J Comput Virol (2011) 7:201–214
DOI 10.1007/s11416-010-0148-y

ORIGINAL PAPER

Hunting for undetectable metamorphic viruses

Da Lin · Mark Stamp

Received: 21 May 2010 / Accepted: 29 November 2010 / Published online: 25 December 2010
© Springer-Verlag France 2010

Abstract Commercial anti-virus scanners are generally signature based, that is, they scan for known patterns to determine whether a file is infected. To evade signature-based detection, virus writers have employed code obfuscation techniques to create metamorphic viruses. Metamorphic viruses change their internal structure from generation to generation, which can provide an effective defense against signature-based detection. To combat metamorphic viruses, detection tools based on statistical analysis have been studied. A tool that employs hidden Markov models (HMMs) was previously developed and the results are encouraging—it has been shown that metamorphic viruses created by a reasonably strong metamorphic engine can be detected using an HMM. In this paper, we explore whether there are any exploitable weaknesses in an HMM-based detection approach. We create a highly metamorphic virus-generating tool designed specifically to evade HMM-based detection. We then test our engine, showing that we can generate metamorphic copies that cannot be detected using existing HMM-based detection techniques.

1 Introduction

A virus is designed to infect and potentially damage a computer system. Viruses are generally parasitic, in the sense that they attach themselves to executable files that are part of legitimate programs [2]. When an infected program is launched, the embedded virus is also executed and may replicate itself to infect other files and programs. In contrast, a worm is often defined as malware that is self-replicating, but not parasitic, and propagates over a network [2]. Here we use the term "virus" generically.

Some viruses may perform damaging activities on the host machine, such as corrupting data. Other viruses are relatively harmless and might, for example, print annoying messages on the screen. In any case, viruses are undesirable for computer users. Modern viruses take advantage of the Internet to spread on a global scale. Therefore, early and effective detection is necessary to minimize potential damage.

There are many antivirus defense mechanisms in use today, but the most widely used strategy is signature detection, which scans for viruses by searching for predetermined binary strings or other patterns [18]. Another popular mechanism for virus detection is code emulation, which creates a virtual machine to execute suspicious programs and monitor for unusual activity.

To evade signature detection, virus writers have adopted various strategies, including code obfuscation techniques to create metamorphic computer viruses. Metamorphic viruses change their internal structure from generation to generation. Consequently, signature-based scanners cannot reliably detect well-designed metamorphic malware.

To combat metamorphic viruses, detection tools based on statistical analysis have been previously studied. A tool based on hidden Markov models (HMMs) was developed in [22], and the results are encouraging—it has been demonstrated that metamorphic viruses created by a reasonably well-designed hacker-produced metamorphic engine can be detected using HMMs.

The goal of this work is to develop a standalone metamorphic engine that will defeat the HMM-based detection tool developed in [22]. We employ elementary code obfuscation techniques to generate highly metamorphic copies of a given

D. Lin
Cisco Systems, Inc., San Jose, CA, USA

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

base virus. In addition, each viral copy is made similar to a randomly selected "normal" file. These morphed copies have been tested against commercial virus scanners and the HMM detection tool in [22].

Intuitively, it is clear that it is possible to defeat commonly used virus detection schemes. This can be made rigorous in some cases. For example, in [10] it is shown that detection schemes that rely on "spectral analysis" can be defeated, where the "spectrum" consists of a listing of instructions from the suspect code. The idea is simply to modify viral code until the spectrum is statistically indistinguishable from normal code. The HMM-based detector considered here is based on opcode sequences, so it is not surprising that it can be defeated. However, we show that a straightforward spectral modification is not sufficient to break the HMM detector.

The techniques we employ to evade detection may also be of independent interest. Our method is relatively simple and the metamorphic generator we have constructed is quite strong, in spite of the fact that it only uses elementary morphing techniques. In addition, the resulting morphed viruses could be made considerably more difficult to detect by relatively simple modifications to our generator. We have more to say about these topics below.

This paper is organized as follows. In Sect. 2, we provide background information on computer viruses and discuss some common defenses. Section 3 describes a similarity test that is useful for measuring the degree of variability between files. Also in Sect. 3, we provide a brief discussion of HMMs and their potential role in virus detection. Section 4 gives the design and implementation of our metamorphic generator. Then in Sect. 5 we discuss our experiments and give our results, while Sect. 6 contains a statistical analysis related to these empirical results. Finally, Sect. 7 presents our conclusions and future work.

2 Antivirus and metamorphic viruses

2.1 Antivirus defense techniques

Techniques for generating viruses have evolved over time, primarily as a reaction to improved anti-virus techniques. In this section, we briefly outline some popular antivirus techniques.

2.1.1 Signature detection

Signature detection was the earliest antivirus technique and is still the most widely used approach. Ideally, a signature is a string of bits (that may include wildcards) found in a particular virus, but not in other executables [2]. During the scanning process, a signature-based virus detection tool will search the files on a system for known signatures. It will flag a file as possibly infected if a known virus signature is found. Secondary testing may be required to confirm an infection.

2.1.2 Heuristic analysis

Heuristic analysis is a method designed to detect previously unknown computer viruses, as well as new variants of existing malware [2]. One popular dynamic analysis method is to emulate execution of a questionable program or script and monitor for common viral activities, such as replication, overwriting executable files, etc. If such actions are detected, the suspicious program is flagged as a possible virus.

One method of static heuristic analysis is to disassemble the viral program, then analyze the code. This analysis might look for instructions that are commonly found in viral programs. If the source code contains a certain percentage of instructions that match common viral instructions, the file is flagged.

Heuristic analysis has a relatively high false positive rate. Consequently, signature detection is preferred, if it is applicable. However, for previously unseen malware, heuristic analysis is a viable option while signature detection is not.

2.2 Viruses

2.2.1 Virus obfuscation techniques

Virus-like programs first appeared on a wide scale in the 1980s [5]. Since then, the arms race between antivirus researchers and virus writers has continued unabated. To avoid detection by signature scanning, a virus can modify its code and alter its appearance on each infection. The techniques that have been employed to achieve this end range from *encryption*, to *polymorphism*, to *metamorphism*; see [2] for additional information on these approaches. Here, our focus is on metamorphic viruses.

2.2.2 Metamorphic viruses and metamorphic techniques

Metamorphism refers to code that changes its internal structure without changing its function [21]. Intuitively, different generations of a metamorphic virus have different "shapes" while maintaining the original behavior, that is, the internal structure varies, but the function remains the same.

Next, we briefly discuss some of elementary techniques employed by metamorphic virus writers. This list is not by any means exhaustive—more advanced techniques can be found, for example, in [12]. However, the techniques considered here are more than sufficient for our purposes.

