THE NEW AGE OF COMPUTER VIRUS AND THEIR DETECTION

Nitesh Kumar Dixit[1], Lokesh mishra[2], Mahendra Singh Charan[3] and Bhabesh Kumar Dey[4],

[1]Department of Electronics and Communication Engineering, BIET, Sikar, Raj., INDIA
[1]nitesh20.dixit@gmail.com
[2]Department of Computer Science, NCAC,Sikar,Raj. INDIA
[2]llokesh.mishra@gmail.com
[3,4]Department of Computer Engineering, BIET, Sikar, Raj., INDIA
[3]charanmahendrasingh@gmail.com
[4]mr.bhabesh@gmail.com

ABSTRACT

This paper presents a general overview on computer viruses and defensive techniques. Computer virus writers commonly use metamorphic techniques to produce viruses that change their internal structure on each infection. On the other hand, anti-virus technologies continually follow the virus tricks and methodologies to overcome their threats. In this paper, anti-virus experts design and develop new methodologies to make them stronger, more and more, every day. The purpose of this paper is to review these methodologies and outline their strengths and weaknesses to encourage those are interested in more investigation on these areas. In this paper, first analyze four virus creation kits to determine the degree of metamorphism provided by each and able to precisely quantify the degree of metamorphism produced by these virus generators. While the best generator, the Next Generation Virus Creation Kit (NGVCK), produces virus variants that differ greatly from one another, the other three generators examined are much less effective.

KEYWORDS

Antivirus Techniques, Computer Antivirus, Creation, Defensive, Metamorphism, NGVCK, Virus.

1. INTRODUCTION

"A computer virus is a program that recursively and explicitly copies a possibly evolved version of itself" [1]. A virus copies itself to a host file or system area. Once it gets control, it multiplies itself to form newer generations. Over the past two decades, the number of viruses has been increasing rapidly. In 1999, the infamous Melissa virus infected thousands of computers and caused damage close to $80 million; while the Code Red worm outbreak in 2001 affected systems running Windows NT and Windows 2000 server and caused damage in excess of $2 billion [2]. To simplify the virus creation process, virus writers have made virus construction kits readily available on the Internet [3].

In this paper, a single hidden Markov model (HMM) is used to determine whether a given program belongs to the virus family that the HMM represents. This approach can be used to distinguish family member viruses from non-member programs. The challenges with the HMM

DOI : 10.5121/ijnsa.2012.4305

approach include finding the right balance between sensitivity and specificity, and conforming to the time and space constraints of the computers performing the detection, and evaluated the effectiveness of this approach by its detection rate, the false positive and false negative rates, and the overall accuracy of the classification.

2. EVOLUTION OF VIRUS

Computer malwares can be classified according to their different characteristics in several various manners, such as classification by target or classification by infection mechanism. One of these classification types is according to concealment techniques employed [4].

2.1 Virus Obfuscation Techniques

Virus-like programs first appeared on microcomputers in the 1980s. To challenge virus scanning products, virus writers constantly develop new obfuscation techniques to make virus code more difficult to detect. To escape generic scanning, a virus can modify its code and alters its appearance on each infection. The techniques that have been employed to achieve this end range from *encryption* to *polymorphic* techniques, to modern *metamorphic* techniques [5] [6].

2.1.1 Encrypted Viruses

The simplest way to change the appearance of a virus is to use encryption. An encrypted virus consists of a small decrypting module (a decryptor) and an encrypted virus body. If a different encryption key is used for each infection, the encrypted virus body will look different. Typically, the encryption method is rather simple, such as xor of the key with each byte of the virus body. Simple xor is very practical because xoring the encrypted code with the key again will give the original code and so a virus can use the same routine for both encryption and decryption. With encryption, the decryptor remains constant from generation to generation. As a result, detection is possible based on the code pattern of the decryptor. A scanner that cannot decrypt or detect the virus body directly can recognize the decryptor in most cases.

2.1.2 Polymorphic Viruses

To overcome the problem of encryption, namely the fact that the decryptor code is long and unique enough for detection, Polymorphic viruses can change their decryptors in newer generations. They can generate a large number of unique decryptors which use different encryption method to encrypt the virus body. A polymorphic virus thus has no parts that stay constant on each infection. To detect polymorphic viruses, anti-virus software incorporates a code emulator which emulates the decryption process and dynamically decrypts the encrypted virus body. Because all polymorphic viruses carry a constant virus body, detection is still possible based on the decrypted virus code.

