Detecting Undetectable Metamorphic Viruses

Sujandharan Venkatachalam[1] **and Mark Stamp**[1]
[1]Department of Computer Science, San Jose State University, San Jose, California, USA

Abstract—*Signature-based detection provides a relatively simple and efficient method for detecting known viruses. At present, most antivirus systems rely primarily on signature detection.*

Metamorphic viruses are potentially one of the most difficult types of viruses to detect. Such viruses change their internal structure, which provides an effective means of avoiding signature detection. Previous work has shown that a specific and straightforward metamorphic engine can generate viruses for which reliable detection using "static analysis" is NP-complete. In this paper, we implement this metamorphic engine and show that, as expected, popular antivirus scanners fail to detect the resulting viruses. Finally, we analyze these same viruses using a previously developed detection approach based on hidden Markov models (HMM). This HMM-based detector, which by most definitions would be considered a static approach, easily detects the viruses.

Keywords: malware, metamorphic, static analysis, hidden Markov models

1. Introduction

Since the advent of malware, virus creation techniques and detection methodologies have evolved in an ongoing "arms race" [3]. Virus writers want their handiwork to evade detection and since signature detection is the most popular, considerable effort has gone towards hiding or obfuscating signatures.

Metamorphic viruses rely on code morphing to prevent signature detection [12]. While metamorphism can effectively obfuscate signatures, the paper [16] shows that metamorphic viruses produced by the hacker community are generally not very effective, and of those tested, even the most highly metamorphic are detectable using machine learning techniquesÑspecifically, hidden Markov models (HMMs).

The authors of [4] claim to have obtained the following intriguing result:

> In particular, we prove that reliable static detection of a particular category of metamorphic viruses is an NP-complete problem. Then we empirically illustrate our result by constructing a practical obfuscator which could be used by metamorphic viruses in the future to evade detection.

Note that the authors of [4] appear to have the usual concept of "static analysis" in mind, as indicated by the following quote:

> Here, static analysis is conceived as a process whose goal is to extract the semantics of a given program without any help of code execution.

It is well known that, in theory, metamorphic virus writers have an insurmountable advantage [6], [10], [17]. However, it is a curious fact that relatively few metamorphics have appeared in the wild and, furthermore, very few of these provide strong metamorphism and, in any case, none has proven particularly difficult to deal with in practice. This suggests that there are many practical difficulties for virus writers to overcome if they want to take advantage of metamorphism. When viewed in this light, it might seem that the most impressive result in [4] is its claim to provide a simple and practical design for a metamorphic generator that yields viruses that cannot be reliably detected using "static analysis."

In this paper, we have implemented a stand-alone metamorphic generator that satisfies the conditions in [4] and we have applied it to selected viruses. We show that, as expected, the resulting morphed viruses are not detected using popular signature-based antivirus software. However, we also show that these metamorphic viruses are, in fact, easily detected using the machine learning technique developed in [16]. Note that the detection method in [16] would generally be considered a static approach, since it only relies on extracted opcode sequences—no code execution or emulation is used. Indeed, this would seem to fit the informal description of static analysis given in [4], as quoted above.

The work presented in this paper demonstrates that metamorphic viruses generated following [4] would not be particularly difficult to detect in practice, even if we restrict ourselves to static analysis, as the term is generally understood. The loophole here is that the formal definition of "static analysis" in [4] is extremely narrow—much narrower than suggested by the informal discussion in the paper itself.

The organization of this paper is as follows. In Section 2 we briefly discuss the evolution of metamorphic viruses. Then in Section 3 we consider elementary code obfuscation techniques used in metamorphic generators, and in Section 4 we briefly discuss the use of HMMs for metamorphic detection. In Section 5 we provide details of the metamorphic generator that we have developed—a generator that satisfies the conditions given in the paper [4]. Section 6 summarizes our experimental results. Finally, Section 7 concludes the paper.