2.2.2.1 Register swap Register swapping is one of the simplest metamorphic techniques. For example, "pop edx" might be replaced with "pop eax." The W95/Regswap virus is among the earliest metamorphic viruses and it primarily

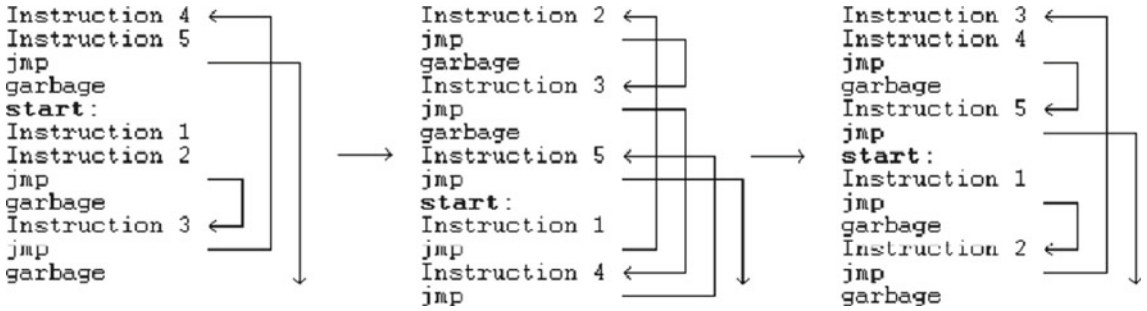

Fig. 1 Zperm virus [19]

used this technique. Note that with register swapping, the opcode sequence remains unchanged. Register swapping is a relatively weak form of metamorphism and viruses such as RegSwap can be detected by using wildcard strings in a standard signature scan [19].

2.2.2.2 Subroutine permutation If a virus has n subroutines, then we can generate n! distinct variants by simply permuting the subroutines. The W32/Ghost virus, which incorporates this technique, has 10 subroutines and, therefore, it can generate 10! = 3,628,800 unique copies. However, the virus can still be detected with search strings since the content of each subroutine remains constant [19].

2.2.2.3 Garbage insertion Many metamorphic viruses incorporate the technique of garbage instruction insertion. Garbage instructions are either not executed (dead code) or have no effect (do nothing) on the program outcome [6]. By inserting garbage instructions within useful code, a virus can generate an unlimited number of distinct copies while effectively breaking signatures.

Dead code insertion can be accomplished by including jump instructions to avoid the dead code. The Win95/Zperm virus is one virus that incorporates this technique, as illustrated in Fig. 1.

2.2.2.4 Instruction substitution Instruction substitution is the replacement of an instruction or a group of instructions with an equivalent instruction or group. For example, "inc eax" is equivalent to "add eax, 1" and as another example, "move eax, edx" can be replaced by "push edx" followed by "pop eax." In the context of metamorphic code, instruction substitution is discussed in [15,21].

2.2.2.5 Transposition Transposition is the reordering of the instruction execution sequence. This can only be done if the affected instructions have no dependencies. Consider the following generic example:

op1 r1, r2

op2 r3, r4; r1 and/or r3 are modified

1. mov R_1, len
2. mov R_2, beg
3. xor [R_2], key
4. add R_2, 4
5. sub R_1, 4
6. jnz step_3

Fig. 2 Decryptor code

```
00 PUSH 44554433                00 XOR EDI,EDI
01 POP ESI                      01 LEA EDI,[EDI+124]
02 SUB EBX,EBX                  02 PUSH 44554433
03 ADD EBX, 124                 03 POP ESI
04 XOR [ESI],d20b9a65           04 MOV EDX,[ESI]
05 ADD ESI,4                    05 NOT EDX
06 SUB EBX,4                    06 AND EDX,d75d40bc
07 JZ $ + 2                     07 AND [ESI],28a2bf43
08 JMP $ + f0                   08 OR [ESI],EDX
                                09 ADD ESI,4
                                10 SUB EDI,4
                                11 JZ $ + 2
                                12 JMP $ + e4
```

Fig. 3 Metamorphic versions of decryptor code

We can swap the above two instructions provided that r1 is not equal to r4, and r2 is not equal to r3, and r1 is not equal to r3; see [14] for more details.

2.2.3 Formal grammar mutation

Formal grammar mutation is the formalization of code mutation techniques by means of formal grammars and automatons. In general, classic metamorphic generators can be presented as non-deterministic automata, with all possible input characters specified for each state of the automata [12,23]. By formalizing existing code mutation techniques into formal grammars, formal grammar rules can be applied to create new viral copies with large variations. For example, this technique has been used to morph the simple decryptor code in Fig. 2 to create the variants in Fig. 3.

3 Similarity and hidden Markov models

This section outlines the similarity test and the HMM technique developed in [22]. In subsequent sections, we employ this similarity technique to quantify the degree of metamorphism produced by our generator, and we employ HMMs for detection.

3.1 Similarity test

Metamorphism is a practical approach to evading signature detection. However, for this to be effective, different generations of a virus must be sufficiently different so that no common signature is present. Consequently, we need a way to determine the effectiveness of a given metamorphic generator. We will apply a similarity index to measure the "closeness" of two assembly language programs. Then we can measure the effectiveness of our metamorphic generator by compare the similarity of viruses from a given family, where "family" refers to a set of viruses derived from the same metamorphic generator. We also apply this similarity measure to determine how much the morphed code differs from selected non-viral (i.e., "normal") code. In addition, we can compare samples of normal code to other normal code determine the expected similarity between non-viral programs.

Various similarity tests have been proposed. For example, three standard approaches to measuring similarity (edit distance, inverted index, and Bloom filters) are discussed in [11]. Another approach is pairwise alignment, which is commonly used to align sequences in bioinformatics applications [8]. Pairwise alignment is an edit distance that allows for insertions, deletions, and gaps to better align sequences. In [1], pairwise alignment is applied to opcode sequences extracted from metamorphic viruses. A sophisticated similarity measure based on the call graph and control flow graph is discussed in [4], while a similarity measure based on code behavior is given in [3].

We are focused static analysis, so comparing code behavior is not considered here. The similarity measure in [4] does not appear to have been applied specifically to metamorphic malware, so its performance is unclear. The remaining static approaches mentioned in the previous paragraph are generally not well suited to metamorphic malware. In metamorphism, code can be shuffled arbitrarily far apart. Dealing with basic blocks can help, but aligning the code is decidedly nontrivial, particularly in the presence of junk code and/or equivalent code substitution. For example, as demonstrated in [1], metamorphic generators with modest amounts of code shuffling result in weak pairwise alignment scores. Consequently, here we have opted to use the similarity test given in [22], which was originally developed in [15]. This score was designed to deal with mutated code, it is easy to compute, and it has a demonstrated track record of producing useful results when applied to metamorphic code. Next, we briefly outline the steps needed to compute this similarity index and we give an example that illustrates the process.

We compare two assembly programs and assign a score from 0 to 100%, where 0% implies no similarity, and 100% implies the programs are virtually identical. The idea behind the method is to look for overlapping opcode subsequences at any offset in the two programs. The score is then, essentially, the fraction of subsequences for which a match is found. Based on previous work, subsequences of length 3 are used, and we considered it a match if the 3 opcodes agree, irrespective of order. To reduce random noise, a threshold of five consecutive 3-opcode matches must be found before the result contributes to the score.

This score can be illustrated graphically, as shown in Fig. 4. Note that if a program is compared with itself, the main diagonal will be a solid line with some random matches appearing off the diagonal. Segments parallel to the main diagonal represent matches at some offset.

