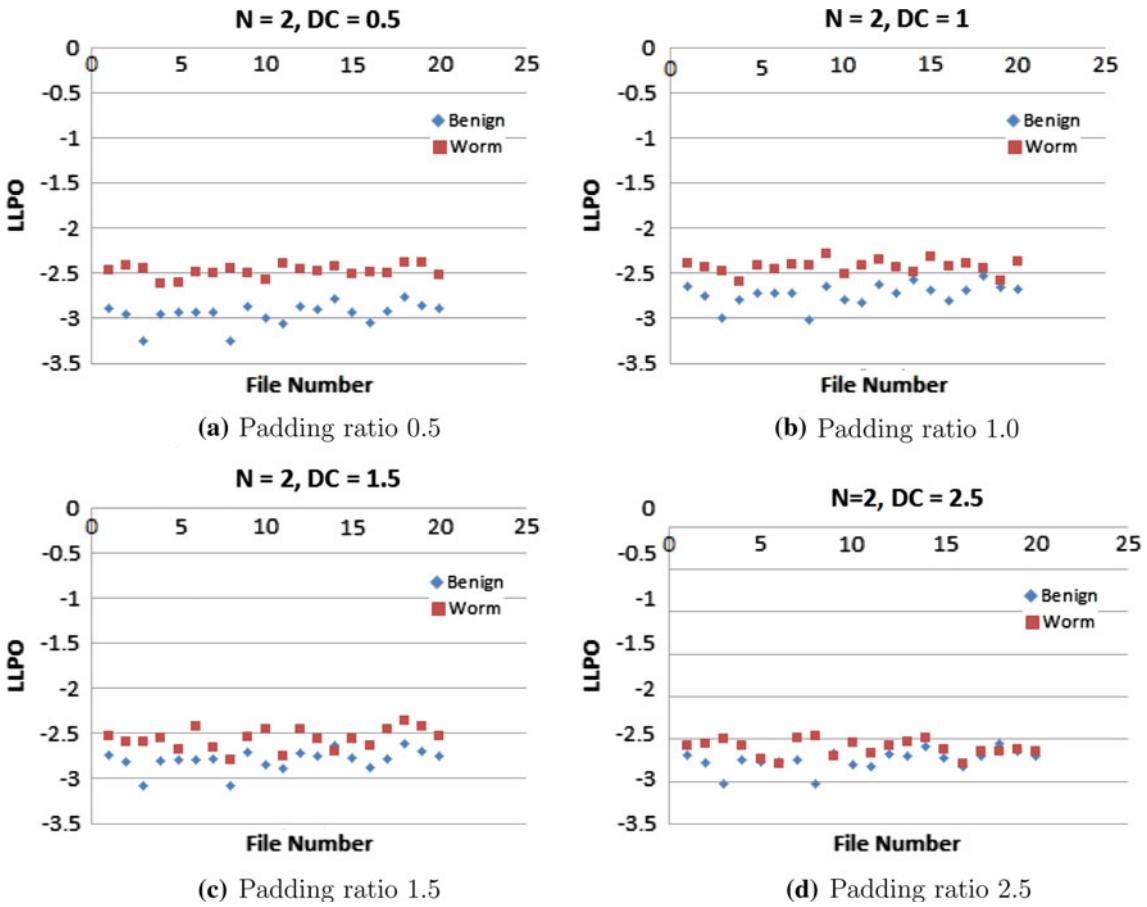

Fig. 6 HMM classifiers for various padding ratios

Our detection results for various padding ratios are summarized in the ROC curves in Fig. 7. For an ROC curve, the area under the curve (AUC) is equal to the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one [8]. Therefore, an AUC of 1 implies perfect classification, while an AUC of 0.5 implies that the classification is no better than flipping a coin. The AUC for each of the curves in Fig. 7 is