2.1.3 Metamorphic Viruses

To make viruses more resistant to emulation, virus writers developed numerous advanced metamorphic techniques. According to Muttik, “Metamorphics are bodypolymorphics”. A metamorphic virus not only changes it decryptor on each infection but also its virus body. New virus generations look different from one another and they do not decrypt to a constant virus body. A metamorphic virus changes its “shape” but not its behavior. This is illustrated diagrammatically by Szor in [7], and is shown in Figure 1. Because all polymorphic viruses carry a constant virus body, detection is still possible based on the decrypted virus code.

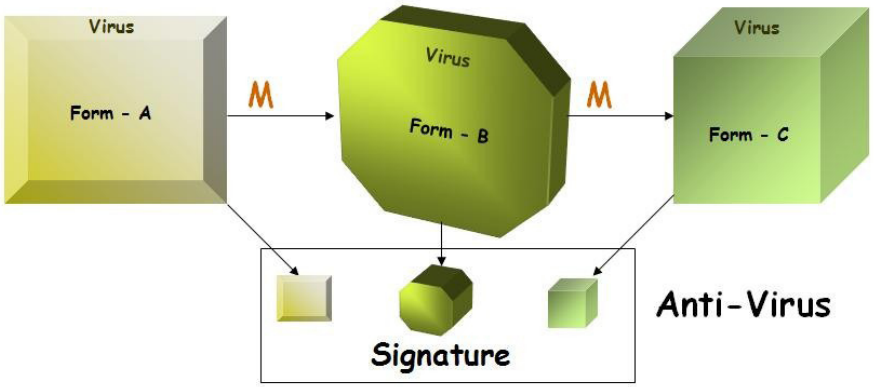


Figure 1: Metamorphic virus generations

3. VIRUS DETECTION SYSTEMS

3.1 Signature based virus detection

Signature based detection systems scan the files for specific signatures that are present in them. The pattern of instructions present in a virus code is identified as the signature of the virus file. This will raise an alarm for virus if the signature of a virus is detected in any of the files scanned. This method of intrusion detection is fast and accurate since the chances of false alarms are very low in this system. The main requirement of the system is to have an updated database of all the signature files of malware. The accuracy is totally dependent on the signature database of the system. Signature based detection systems cannot detect a new virus since the database will not have any information about the new virus. An antivirus scanner extracts the opcode pattern from an executable file and searches the signature database for the input opcode pattern. The input opcode pattern is considered as the signature of the input file. If a match is found in the signature database, the input file is classified as the corresponding virus family matched in the signature database. For example, if the signature of the input file is 83EB 0274 EB0E 740A 81EB 0301 0000, then this will be searched in the signature database and the file will be classified as W32/Beast virus since 83EB 0274 EB0E 740A 81EB 0301 0000 is the signature of the W32/Beast virus [8][9]. A similar search pattern used to detect Stoned virus is shown in Figure 2.

```
seg000:7C40 BE 04 00                    mov     si, 4           ; Try it 4 times
seg000:7C40                                                     ;
seg000:7C43
seg000:7C43                 next:                               ; CODE XREF: sub_7C3A+27↓j
seg000:7C43 B8 01 02                    mov     ax, 201h        ; read one sector
seg000:7C46 0E                          push    cs
seg000:7C47 07                          pop     es
seg000:7C48                             assume es:seg000
seg000:7C48 BB 00 02                    mov     bx, 200h        ; to here
seg000:7C4B 33 C9                       xor     cx, cx
seg000:7C4D 8B D1                       mov     dx, cx
seg000:7C4F 41                          inc     cx
seg000:7C50 9C                          pushf
seg000:7C51 2E FF 1E 09 00              call    dword ptr cs:9  ; int 13
seg000:7C56 73 0E                       jnb     short fine
seg000:7C58 33 C0                       xor     ax, ax
seg000:7C5A 9C                          pushf
seg000:7C5B 2E FF 1E 09 00              call    dword ptr cs:9  ; int 13
seg000:7C60 4E                          dec     si
seg000:7C61 75 E0                       jnz     short next
seg000:7C63 EB 35                       jmp     short giveup
```

Figure 2: Search Pattern for Stoned virus

3.2 Anomaly based virus detection

Anomaly based detection systems monitor the processes on a host machine for any abnormal activity. If any abnormal activity is identified, the system raises an alarm signaling the possible presence of malware [10]. In this detection technique, the system uses the collected heuristics to categorize an activity as normal or malicious. Even though chances of false alarm are relatively higher in this method, it is more reliable because it is also capable of detecting new viruses. The important thing to note is that raising a false alarm is not as potential harmful as allowing a new virus. However, these systems can be trained gradually by intruders to consider abnormal behavior as normal. Thus, system will fail to detect the abnormal activity in such cases [10].