Previous work [22] has shown that of the hacker-produced metamorphic engines studied, the best achieves a similarity score of about 10%, that is, the family viruses are only about 10% similar to each other, on average. In contrast, a control set of normal files is shown to have an average similarity of about 30%. For more details on this similarity scoring technique, including numerous examples, see [15,22].

3.2 Hidden Markov models

A hidden Markov model (HMM) is a machine learning technique, which can also be viewed as a discrete hill climb [16,17]. Conceptually, in an HMM there is a Markov process that cannot be directly observed. However, we do have access to a series of observations (or emissions) that are related to the underlying Markov process. A generic HMM is illustrated in Fig. 5, where the X_i represent the hidden states and A is the (row stochastic) matrix that drives the hidden Markov process. In addition, the O_i are the observations and B is a (row stochastic) matrix that contains probability distributions relating the observations to the states of the Markov process. Finally, the dashed line in Fig. 5 indicates that we cannot directly observe the hidden Markov process.

HMMs are widely used in applications such as protein modeling and speech recognition [8,16]. An HMM model can be trained to match a set of data. The trained model can then be used to score data to determine its similarity to the training data. There are efficient algorithms for both the training and scoring phases. Another highly desirable feature of HMMs is that virtually no a priori assumptions are required—essentially, the only free parameter is the size of the square matrix A, and for most applications a 2×2 or 3×3 matrix will suffice. For more information on HMMs,

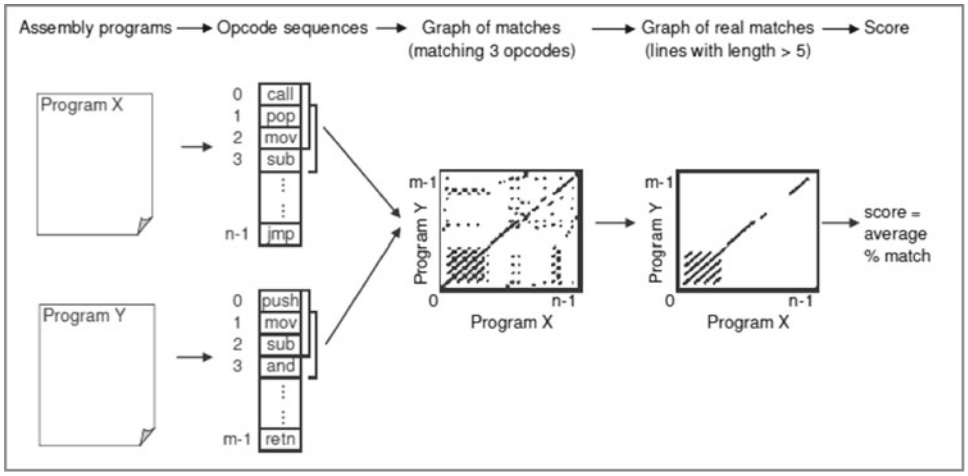

Fig. 4 Similarity between two assembly programs [22]

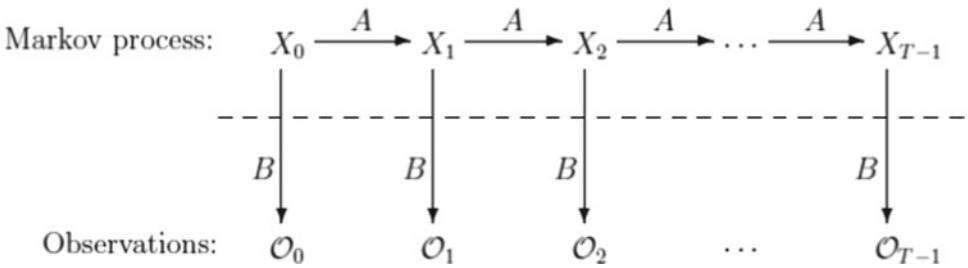

Fig. 5 Generic hidden Markov model [17]

including a motivating application and detailed pseudo-code, see [17].

Recently, HMMs have been successfully used to detect metamorphic viruses [20,22]. Although metamorphic engines use various code obfuscation techniques to change the appearance of their viral copies, some statistical similarities likely remain within a virus family. An HMM can be trained using opcode sequences extracted from known family viruses. Subsequently, any file can be scored using the model to determine its similarity to the virus family. Intuitively, HMMs seem particularly well suited for such analysis, since the actual opcode sequence can be viewed as a Markov process [10].

To train an HMM for virus detection, a set of viruses from the same metamorphic family are disassembled using IDA Pro [13]. We append these sequences to obtain one long opcode sequence, which constitutes the observations. An HMM model is then trained on this data. For example, given the training data in Fig. 6, part of the corresponding trained HMM model is shown in Fig. 7. Note that only the initial part of the training sequence appears in Fig. 6 and only the initial section of the B matrix appears in Fig. 7.

As mentioned above, after generating the HMM model, it is used to score files to determine the probability that they belong to the virus family from which the training data was obtained. If a file has a score above a certain threshold, then it is assumed to belong to the same family. The threshold can be determined experimentally.

4 Metamorphic generator

4.1 Introduction

To produce viral copies that are difficult to detect, a metamorphic engine must implement various code obfuscation techniques. Many of the metamorphic viruses tested in [22] could not be detected using commercial scanners, yet an HMM detector was able to correctly classify the family viruses and normal code with 100% accuracy. Our goal here is to create a metamorphic generator that evades signature detection and is also undetectable using the HMM-based approach in [22].

The paper [7] contains a highly metamorphic generator that was designed to evade HMM detection. However, the approach in [7] was largely unsuccessful. Our goal here is to improve upon this previous work.

4.2 Design

Our metamorphic generator is designed to achieve the following:

- Generate morphed copies of a single input virus. The morphed copies should have the same functionality as the base virus. Ideally, these morphed copies should have a similarity of less than 30%, when compared to the base virus and when compared to each other, as measured by the similarity index discussed in Sect. 3.1. Based on previous