2. Evolution of Metamorphic Viruses

Early on in the Titanic struggle between the forces of good and evil (code, that is), signatures became the preferred means of detecting malware. Predictably, virus writers reacted by developing new techniques designed to evade signature detection. Here, we briefly outline the evolution of virus development and the parallel history of virus detection.

As the name indicates, encrypted viruses try to bypass virus detection by self-encryption. The code encryption implemented in such viruses effectively hides the signature of the underlying virus. However, the virus body must be decrypted before it can execute, and the decryption code is susceptible to signature scanning [3].

Like encrypted viruses, polymorphics try to bypass detection by self-encryption. However, unlike encrypted viruses, these viruses mutate the decryptor code, making scanning much more challenging [12]. Figure 1 illustrates different variants of a polymorphic virus. Note that the decrypted body is the same in each case.

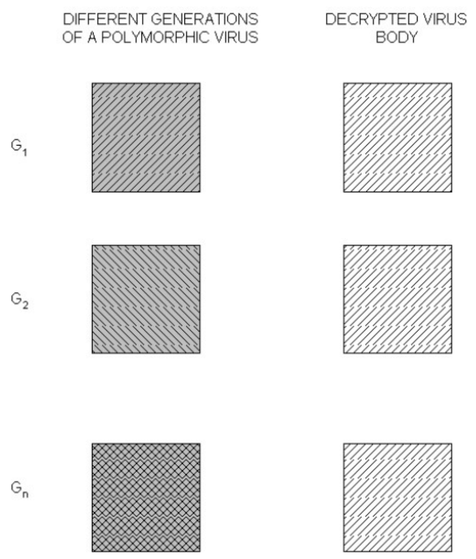


Fig. 1: Polymorphic viruses [12]

Polymorphic viruses are often detected using emulation—if the emulated code is actually a virus, it will eventually decrypt itself, at which point standard signature scanning can be applied [3].

Metamorphic viruses modify their entire code in such a way that each copy is functionally the same, but structurally different [3]. If the copies are sufficiently different, no common signature will be present. Figure 2 illustrates different generations of a metamorphic virus. Note that the code structure differs in each case, yet the viral copies all have the same function.

It is intuitively clear that well designed metamorphic code cannot be effectively detected via signature-based methods—a fact that is made rigorous in [4]. However, it has previously

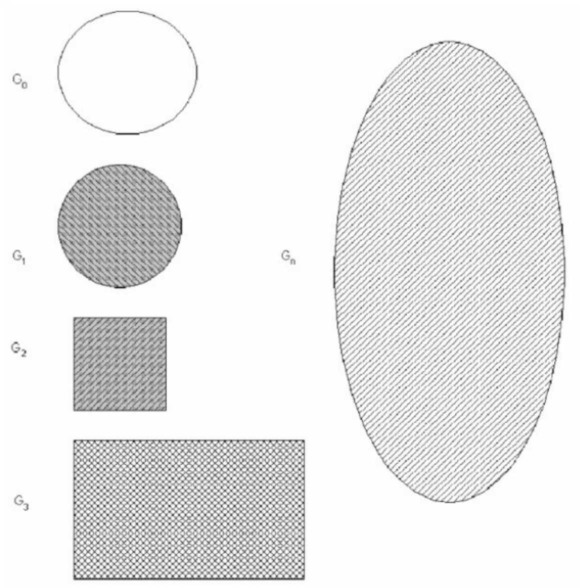


Fig. 2: Metamorphic viruses [12]

been shown that metamorphic viruses produced by the hacker community can be detected using machine-learning techniques [16]. Below, we have more to say about the approach used in [16].

3. Code Obfuscation

Metamorphic viruses use one or more obfuscation techniques to produce structurally different versions of the same virus, while not altering the function of the code. The primary goal of the obfuscation is to avoid signature detection—if the viruses are sufficiently different, no common signature will exist and, ideally, no fixed signature will detect any significant percentage of the viruses. Below, we briefly discuss a few of the most common code morphing techniques. The code obfuscation techniques implemented in several hacker-produced metamorphics are summarized in Table 1.