Fig. 7 ROC curves

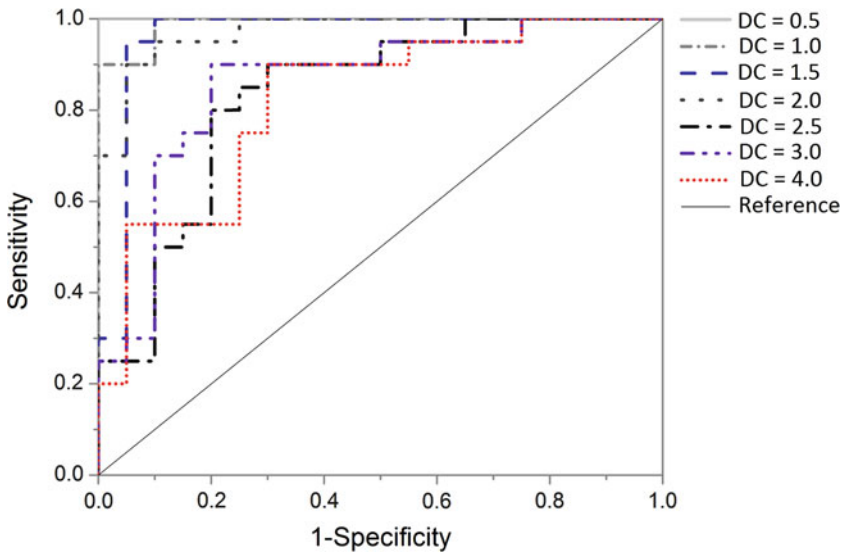

Table 7 AUC at various padding ratios

Padding ratio	AUC
0.5	1.0000
1.0	0.9900
1.5	0.9625
2.0	0.9725
2.5	0.8325
3.0	0.8575
4.0	0.8225

given in Table 7. Note that for a padding ratio of 2.5, the area under the curve is 0.8325 and, consequently, at this padding level, the HMM detector has an impractically high level of misclassification.

5 Conclusion and future work

The metamorphic worms considered in this paper carry their own morphing engine. The worms also include strong defenses against static detection techniques. We presented experimental results showing that for sufficiently high padding ratios, these worms cannot be reliably detected using signature-based methods. Furthermore, the worms cannot be detected using a static technique based on HMMs. The HMM detector performance is acceptable for padding ratios up to about a factor of 2. However, the probability of misclassification is clearly unacceptable for padding-ratios greater than 2.0.

We measured the similarity of the worms using an n-gram technique and an opcode graph technique. The n-gram similarity was shown to be sufficiently low so as to prevent the extraction of a common signature. The average similarity scores measured using the opcode graph technique are comparable to the similarity scores for randomly selected benign files. This is another strong indicator that signature-based detection is futile. These results show that the worms cannot be reliably distinguished from benign files based on these similarity measures.

The worm presented in this paper is designed to serve as a practical tool for studying detection strategies for advanced metamorphic malware. Techniques aimed at identifying possible dead code could prove useful for detecting the worm presented here. Although dead code can be disguised to the point where automatic identification is impractical, the techniques used in this research—as with the similar techniques used by metamorphic malware in the wild—are fairly elementary. Consequently, statistical analysis may prove useful for identifying regions of code that are of interest. For example, in [24], Kullback–Leibler divergence was used to enhance a masquerade detection system by statistically separating the attack from other activities. A similar principle might be of practical use when applied to metamorphic malware.