3.3 Emulation based detection

The emulation based detection is an effective method where a virus is executed in a virtual environment by emulating the instructions in the virus code. This type of detection is used to detect polymorphic, as well as metamorphic, viruses. The virus instance can be executed in the virtual environment in order to identify instruction sequence or behavior of the virus [8][11]. In addition to the virtual environment, code optimization techniques can be applied to the execution process to decrease the time for detection. Table 1 lists the strength and weakness of these detection methods.

Table 1: Virus Detection Techniques

Detection Technique	**Strength**	**Weakness**
Signature based	Efficient	New Malware
Anomaly based	New Malware	Costly to implement, False Positives, Unproven
Emulation based	Encrypted Viruses	Costly to Implement

3.4 Use of Machine Learning Techniques

Various researchers have attempted to use machine learning techniques to perform heuristic analysis on metamorphic viruses. This section covers the result and potential of some of the techniques, which include[12]:

1) Data mining methods
2) Neural networks
3) Hidden Markov models.

3.4.1 Data Mining Approach

Data mining methods are often used to detect patterns in a large set of data. These patterns are then used to identify future instances in a similar type of data. Schultz et al. experimented with a number of data mining techniques to identify new malicious binaries [13]. They used three learning algorithms to train a set of classifiers on some publicly available malicious and benign executables. They compared their algorithms to a traditional signature-based method and reported a higher detection rate for each of their algorithms. However, their algorithms also resulted in higher false positive rates when compared to signature-based method.

The key to any data mining framework is the extraction of features, which are properties extracted from examples in the dataset. Schultz et al. extracted some static properties of the binaries as features. These include system resource information (the list of DLLs, the list of DLL function calls, and the number of different function calls within each DLL) obtained from the program header, and consecutive printable characters found in the files. The most informative feature they used was byte sequences, which were short sequences of machine code instructions generated by the hexdump tool.

The features were used in three different training algorithms. There was an inductive rule-based learner that generated Boolean rules to learn what a malicious executable was; a probabilistic method that applied Bayes rule to compute the likelihood of a particular program being malicious, given its set of features; and a multi-classifier system that combined the output of other classifiers to give the most likely prediction.

3.4.2 Neural Networks

Researchers at IBM implemented a neural network for heuristic detection of boot sector viruses [13]. The features they used were short byte strings, called trigrams, which appear frequently in viral boot sectors but not in clean boot sectors. They extracted about 50 features from a corpus of training data, which consisted of both viral and legitimate boot sectors. Each sample in the dataset was then represented by a Boolean vector indicating the presence or absence of these features.

The network was single-layered with no hidden units. It was trained using classic back propagation technique. One common problem with neural network is over fitting, which occurs when a network is trained to identify the training set but fails to generalize to unseen instances. To eliminate this problem, multiple networks were trained using different features and a voting scheme was used to determine the final prediction. The neural network was able to identify 80-85% of viral boot sectors in the validation set with a false positive rate of less than 1%. The neural network classifier has been incorporated into the IBM AntiVirus software which has identified about 75% of new boot sector viruses since it was released [13]. A similar technique was later applied by Arnold and Tesauro to successfully detect Win32 viruses [1]. From , we can conclude that neural networks are very effective in detecting viruses closely related to those in the training set. They can also identify new families of viruses containing similar features as the training samples.