Table 1: Code obfuscation techniques [4]

	Evol 2000	Zmist 2001	Zperm 2000	Regswap 2000	MetaPHOR 2001
Substitution				X	
Permutation	X	X			X
Garbage code	X	X			X
Variable substitution	X	X		X	X
Alter control flow		X	X		X

Inserting garbage instructions between useful code blocks is a simple obfuscation technique used in all of the virus generators listed in Table 1. Garbage instructions do not alter the functionality but will tend to increase the size of the code. Viruses that contain garbage instructions are harder to detect using signatures since these instructions tend to break up predetermined signatures.

Instruction reordering is another common metamorphic technique. In this method, the instructions in the virus code are shuffled, with the control flow adjusted (via jump statements, for example) to make the code execute in the appropriate order. Thus, the instructions are reordered within the code without altering the actual control flow. This method can also effectively break signatures. However, if too many jump instructions are inserted, this could be used as a heuristic for detecting malware. Figure 3 shows an example of code reordering. Subroutine reordering is a special case of code reordering. Reordering subroutines in the virus does not change the control flow but could make signature detection more difficult.

Instruction interchange is another useful obfuscation technique. In this method, instructions are replaced with equivalent instructions. Then metamorphic versions of a given base virus will have different patterns of opcodes that perform the same function.

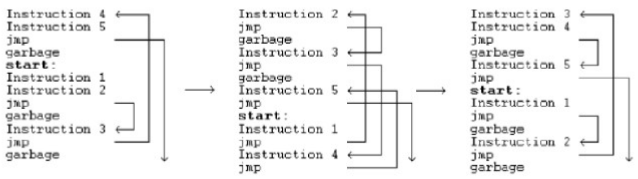


Fig. 3: Code reordering [13]

Register swapping, where different registers are used for equivalent operations, is a simple special case of interchanging instructions. Again, the idea is to change the opcode pattern and thereby bypass signature detection. This technique was the primary means of obfuscation used in one of the first metamorphic viruses, W95/Regswap. Register swapping is a particularly weak form of metamorphism, since it is subject to signature detection using wildcards [13].

Of the hacker-produced metamorphic generators tested in [16], the most advanced is the Next Generation Virus Construction Kit (NGVCK) [14]. Table 2 provides examples of code from NGVCK viruses.

4. HMMs for Virus Detection

Machine learning techniques have been successfully applied to the problem of detecting metamorphic viruses [16]. These techniques extract statistical information from training data and the resulting model can then be used to test any input file for similarity to the training data.

A hidden Markov model (HMM) is a state machine that attempts to model a Markov process. The Markov process is hidden, in the sense that it cannot be directly observed. The actual observations are probabilistically related to the hidden process. In the context of metamorphic viruses, an HMM is trained to detect a specific metamorphic family. The training data consists of a sequence of opcodes derived from viruses, all of which were produced by a single metamorphic engine. Once the model is trained, it can be used to score an unknown file, using an extracted opcode sequence, to determine its similarity to the metamorphic family. For more details on the use of HMMs for metamorphic virus detection, see [16]; for related work involving profile HMMs see [2]; for additional background on HMMs in general, see [11] or [9].

5. Metamorphic Generator

We have implemented a metamorphic virus generator that satisfies the conditions given in [4]. Recall that the paper [4] provides a rigorous proof that viruses generated using their approach cannot be efficiently detected using static analysis (as they define the term). Next, we briefly discuss the details of our metamorphic generator, which implements the practical generator given in [4].