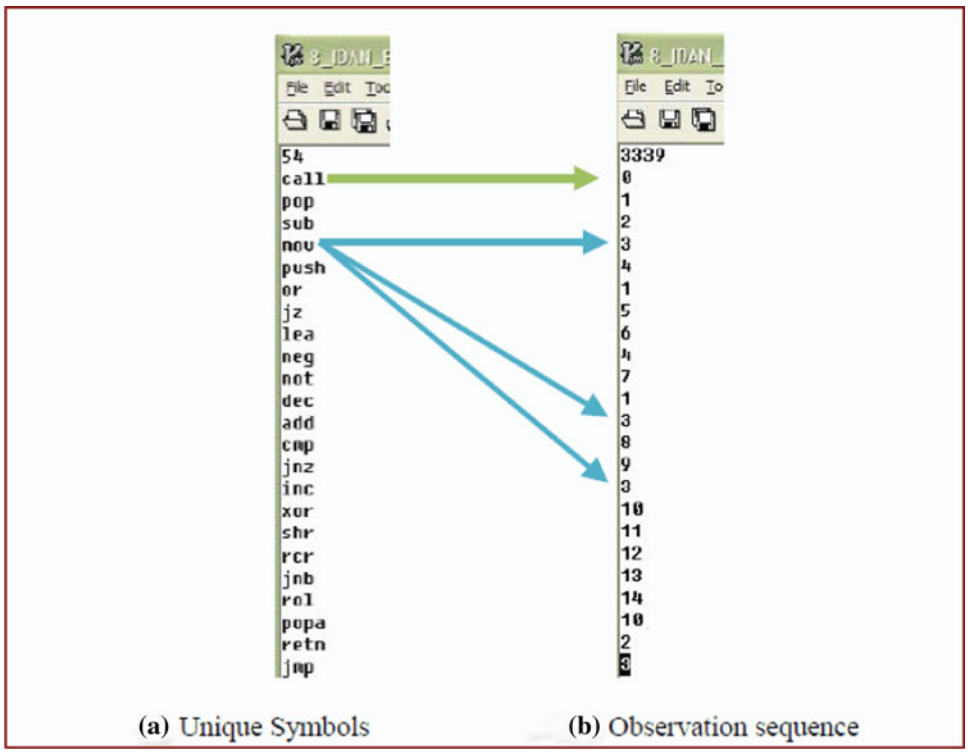

Fig. 6 Training data example [7]

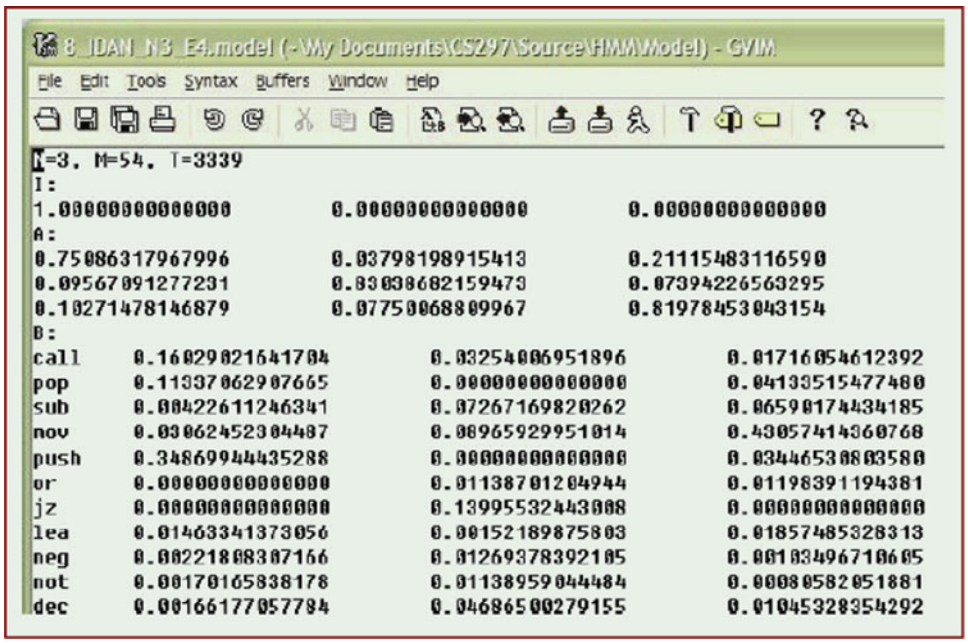

Fig. 7 HMM model example [7]

work, this level of similarity will enable the viruses to evade signature detection [7,22].

- The morphed copies should be "close" to normal programs. For the normal programs we use cygwin utility files of roughly the same size as the base virus. The reason for using these files is that they engaged in low-level operations, which should be somewhat similar to the activities of viruses. That is, these utility files should be relatively difficult to distinguish from the viruses. For efficiency, when morphing viruses, we measure closeness based on opcode mono-graph and di-graph frequency counts. Intuitively, if the morphed viruses are sufficiently close to the normal files, they should evade HMM detection. This concept of closeness can be formalized [10] and we have more to say about the topic in Sect. 6.

In short, our goal here is to generate viral copies that evade signature detection and also evade HMM detection. This is the precise sense in which our viruses are "undetectable."

4.3 Code obfuscation techniques

Our metamorphic engine leverages the code obfuscation techniques implemented in [7] with some important refinements. In our engine, we apply randomly selected code obfuscations only if they make the resulting virus closer to a normal program. We have developed a simple dynamic scoring algorithm for efficiently determining whether a potential modification makes the virus code closer to a given program.

4.3.1 Dynamic scoring algorithm

We have developed a simple dynamic scoring algorithm to calculate the closeness between two files—the lower the score, the closer the files. Since this algorithm will need to run each time we consider a change to an instruction, it must be efficient. Our algorithm does not compute the score from scratch each time. Instead, it modifies the score based on the actual modifications made. That is, we only need to compute deltas to update the score.

4.3.1.1 Algorithm initialization To initialize the dynamic scoring algorithm, two files are passed in as parameters, where the first one is a the file to be morphed (i.e., a virus), and the second one is a normal file. The algorithm initializes four master lists, namely, mono-graphic and di-graphic opcode counts for both the normal file and the virus file. For example, given the two short files with five opcodes each that appear in Table 1, the algorithm initialization yields the four lists shown in Table 2.

We compute the initial score by summing the differences of the mon-graph and di-graph counts. In the example above, the initial score is 8.

4.3.1.2 Updating the score To determine whether a change will yield a better score, we only need to compute the score change in terms of the opcode sequence changes, as discussed in detail in [14]. Here, we simply illustrate the process with an example. Suppose we transpose "add, mov" to "mov, add." Then we obtain the result given in Table 3. Since the new score of 5 is less than the previous score of 8, this is an improvement, and we make the change. That is, by making this change, we will make the virus file closer to the normal file.

Table 1 Opcode sequences

Virus opcode	Normal file opcode
Mov	Mov
Add	Mov
Mov	Sub
Pop	Popf
Retn	Retn

4.3.2 Code morphing and scoring

Opcode mono-graph and di-graph frequencies in a virus are generally statistically different than those in normal programs. Examples of such mono-graph statistics, which were analyzed in [7], appear in Figs. 8 and 9.

One morphing technique that we use is to insert dead code into the generated viruses. We insert dead code by copying blocks of instructions and subroutines directly from a selected normal program. Since the dynamically generated "dead code" is, in fact, actual code from a normal program, using this code will make our virus closer to the normal programs.

When we copy code from a normal file, we take blocks of five or more consecutive instructions, or complete subroutines. When we copy a block of consecutive instructions, we insert an unconditional jump instruction before the block so that these instructions will not be executed. While this could itself provide a heuristic for detection, it would be easy to modify this approach to make it stealthier by, for example, using opaque predicates [15].