3.4.3 Hidden Markov Models

Hidden Markov models (HMMs) are well suited for statistical pattern analysis. Since their initial application to speech recognition problems in the early 1970's , HMMs have been applied to many other areas including biological sequence analysis [14] . An HMM is a state machine where the transitions between states have fixed probabilities. Each state in an HMM is associated with a probability distribution for observing a set of observation symbols. We can "train" an HMM to represent a set of data, which is usually in the form of observation sequences. The states in the trained HMM then represent the features of the input data, while the transition and the observation probabilities represent the statistical properties of these features. Given any observation sequence, we can match it against a trained HMM to determine the probability of seeing such a sequence. The probability will be high if the sequence is "similar" to the training sequences. In protein modeling, HMMs are used to model a given family of proteins . The states correspond to the sequence of positions in space while the

observations correspond to the probability distribution of the 20 amino acids that can occur in each position. A model for a protein family assigns high probabilities to sequences belonging to that family. A trained HMM can then be used to discriminate family members from non-members. Metamorphic viruses form families of viruses. Even though members in the same family mutate and change their appearances, some similarities must exist for the variants to maintain the same functionality. Detecting virus variants thus reduces to finding ways to detect these similarities. Hidden Markov models provide a means to describe sequence variations statistically. We propose to use HMMs similar to those used in protein sequence analysis to model virus families. In virus modeling, the states correspond to the features of the virus code, while the observations are instructions or opcodes making up the program. A trained model should then be able to assign high probabilities to and thus identify viruses belonging to the same family as the viruses in the training set.

4. HIDDEN MARKOV MODEL

Hidden Markov Model also known as HMM is a statistical pattern analysis tool. HMM creates a model representing the input data. This input data is called training data. The training data consists of a list of unique symbols and their positional information in input sequence. HMM uses this model to determine if a given input sequence follows similar pattern as the model. HMM is widely used for speech recognition and protein modeling. Recently HM M has been successfully used to detect metamorphic viruses [15][16]. Metamorphic viruses are a family of viruses that changes in appearance while preserving the same functionality. Generally a family of viruses have similar pattern. Given a family of viruses HMM can come up with the statistical model representing the family [17][18][19]. Now any virus can be tested against several such models to determine which family it belongs to.

4.1 HMM as Virus Detection Tool

HMM as virus detection tool requires training data to produce a model. The training data consists of observation sequence and unique symbols. The observation sequence and unique symbols are derived from several viruses of a family [20][21][22]. These viruses are programs written in assembly language. The observation symbols are unique assembly opcodes among all viruses. The opcodes of all viruses are concatenated to produce one long observation sequence. HMM is trained on this observation sequence to produce the model. An example of such observation sequence is shown in figure 3. The model is shown in figure 4, and result shown in Figure 5.