A seed virus is input to our metamorphic generator. The seed virus assembly code is split into small blocks, which are then reordered using conditional jump instructions and labels. The number of instructions in each block is variable and for the experiments described here is set to an average value of six. The virus code is split into blocks, respecting the conditions given in [4], namely, code blocks cannot end with a label, jump, or a NOP instruction. A precondition on the seed virus is that the entire code must appear in the code section of the assembly file, which implies that viruses that hide code in their data section cannot be used.

After splitting the code into blocks, the blocks are randomly shuffled. Then labels are inserted and conditional jump instructions are used so as to maintain the original control flow. Optionally, garbage code insertion is applied for additional code obfuscation. In summary, our metamorphic engine performs the following steps:

1) Input a base virus file
2) Blocks are identified subject to the following conditions:
    a) The first and last block of the code are fixed
    b) The last instruction of a block is not a label, jump, or NOP
3) Blocks are randomly permuted and labels and conditional jumps are inserted
4) Garbage instructions are randomly inserted according to a threshold value
5) Write the morphed output file

The garbage insertion is optional and the amount of garbage inserted is adjustable. The garbage instructions include various copy instructions and opaque predicates, with the garbage inserted between pairs of code blocks, after the block shuffling is completed. Our generator has been successfully tested with several virus families. A typical test

Table 2: Code obfuscation in NGVCK

Base Version	Morphed Version 1	Morphed Version 2
call delta	call delta	add ecx, 0031751B ; junk
delta: pop ebp	delta: sub dword ptr[esp], offset delta	call delta
sub ebp, offset delta	pop eax	delta: sub dword ptr[esp], offset delta
	mov ebp, eax	sub ebx,00000909 ; junk
		mov edx, [esp]
		xchg ecx, eax ; junk
		add esp, 00000004
		and ecx, 00005E44 ; junk
		xchg edx, ebp
HEX equivalent:	HEX equivalent:	HEX equivalent:
`E8000000005D81ED05104000`	`E800000000812C2405104000588BE8`	`⋆812C240B104000⋆8B1424⋆83C404⋆87EA`

case is discussed in the next section; for more examples, see [15].

Next, we applied an HMM virus detection technique to our metamorphic viruses. Here, we mimic the training and scoring methodology used in [16]. To train an HMM model, 200 distinct metamorphic copies of a given seed virus were created using our metamorphic engine. The metamorphic engine generates ASM files, each of which yields executable code having the same functionality as the seed virus. These 200 files were assembled using the Borland Turbo TASM 5.0 assembler and linked using the Borland Turbo TLINK 7.1 linker to produce EXE files. The EXE files thus obtained were then disassembled using IDA Pro [7] and opcode sequences were extracted. The steps performed in preparing the test data are summarized in Figure 4.

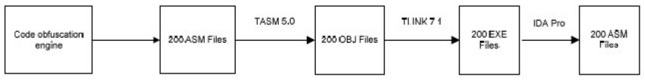


Fig. 4: Test data preparation

Note that disassembled files obtained from EXE files were used for training and testing. Consequently, our training and testing is realistic in the sense that only EXE files would be available to antivirus software.

We performed 5-fold cross validation, that is, we split the 200 metamorphic virus files into 5 subsets of 40 viruses each. From among these five subsets, four were used for training and the remaining one was reserved to test the trained HMM model. This process was repeated five times, once for each distinct 4-subset collection of morphed files. In each case, 40 metamorphic files were scored along with 40 "normal" files. For the normal files, we used Cygwin utility files, since these files were also used as the representative normal files in [16] and [8].

6. Experimental Results

For the test case considered in this paper, the Next Generation Virus Creation Kit (NGVCK) was used to create the seed viruses. Other viruses were considered, with equally strong results obtained in each case; see [15] for more details on these other experiments.

In each case, popular antivirus scanners could detect the seed virus, but not the viruses produced by our metamorphic generator. That is, our metamorphic generator is able to successfully bypass signature detection, as expected. However, regardless of the seed virus used, the HMM engine was able to distinguish the morphed viruses from normal code, as discussed below.