Table 2 Opcode mono-graph and di-graph counts

Virus opcode mono-graphs	Normal file mono-graphs	Difference	Virus opcode di-graphs	Normal opcode di-graphs	Difference
Mov (2)	Mov (2)	0	Mov_add (1)	Mov_add (0)	1
Add (1)	Add (0)	1	Add_mov (1)	Add_mov (0)	1
Pop (1)	Pop (1)	0	Mov_pop (1)	Mov_pop (0)	1
Retn (1)	Retn (1)	0	Pop_retn (1)	Pop_retn (1)	0
Sub (0)	Sub (1)	1	Mov_mov (0)	Mov_mov (1)	1
			Mov_sub (0)	Mov_sub (1)	1
			Sub_pop (0)	Sub_pop (1)	1

Table 3 New score after changes

New virus opcode count list	Normal file opcode count list	Difference after changes	New virus opcode sequence count list	Normal file opcode sequence count list	Difference after changes
Mov (2)	Mov (2)	0	Mov_add (1)	Mov_add (0)	1
Add (1)	Add (0)	1	Add_mov (0)	Add_mov (0)	0
Pop (1)	Pop (1)	0	Mov_pop (0)	Mov_pop (0)	0
			Mov_mov (1)	Mov_mov (1)	0
			Add_pop (1)	**Add_pop (0)**	1

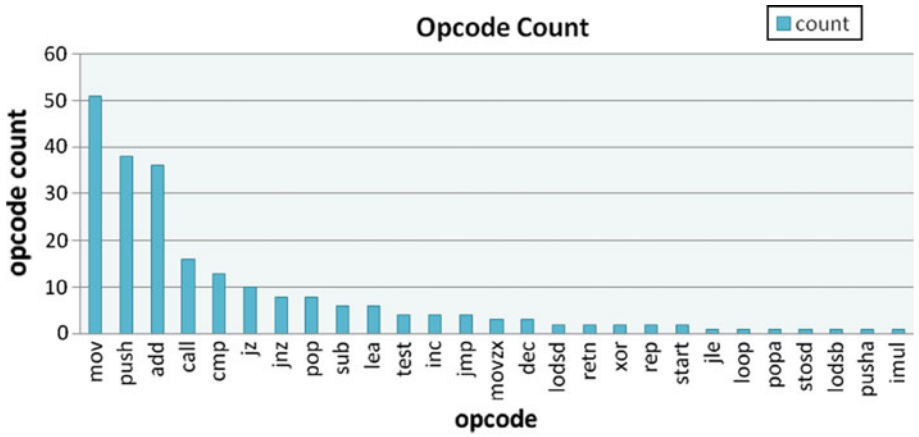

Fig. 8 Virus opcode mono-graph counts [7]

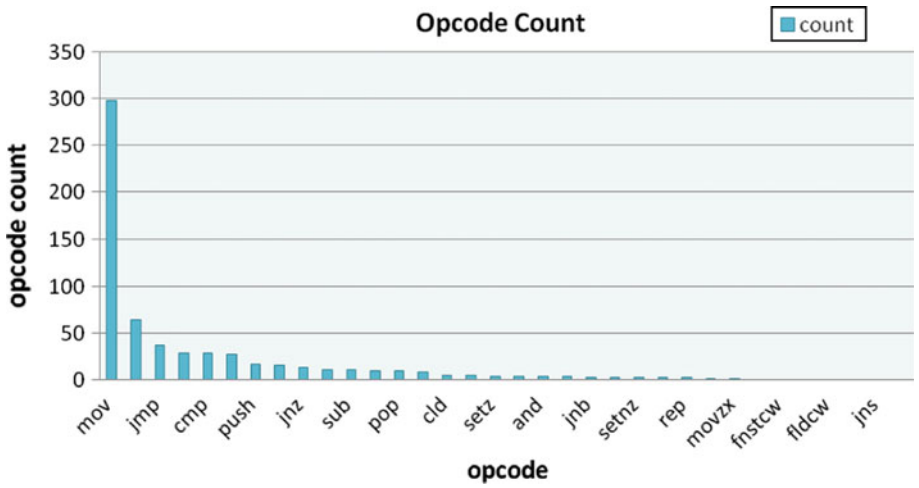

Fig. 9 Normal file opcode mono-graph counts [7]

In addition to inserting a jump instruction, in some cases we also need to modify the operands of instructions so that the generated virus file can be assembled correctly [9]. For example, if an instruction contains a label that is only valid in the normal file, then we will need to replace that label with one that is valid in the virus file.

When copying a subroutine, we also need to modify the operands of some instructions, but we do not need to insert jump instructions. A copied subroutine is placed between two existing subroutines in the virus file. Since the copied subroutine never gets called, it will not impact the behavior of the virus. Again, in our implementation this dead code is not stealthy, but it would be straightforward to make it much more so.

Our metamorphic generator also employs equivalent code substitution, but only when it makes the virus opcode frequency counts closer to that of a normal program. When considering equivalent substitutions, we also use the dynamic scoring algorithm discussed above to measure closeness.

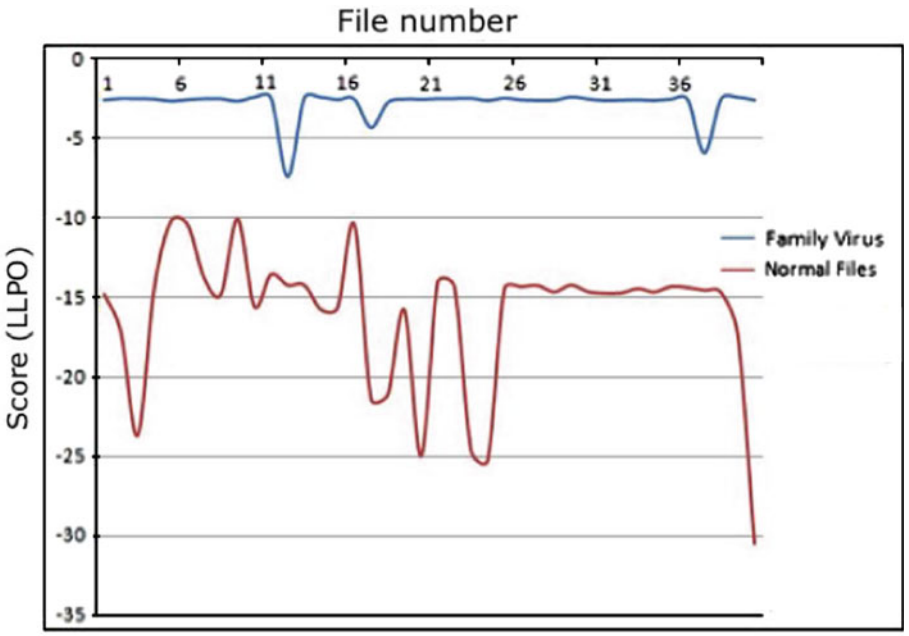

Fig. 10 HMM results for base viruses generated by NGVCK

After generating dead code and performing equivalent instruction substitution, we perform transposition. Again, the dynamic scoring algorithm is used, and we only make changes that cause the virus to be closer to the normal program.

5 Experiments

We used the similarity algorithm and HMM detection tools developed in [22] to test our metamorphic generator. First, we demonstrate that our engine is able to evade HMM-based detection. Then we repeat our test with different settings to esitmate the threshold at which the HMM detector begins to fail.

5.1 Base virus

To test our engine, we used the Next Generation Virus Construction Kit (NGVCK) to generate 200 virus files. These 200 files serve as our base viruses. We then obtained 40 normal files consisting of cygwin utility files. Note that this is the same procedure followed in [22].