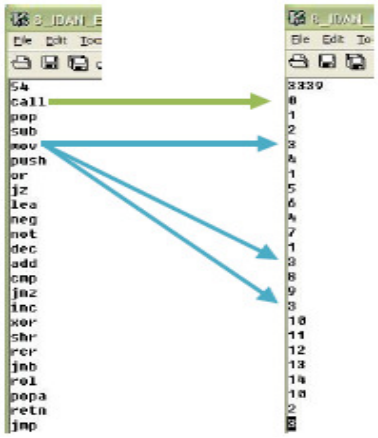


Figure 3: Observation sequence

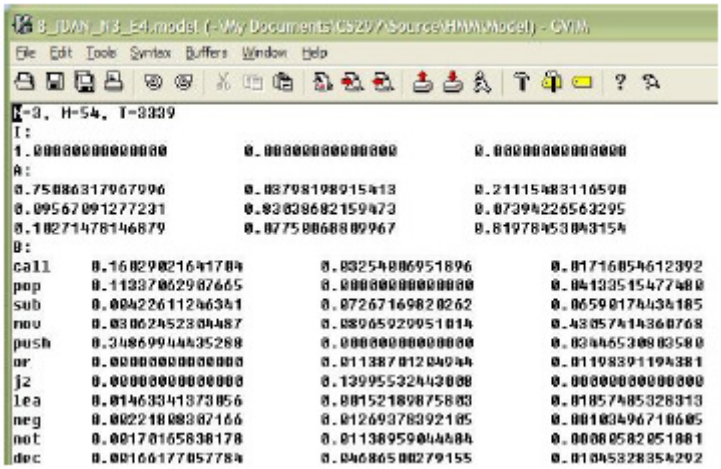


Figure 4: Process Model

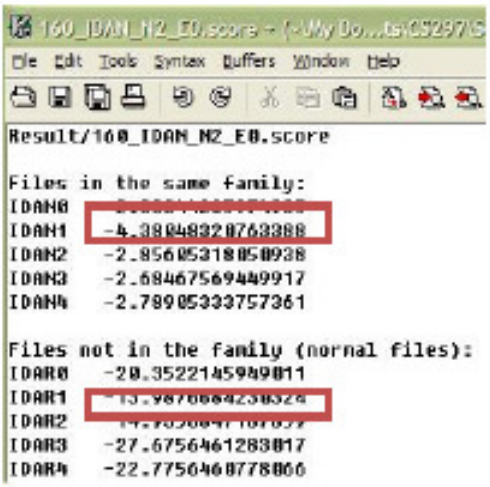


Figure 5: The Result File

In the result file, IDAN0 to IDAN4 are the viruses from the same family. The score for these viruses is greater than -4.38 which are defined as a threshold. A file with a score less than the threshold is not considered as part of this family. The files IDAR0 to IDAR4 have scores less than the threshold and therefore not in the family.

5. IMPLEMENTATION

5.1 Introduction

In general metamorphic engine has to implement some or all code obfuscation techniques. In addition to using these techniques, each implementation will have its own heuristics. These heuristics may include processes that decide type of obfuscation techniques to use, when to apply them, and how to apply them. The implementation by following some of the existing metamorphic engines like Evol. Evol is a metamorphic virus that used code obfuscation techniques such as dead code insertion, register / operands usage exchange, and equivalent

instruction substitution. In addition to the techniques used by Evol, we added few more variations of these techniques. This section gives detailed explanation of the code obfuscation techniques we used.

5.2 Goals

The implementation has following goals:

1) Generate morphed copies of a single input virus. These morphed copies should have minimum similarity with the base virus and among themselves.

2) The morphed copies should have same functionality as the base virus.

3) Morphed copy should be close to normal program.

4) The metamorphic engine should work on any assembly program.

Assumption here is the normal programs are the cygwin utility files of the same size as the base virus.

The reason behind using cygwin utility files is they probably are doing same low level operations as a virus.

5.3 Code Obfuscation Techniques Used

5.3.1 Dead Code Insertion

Dead code insertion is adding NOP or do-noting instructions. The dead code insertion to introduce opcodes that are alien to the base virus. The alien opcodes were determined by analyzing the base virus and normal programs. First generate statistics of the base virus to find out all the opcodes used. The graph in figure 6 below lists the opcodes used in the base virus with their frequency.

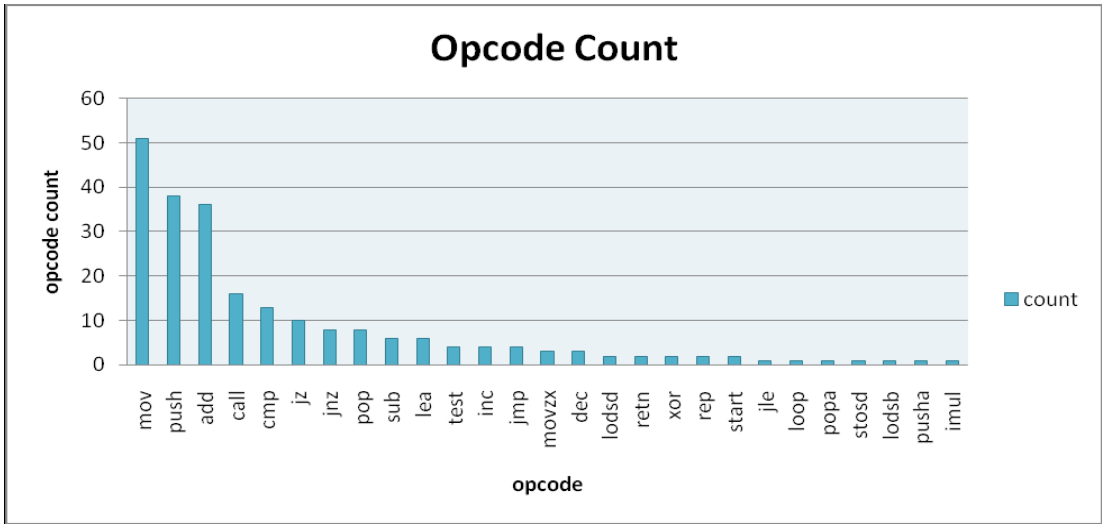


Figure 6: Base virus opcodes and their frequency

The base virus has 27 unique opcodes and six of them appear more than 10 times. Opcodes mov, push, add, call, cmp, and jz are the most frequent appearing opcodes. The designed dead opcode set to include more of the infrequent used opcodes. After then analyze the normal program for its opcode frequency. The graph in figure 7 shows the statistics of a normal file.