Virus creation, analysis and testing experiments were conducted using the platform and tools listed in Table 3. Again, the procedure followed here follows that used in [16].

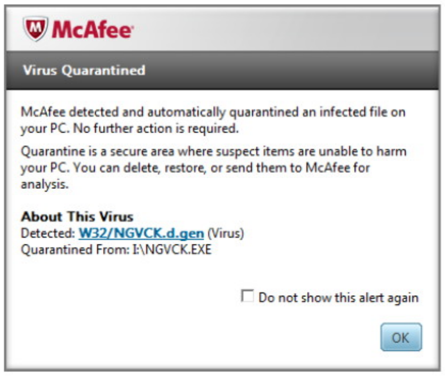
Table 3: Experimental setup [15]

Platform:	Windows XP/VMware
Language:	Perl5
Disassemblers:	OllyDbg v1.10 and IDA Pro 4.9
Assembler:	Borland Turbo Assembler 5.0
Linker:	Borland Turbo Linker 7.1
Virus generators:	MPCGEN (Phalcon/Skism Mass Code Generator)
	G2 (Generation 2 Virus Generator)
	VCL32 (Virus Creation Lab for Win32)
	NGVCK (Next Generation Virus Creation Kit)
Virus scammers:	Avast Home Edition 4.8
	McAfee Antivirus 2009

As mentioned above, the seed viruses were detected by commercial antivirus software. For example, Figure 5 shows a screenshot of the security alert displayed by McAfee antivirus when it scanned one of our seed viruses.

Next, we present the results from one typical experiment. In this example, we used our engine to generate 200 metamorphic variants of an NGVCK seed virus. The parameters were set to generate variants with a threshold of two garbage instructions. Snippets of code from two of the resulting metamorphic variants appear in Figure 6.

The metamorphic viruses were then assembled and the resulting morphed executables scanned using popular antivirus scanners by McAfee and Avast [1]. As expected, these scanners were not able to identify the morphed executables as viruses.



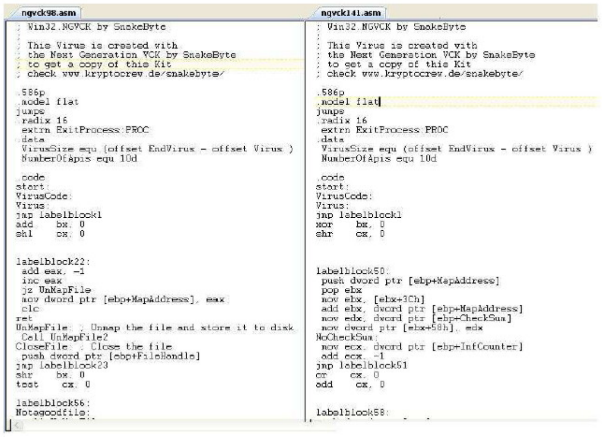
McAfee detected and automatically quarantined an infected file on your PC. No further action is required.

Quarantine is a secure area where suspect items are unable to harm your PC. You can delete, restore, or send them to McAfee for analysis.

About This Virus
Detected: W32/NGVCK.d.gen (Virus)
Quarantined From: I:\NGVCK.EXE