After we generated our base viruses and normal files, we used the HMM detector to verify that viruses generated by NGVCK were still detectable. We first generated an HMM model using 160 viruses. We then scored the remaining 40 viruses against the HMM model and we also computed scores for the 40 normal files against the same HMM model.

If none of the normal files score higher than the viruses, then for an appropriate threshold, the HMM detector will detect the family viruses without fail. On the other hand, if some normal files score higher than some of the virus files, then the HMM detector will not provide a clear threshold for determining whether a given file is a virus or not. In this latter case, HMM-based detection will fail, at least for some percentage of the files.

Figure 10 shows the result of scoring NGVCK viruses against normal files. The scores are given in the form of log likelihood per opcode (LLPO), that is, log odds are computed and then normalized to a per opcode score. Note that in Fig. 10, all of the normal files score lower than virus files. Therefore, the base viruses we generated from NGVCK are detectable by the HMM detector. This serves as a confirmation of the relevant work presented in [22].

Next, we used our metamorphic generator to morph the base viruses. That is, we used our engine to perform additional code obfuscation on the NGVCK viruses. As discussed above, our engine takes one base virus and one normal file as inputs and it applies code obfuscation techniques to the base virus so that the resulting virus is statistically closer to the normal file. Before testing the HMM detector on our morphed viruses, we consider their similarity to normal files. For similarity comparisons of NGVCK files with each other, see [22].

5.2 Similarity test

We compared the similarity of our metamorphic viruses using the similarity measure discussed in Sect. 3. Since our engine is designed to make virus files closer to normal files, we compared the similarity of a virus file with its peer normal file. We computed the similarity multiple times as we increase the percent of dead code copied from the normal file.

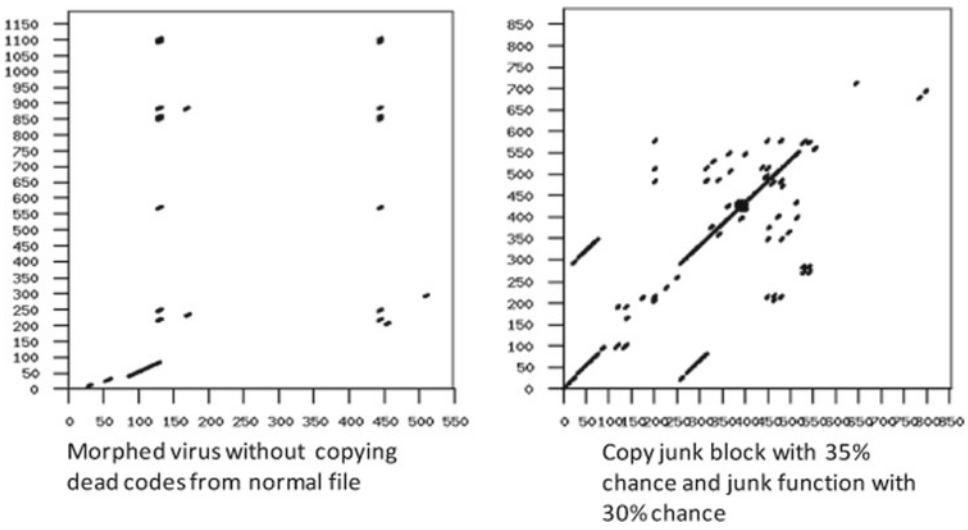

Fig. 11 Similarity graphs for morphed virus versus normal files

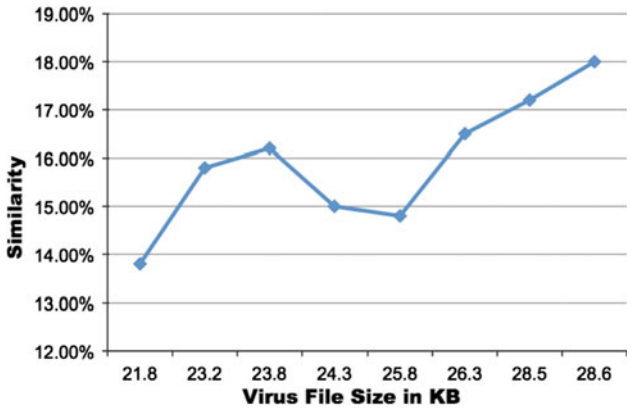

Fig. 12 Similarity score of virus and file size

We first compared a base virus against a normal file, and there was essentially no similarity between them. Then we ran the two files through our engine without any dead code copying and we obtained a similarity score of 13.8% between the two files. A typical similarity graph for this case is given in Fig. 11.

Then we copied dead code from the normal file into the virus and computed the similarity score again. As expected, the more dead code we copied into the virus, the higher the similarity score; see Fig. 12. However, for the range of values given in Fig. 12, the similarity does not increase dramatically. To increase the similarity to higher levels would require insertion of relatively large amounts of dead code and would significantly increase the file size.

5.3 HMM-based detection

We performed HMM tests for the virus sets that we generated. The goal here is to empirically determine how much dead code we need to copy from normal files to make our viruses undetectable (with respect to the HMM-based approach).

We conducted tests on our viruses with the number of HMM hidden states ranging from 2 to 5. However, based on the previous work [7,22], and confirmed by our results, it appears that the number of hidden states does not significantly affect the results. Therefore, we focus on analyzing the results for the HMM tests with three hidden states; see [14] for tests involving other numbers of hidden states.

5.3.1 Zero percent dead code

A set of viruses was generated using our engine without any dead code copied from the normal files. With this setting, the average virus file size increase from 17 to 21.8 kB. The similarity score also increase from essentially 0 to 13.8%. From the results in Fig. 13, we see that HMM-based detection is still possible in this case.

5.3.2 Copying dead code from normal file

This next set of viruses we consider were generated by applying our engine with the probability of copying dead code blocks set to 35%. This probability is applied after each instruction, that is, 35% of the time a block of dead code is inserted after an instruction. Also, the block of dead code ranged from 3 to 5 instructions. With this setting, the average file size increased from 17 to 24.3 kB. The HMM detection results are shown in Fig. 14.

In spite of these high settings for dead code insertion, the HMM was able to distinguish the family virus. These strong detection results are, perhaps, somewhat surprising given that the dead code was carefully selected so as to increase the similarity between the virus and normal files.