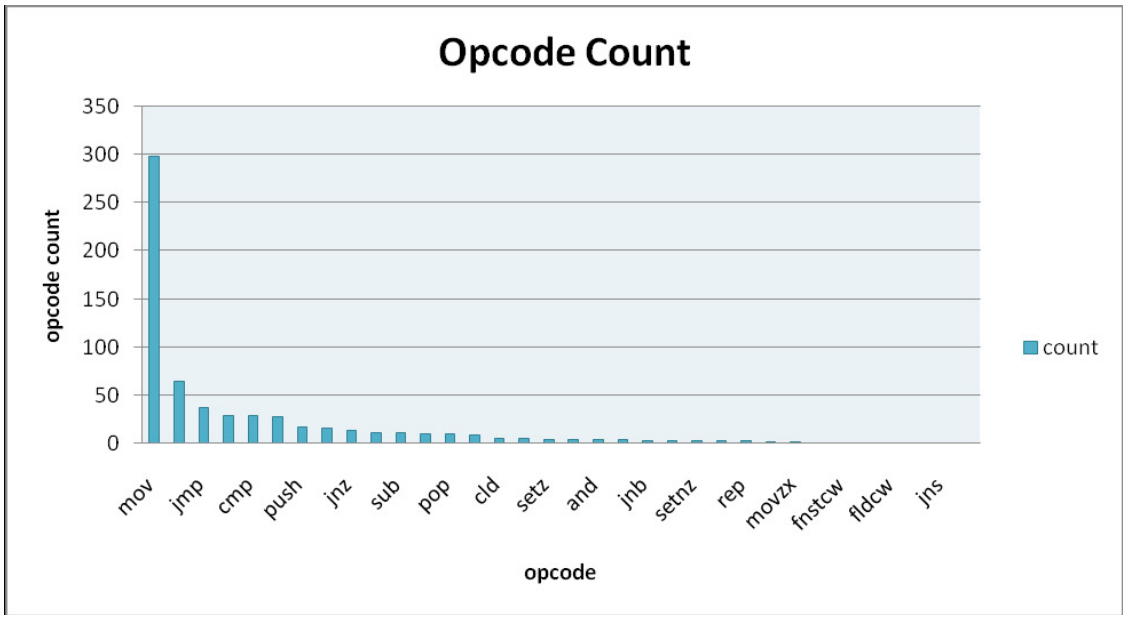


Figure 7: Opcodes of normal file and their frequency

When the statistics of a normal file is compared with the base virus, get the list of opcodes that are unique to a normal file. The unique opcodes are *AND, INT, FNSTCW, OR, FLDCW, LEAVE, JNS, SETNZ, SETZ, JB, CLD, JNB, SHL, INC, FLD, FSTP,* and *REPE*.

This comparison shows that the above unique opcodes should be included in morphed copies to make them look more like a normal file. Based on this conclusion the dead code instructions are modeled to include most of the above unique opcodes. The table 2 shows some examples of dead code instructions used. Refer to Appendix A for complete list of dead code instruction.

Table 2: Arithmetic Dead Code Instructions

1. Add R,0
2. Sub R,0
3. Adc bx,0
4. Sbb bx,0
5. Inc R followed by dec R

These dead code instructions are injected at randomly selected locations in the base virus. For every selected location, insert a single dead code instruction. The dead code instruction to be inserted is randomly selected. These are categorized as simple single NOP instruction substitution. As the variation to simple single NOP instruction substitution, we introduced

unconditional jump NOP instruction substitution. The jump NOP works by introducing unconditional jump to next immediate instruction. An example of this variation is shown below.