☐ Do not show this alert again

Fig. 5: Seed virus scanned with McAfee



```
; Win32.NGVCK by SnakeByte
;
; This Virus is created with
; the Next Generation VCK by SnakeByte
; to get a copy of this Kit
; check www.kryptocrew.de/snakebyte/

.586p
.model flat
jumps
.radix 16
 extrn ExitProcess:PROC
.data
 VirusSize equ (offset EndVirus - offset Virus )
 NumberOfApis equ 10d

.code
start:
VirusCode:
Virus:
jmp labelblock1
add     bx, 0
shl     cx, 0


labelblock22:
 add eax, -1
 inc eax
 jz UnMapFile
 mov dword ptr [ebp+MapAddress], eax
 clc
ret
UnMapFile: ; Unmap the file and store it to disk
 Call UnMapFile2
CloseFile: ; Close the file
 push dword ptr [ebp+FileHandle]
jmp labelblock23
shr     bx, 0
test     cx, 0

labelblock56:
Notagoodfile:
```

```
; Win32.NGVCK by SnakeByte
;
; This Virus is created with
; the Next Generation VCK by SnakeByte
; to get a copy of this Kit
; check www.kryptocrew.de/snakebyte/

.586p
.model flat
jumps
.radix 16
 extrn ExitProcess:PROC
.data
 VirusSize equ (offset EndVirus - offset Virus )
 NumberOfApis equ 10d

.code
start:
VirusCode:
Virus:
jmp labelblock1
xor     bx, 0
shr     cx, 0



labelblock50:
 push dword ptr [ebp+MapAddress]
 pop ebx
 mov ebx, [ebx+3Ch]
 add ebx, dword ptr [ebp+MapAddress]
 mov edx, dword ptr [ebp+CheckSum]
 mov dword ptr [ebx+58h], edx
NoCheckSum:
 mov ecx, dword ptr [ebp+InfCounter]
 add ecx, -1
jmp labelblock51
or     cx, 0
add     cx, 0

labelblock58:
```

Fig. 6: Sample metamorphic code

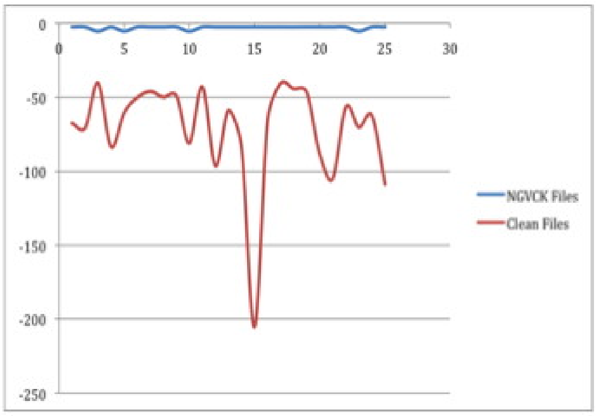


Fig. 7: HMM scores

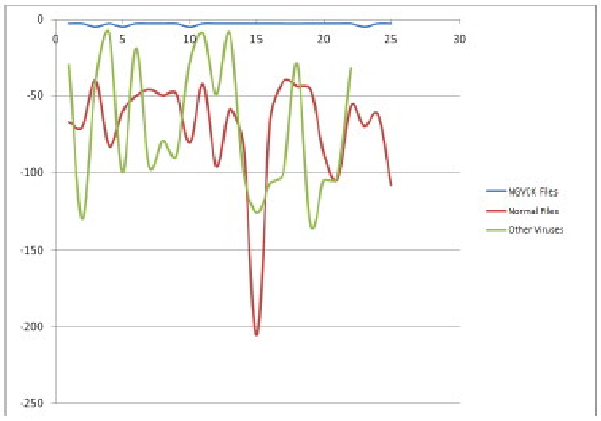


Fig. 8: HMM scores including non-family viruses

Using 5-fold cross validation, HMM models were trained using this set of 200 metamorphic viruses. The number of distinct observation symbols (i.e., opcodes) ranged from 40 to 42 and the total number of observations ranged from 41,472 to 42,151. The resulting model was then used to score 40 viruses and 40 normal files. A typical HMM score graph appears in Figure 7, where the scores are given as log likelihood per opcode (LLPO). That is, the scores were computed based on log odds, then normalized on a per opcode basis. In every case, a threshold could easily be set that would provide flawless detection, that is, no false positives or false negatives would occur. In fact, the score differences are quite large given that the scores are computed on a per opcode basis.