References

1. Anderson, B., et al.: Graph-based malware detection using dynamic analysis. J. Comput. Virol. **7**(4), 247–258 (2011)
2. Attaluri, S., McGhee, S., Stamp, M.: Profile hidden Markov models and metamorphic virus detection. J. Comput. Virol. **5**(2), 151–169 (2009)
3. Aycock, J.: Computer Viruses and Malware (Advances in Information Security). Springer, Berlin (2006)
4. Beaucamps, P.: Advanced metamorphic techniques in computer viruses. In: International Conference on Computer, Electrical, and Systems Science, and Engineering–CESSE '07, Venice, Italy (2007)
5. Bilar, D.: On callgraphs and generative mechanisms. J. Comput. Virol. **3**(4), 285–297 (2007)
6. Bilar, D.: On callgraphs and generative mechanisms, erratum. J. Comput. Virol. **3**(4), 299–310 (2007)
7. Borello, J., Me, L.: Code obfuscation techniques for metamorphic viruses. J. Comput. Virol. **4**(3), 211–220 (2008)
8. Bradley, A.P.: The use of the area under the roc curve in the evaluation of machine learning algorithms. Pattern Recognit. **30**, 1145–1159 (1997)
9. Desai, P.: Towards an undetectable computer virus (2008). Master's Projects. Paper 90. http://scholarworks.sjsu.edu/etd_projects/90
10. Filiol, E.: Metamorphism, formal grammars and undecidable code mutation. Int. J. Comput. Sci. **2**, 70–75 (2007)
11. Konstantinou, E., Wolthusen, S.: Metamorphic virus: analysis and detection. Technical Report RHUL-MA-2008-02, Department of Mathematics, Royal Holloway, University of London (2008)
12. Lin, D.: Hunting for undetectable metamorphic viruses. Master's Projects. Paper 18 (2009). http://scholarworks.sjsu.edu/etd_projects/18
13. Lin, D., Stamp, M.: Hunting for undetectable metamorphic viruses. J. Comput. Virol. **7**(3), 201–214 (2011)
14. Madenur Sridhara, S.: Metamorphic worm that carries its own morphing engine (2012). Master's Projects. Paper 240. http://scholarworks.sjsu.edu/etd_projects/240
15. The Mental Driller: Metamorphism in practice or "How I made MetaPHOR and what I've learnt" (2002). http://biblio.l0t3k.net/magazine/en/29a/
16. Miller, F., Vandome, A.: Gnu Binutils. Alphascript Publishing (2010)
17. Mishra, P.: Taxonomy of uniqueness transformations. Master's Report, Department of Computer Science, San Jose State University (2003). http://www.cs.sjsu.edu/faculty/stamp/students/FinalReport.doc
18. Opdis. libopcodes-based disassembler (2010). http://mkfs.github.com/content/opdis/
19. Orr, The molecular virology of Lexotan32: Metamorphism illustrated (2007). http://www.openrce.org/articles/full_view/29
20. Runwal, N., Low, R.M., Stamp, M.: Opcode graph similarity and metamorphic detection. J. Comput. Virol. **8**(1–2), 37–52 (2012)
21. Snakebyte. Next Generation Virus Construction Kit (NGVCK) (2000). http://vx.netlux.org/vx.php?id=tn02
22. Stamp, M.: Information Security: Principles and Practice. Wiley, New York (2011)
23. Stamp, M.: A revealing introduction to hidden markov models (2012). http://www.cs.sjsu.edu/stamp/RUA/HMM.pdf

24. Tapiador, J., Clark, J.: Masquerade mimicry attack detection: a randomised approach. J. Comput. Virol. **30**(5), 297–310 (2011)
25. Venkatachalam, S.: Detecting undetectable computer viruses. Master's Projects. Paper 156 (2010). http://scholarworks.sjsu.edu/etd_projects/156
26. Venkatesan, A.: Code obfuscation and virus detection. Master's Projects. Paper 116 (2008). http://scholarworks.sjsu.edu/etd_projects/116
27. Wong, W.: Analysis and detection of metamorphic computer viruses. Master's Projects. Paper 153 (2006). http://scholarworks.sjsu.edu/etd_projects/153
28. Wong, W., Stamp, M.: Hunting for metamorphic engines. J. Comput. Virol. **2**(3), 211–229 (2006)
29. Zbitskiy, P.: Code mutation techniques by means of formal grammars and automatons. J. Comput. Virol. **5**(3), 199–207 (2009)