Mov edx, [esi + entryPoint] ⇨ p1010235:

Jmp pl1010235

Mov edx, [esi + entryPoint]

5.3.1.1 NOP sequence insertion

Dead code insertion is used to insert a single NOP Instruction. In NOP sequence insertion, a random sequence of NOP instructions are inserted at randomly selected locations. The locations to insert NOP sequence were categorized in two viz. beginning of the code section and rest of the code section. To insert or not to insert a NOP sequence in the beginning of the code section is decided randomly. While for the rest of the code section, the insertion location and a NOP sequence is selected randomly. The Algorithm to insert NOP sequence on entry point is

1. Determine entry point of a virus.
2. Generate random number between 0 to 3
3. If the random number is 0 then insert NOP sequence
4. To inset NOP sequence:
    a. Randomly select length of a NOP sequence from 3, 5, and
    b. Generate random permutation of the above selected length.
    c. Insert this sequence into a virus.

5.3.1.2 Transformations of Evol

Along with a single dead code insertion and a NOP sequence insertion, we introduced some new dead code insertions. These insertions are inspired from Evol virus [6], shown in table 3. Evol virus substitutes a single instruction by surrounding it with dead code.

Table 3: Evol transformations

Original	**Transformed**
Mov r/m, reg	Push Randomreg
Mov reg, r/m	MOV Randomreg, OriginalReg
TEST r/m, reg	Add Randomreg, RandomImm8
LEA r32, mem	OP r/m – Randomreg, originalReg
Op reg, r/m	POP Randomreg

One disadvantage with these transformations is an instruction is substituted with a block of instructions beginning with push followed by some instructions and ending with pop. Therefore these transformations increase the number of push and pop opcodes. This also creates a pattern of starting with push and ending in pop [20].

5.3.2 Equivalent instruction substitution

Some opcodes appear frequently in the base virus like mov, push, add, call, cmp, and jz. To minimize the number of these opcodes, we used equivalent instruction substitution. In an equivalent instruction substitution, an instruction is replaced with another instruction or a block of instructions with the same functionality. For example substitutions for add are listed in table 4.

Table 4 : substitutions for Add

Add R, imm	1. Sub R, new_imm where new_imm = imm x(-1) 2. Lea R, [R+imm]
Add R, 1	1. Not R 2. Neg R

Here, opcode add is replaced with opcodes like “sub”, “lea”, and “not” followed by “neg”. Similarly opcodes like mov, cmp, test etc are replaced with equivalent instructions. The substitution for each instruction is decided based on the type of operands like

REG (8), REG (8)	REG (16), REG (16)	REG (32), REG (32)
REG (8), MEM	REG (16), MEM	REG (32), MEM
REG (8), IMM	REG (16), IMM	REG (32), IMM
MEM, REG (8)	MEM, REG (16)	MEM, REG (32)
MEM, IMM		

5.3.3 Transpose

After a morph copy is generated using dead code insertion and equivalent substitution, apply transpose to generate final output.

6. EXPERIMENTS

Firstly generate a large of number of metamorphic virus variants of the base virus with new designed metamorphic engine. The metamorphic virus variants were generated by applying the metamorphic engine iteratively over a single base virus. Applying metamorphic engine once on an input is 1st generation metamorphism. Applying the metamorphic engine twice on an input is 2nd generation metamorphism and so on. The metamorphic engine can take any assembly program as input. The output is a morphed copy of the input. These assembly sources are then complied into executables using FASM [21]. These executables are then disassembled using IDA Pro with default settings (686 instruction set) [22] [23][24]. These assembly programs were used to perform all tests. To keep the tests more realistic IDA-pro generated assembly files were used rather than the original assembly source from the engine shown in figure 8. All tests were performed using two different tools. These include Commercial virus scanner, Similarity Test, and statistical pattern analysis tool such as Hidden Markov Model [25][26].