Figure 8 shows a similar graph as that which appears in Figure 7, but with 40 additional, non-family viruses included. Note that some of the non-family viruses score significantly higher than any of the normal files. However, we can still set a threshold that results in no false positives or false negatives. The fact that other viruses have relatively high scores is not too surprising, and might be considered beneficial, since we could adjust the threshold and thereby detect some additional related malware.

7. Conclusions and Future Work

In this paper, we analyzed a metamorphic generator satisfying the conditions given in [4]. The paper [4] provides a rigorous proof that such viruses cannot be efficiently detected using "static analysis," according to their definition of the term. As expected, these metamorphic viruses are not susceptible to signature detection. However—and perhaps surprisingly—these viruses are detected via a machine learning approach. Specifically, we trained HMM models to detect such viruses. While the work presented here does not directly contradict [4], it does call into question the utility of relying on such a narrow definition of "static analysis" since, by most accounts, our HMM approach would be considered a static technique.

At this point, a natural question to ask is whether a practical metamorphic generator can be produced that will evade both signature detection and our HMM-based detector. The paper [5] was a first attempt to settle this question while [8] shows conclusively that a practical metamorphic generator can evade both signature detection and the HMM based approach used in this paper. However, it is not yet entirely clear where, in a practical sense, the ultimate balance of power lies between metamorphic virus writers and detection.

At first blush, the results in [4] seem to prove that in a very practical and real sense, metamorphic virus writers have an insurmountable advantage, at least from the perspective of static analysis. However, the results in this paper show that the reality of the situation is considerably more nuanced.

References

[1] Avast Antivirus, `http://www.avast.com/`

[2] S. Attaluri, S. McGhee and M. Stamp, Profile hidden Markov models and metamorphic virus detection, *Journal in Computer Virology*, vol. 5, no. 2, May 2009, pp. 151–169

[3] J. Aycock, *Computer Viruses and Malware*, Springer, 2006

[4] J. Borello and L. Me, Code obfuscation techniques for metamorphic viruses, *Journal in Computer Virology*, vol. 4, no. 3, August 2008, pp. 211–220

[5] P. Desai, Towards an undetectable computer virus, MasterÕs project report, Department of Computer Science, San Jose State University, 2008, `http://www.cs.sjsu.edu/faculty/stamp/students/Desai_Priti.pdf`

[6] E. Filiol, Metamorphism, Formal grammars and undecidable code mutation, *World Academy of Science, Engineering and Technology*, vol. 26, 2007

[7] IDA Pro, `http://www.hex-rays.com/idapro/`

[8] D. Lin and M. Stamp, Hunting for undetectable metamorphic viruses, to appear in *Journal in Computer Virology*

[9] L. R. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition, *Proceedings of the IEEE*, vol. 77, no. 2, 1989

[10] D. Spinellis, Reliable identification of bounded-length viruses is NP-complete, *IEEE Transactions on Information Theory*, vol. 49, no. 1, January 2003, pp. 280–284

[11] M. Stamp, A revealing introduction to hidden Markov models, January 2004, `http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf`

[12] P. Szor and P. Ferrie, Hunting for metamorphic, Symantec Security Response, `http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf`

[13] P. Szor, *The Art of Computer Virus Defense and Research*, Symantec Press, 2005

[14] VX Heavens, `http://vx.netlux.org/`

[15] S. Venkatachalam, Detecting undetectable computer viruses, MasterÕs project report, Department of Computer Science, San Jose State University, 2010, `http://www.cs.sjsu.edu/faculty/stamp/students/venkatachalam_sujandharan.pdf`

[16] W. Wong and M. Stamp, Hunting for metamorphic engines, *Journal in Computer Virology*, vol. 2, no. 3, December 2006, pp. 211–229

[17] P. Zbitskiy, Code mutation techniques by means of formal grammars and automatons, *Journal in Computer Virology*, vol. 5, no. 3, August 2009, pp. 199–207