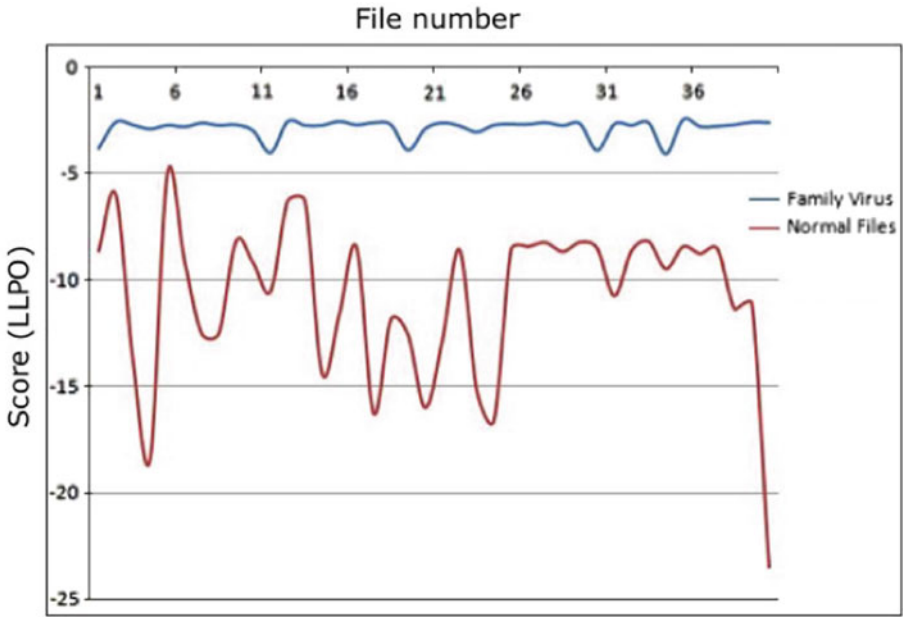

Fig. 13 HMM result with 0% dead code copied

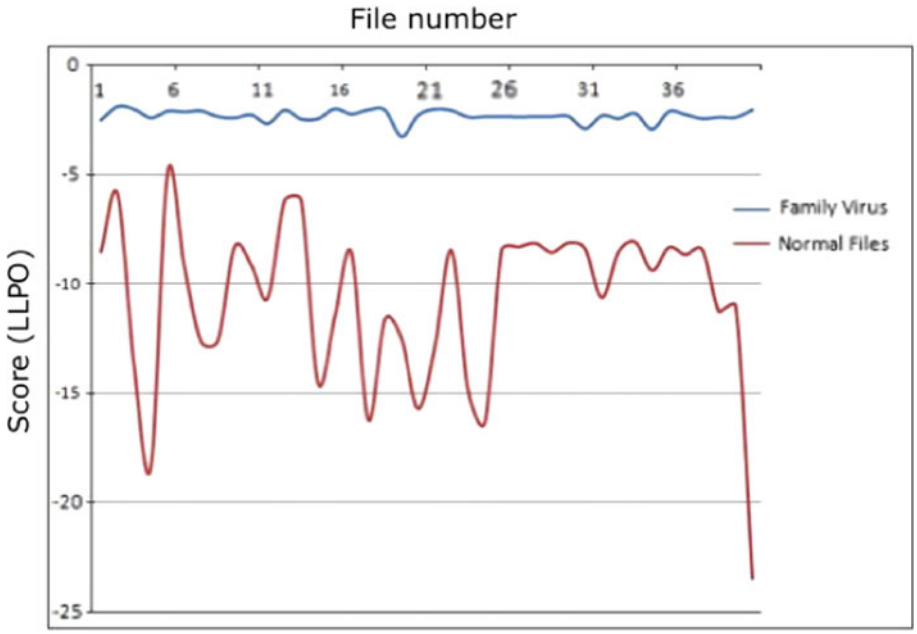

Fig. 14 HMM results with 35% dead code setting

5.3.3 Copying subroutines from normal file

We continued our experiments by copying subroutines from the normal files into our viruses. We configured the subroutine copying probability to 5% along with a 35% probability of copying dead code (as discussed above). The subroutine copying probability refers to the likelihood of copying an entire subroutine following any given subroutine in the base virus. Figure 15 gives the scores of family viruses and normal files using the HMM model with this 5% subroutine copying probability. Even with such a low setting, the HMM detection fails to accurately classify the viruses—16 viruses score lower than the maximum normal file score. This level of failure would make the HMM entirely impractical.

5.3.4 Copying subroutines only from normal file

Based on the results in Fig. 15, we observe that copying subroutines from normal files significantly impacts our scores. Therefore, we conducted additional experiments copying only subroutines into our base viruses without any additional code obfuscation. These results showed that even with as little as a 5% threshold for copying subroutines from the normal files—and no dead code copying—the HMM

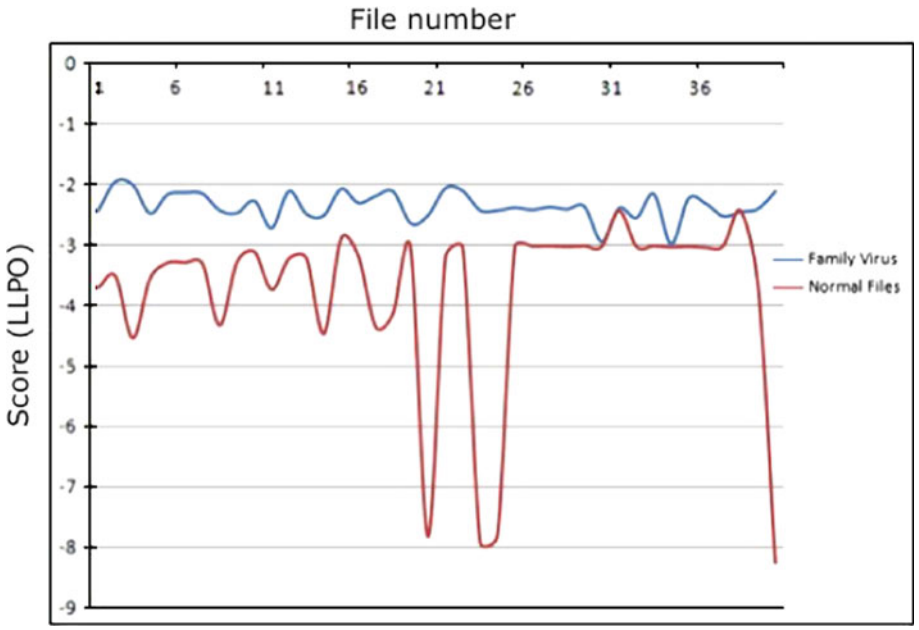

Fig. 15 HMM detection with 35% dead code and 5% subroutine copy

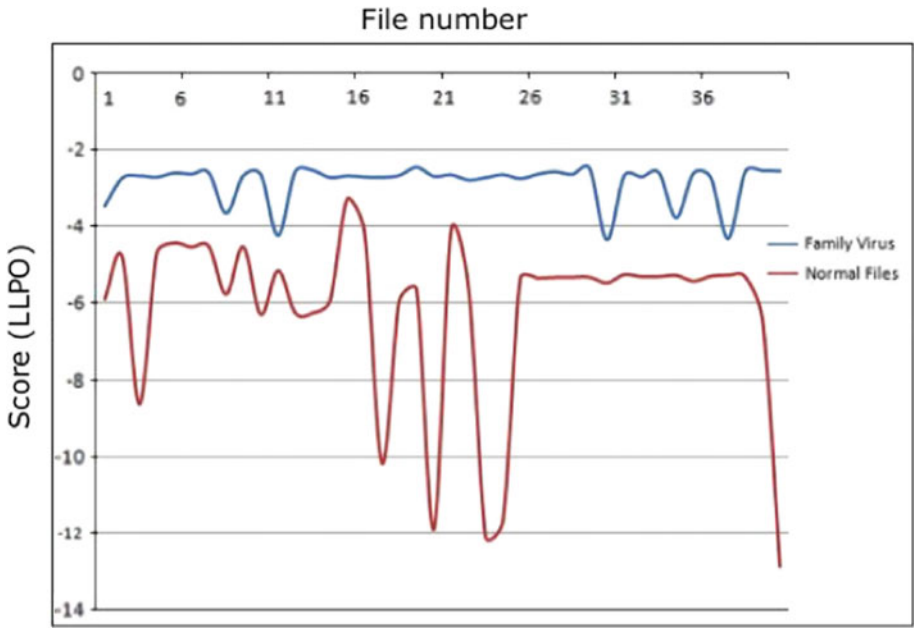

Fig. 16 HMM results with 5% subroutine copied

detector misclassified some of our viruses; see Fig. 16. In this case, the HMM detector would be ineffective as a detection mechanism. Additional experimental results can be found in [14].

6 Statistical analysis

Next, we give a brief statistical analysis of the results of the previous section. These results shed some additional light on our empirical results. The discussion here was motivated by the results in [10].

Suppose that c distinct opcodes appear in normal files. Let n_i, for i = 1,2,…,c, be the relative frequency of opcode i in these normal files and let m_i, for i = 1,2,…,c, be the relative frequency of opcode i in one of our morphed file. Define the null hypothesis to be that opcode frequencies of the morphed file are indistinguishable from that of a normal file, that is,

$$H_0 : m_i = n_i \quad \text{for } i = 1, 2, \ldots, c$$

Table 4 Frequency count statistical analysis

Case	D^2	$P(\chi^2 < D^2)$	Detected with HMM?
Base NGVCK virus	23.77	0.035	Yes
0% dead code, 0% subroutine	1.52	0.000	Yes
35% dead code, 0% subroutine	1.52	0.000	Yes
35% dead code, 5% subroutine	1.52	0.000	No
5% subroutine (no other morphing)	20.67	0.010	No

The alternative hypothesis H_1 is that the frequencies are significantly different, in which case there may exist a simple heuristic for detection of the morphed file.

It is well-known that the statistic

$$D^2 = \sum_{i=1}^{c} (m_i - n_i)^2 / n_i$$

follows a chi-square distribution (asymptotically) with c-1 degrees of freedom, provided that H_0 holds [10]. The values of D^2 corresponding to typical morphed files from each of the various cases considered in the previous section appear in Table 4. For all results in Table 4, c = 39, so the probabilities, $P(\chi^2 < D^2)$, are for a chi-square distribution with 38 degrees of freedom. Also, the probabilities have been rounded to the nearest 1/1000th.

The second row in Table 4 indicates that even without copying any code directly from a normal file, the spectrum of one of our morphed files is statistically indistinguishable from that of a normal file. However, these viruses are easily detected by an HMM which implies that the HMM is superior to a simple spectral heuristic, at least in some cases. On the other hand, the final row in Table 4 indicates that the HMM is highly vulnerable to insertion of normal opcode sequences of sufficient length? these viruses appear vulnerable to detection by a simple heuristic, yet they evade our HMM-based detector. Of course, if we apply our other morphing techniques, the resulting viruses would not be detectable using such a heuristic, as can be seen in the "35% dead code, 5% subroutine" row of Table 4.

7 Conclusions and future work

By modifying viruses so that they are similar to normal files, we were able to make them undetectable using an HMM-based detector. However, a very specific type of similarity was required to achieve this result. Specifically, the HMM-based detector began to fail when we copied subroutines (at a low rate) from normal files, while copying small segments of code did not yield the same effect.

To evade signature detection, a metamorphic engine must generate highly metamorphic viruses, that is, the viruses must exhibit little similarity when compared to other viruses of the same family. However, in [22] it was demonstrated that a high degree of metamorphism is not sufficient to evade HMM-based detection, and it was conjectured that highly metamorphic viruses that are similar to normal code would evade such detection. The work presented here confirms this, while also providing a simple method to obtain the required level of similarity. It is somewhat surprising that such a small addition of carefully selected code can defeat the HMM-based detector, which, in many respects, had proven to be fairly robust.

Acknowledgments The authors thank the anonymous referees for their many constructive comments and suggestions, which greatly improved this paper.

References

1. Attaluri, S., McGhee, S., Stamp, M.: Profile hidden Markov models and metamorphic virus detection. J. Comput. Virol. **5**(2), 151–169 (2009)
2. Aycock, J.: Computer Viruses and Malware. Springer, Berlin (2006)
3. Bailey, M. et al.: Automated Classification and Analysis of Internet Malware, RAID 2007, LNCS 4637, pp. 178–197. Springer, Berlin (2007)
4. Caillat, B.A., Desnos, Erra, R.: BinThavro: Towards a Useful and Fast Tool for Goodware and Malware Analysis, ECIW (2010)
5. Cohen, F.: Computer viruses: theory and experiments. Comput. Secur. **6**(1), 22–35 (1987)
6. Daoud, E., Jebril, I.: Computer virus strategies and detection methods. Int. J. Open Problems Comput. Math. **1**(2). (2008). http://www.emis.de/journals/IJOPCM/files/IJOPCM(vol.1.2.3.S.08).pdf
7. Desai, P.: Towards an undetectable computer virus. Master's report, Departent of Computer Science, San Jose State University. (2008). http://www.cs.sjsu.edu/faculty/stamp/students/Desai_Priti.pdf
8. Durbin, R. et al.: Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, Cambridge (1999)
9. FASM. http://flatassembler.net/
10. Filiol, E., Josse, S.: A statistical model for undecidable viral detection. J. Comput. Virol. **3**(2), 65–74 (2007)
11. Gheorghescu, M.: An automated virus classification system. In: Virus Bulletin Conference (2005)
12. Gueguen, G., Filiol, E.: New threat grammars, IAWACS (2010)
13. IDA Pro. http://www.hex-rays.com/idapro/

14. Lin, D.: Hunting for undetectable metamorphic viruses. Master's report, Department of Computer Science, San Jose State University (2010)
15. Mishra, P., Stamp, M.: Software uniqueness: how and why. In: Dey, P.P., Amin, M.N., Gatton, T.M. (eds.): Proceedings of Conference on Computer Science and its Applications. San Diego, July 2003. http://www.cs.sjsu.edu/~stamp/cv/papers/iccsaPuneet.doc
16. Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. Proc. IEEE **77**(2) (1989)
17. Stamp, M.: A revealing introduction to hidden Markov models, January 2004. http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf
18. Stamp, M.: Information Security: Principles and Practice. Wiley, New York (2005)
19. Szor, P., Ferrie, P.: Hunting for Metamorphic. Symantec Press, Cupertino. (2005). http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf
20. Venkatachalam, S.: Detecting undetectable computer viruses. Master's report, Department of Computer Science, San Jose State University (2010). http://www.cs.sjsu.edu/faculty/stamp/students/venkatachalam_sujandharan.pdf
21. Walenstein, A., et al: The design space of metamorphic malware. In: Proceedings of the 2nd International Conference on Information Warfare, March 2007
22. Wong, W., Stamp, M.: Hunting for metamorphic engines. J. Comput. Virol. **2**(3), 211–229 (2006)
23. Zbitskiy, P.: Code mutation techniques by means of formal grammars and automatons. J. Comput. Virol. **5**(3), 199–207 (2009)