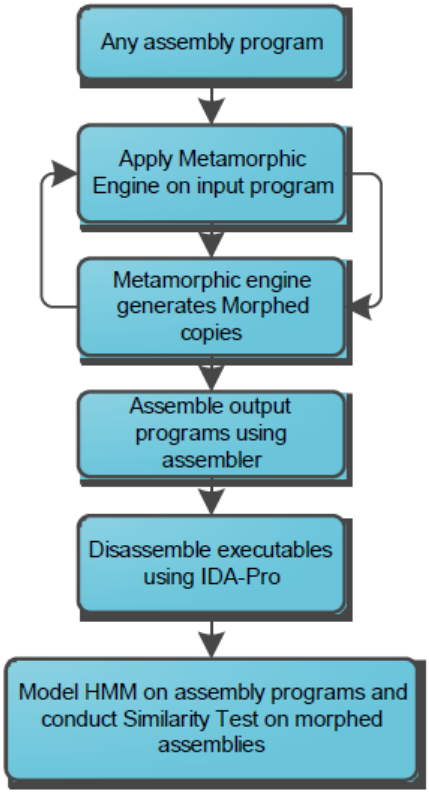


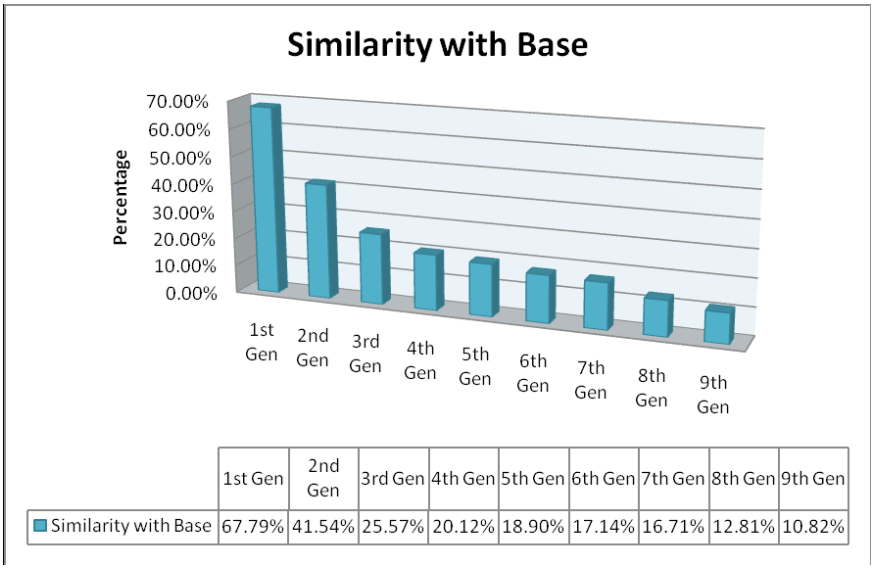
Figure 8: All over Process

6.1 Commercial virus scanner

In our testing, the base virus was successfully detected and quarantined by the commercial virus scanner installed on our machine. But the same virus scanner failed to detect morphed copies of the base virus[27].

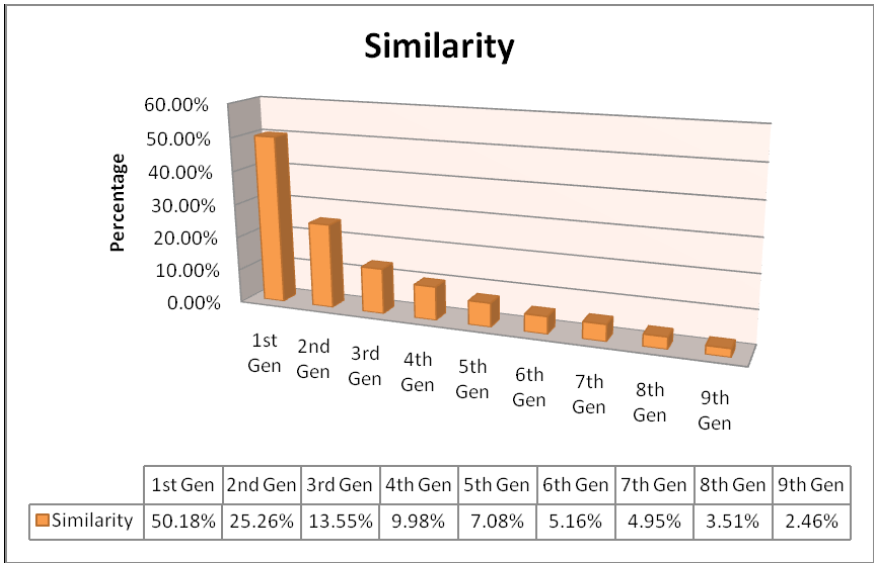
6.2 Similarity Test

Similarity test compares and reports the percentage of similarity of two assembly programs. The purpose of the similarity test is to measure the code diversity of the morphed copies. We compared the base virus with 1st to 9th generations of metamorphic copies. These comparisons were performed using the default settings of similarity test i.e. 10 opcodes in a sequence is considered a match. The result of this test is shown below in figure 9. The similarity between the base virus and 1st generation virus is about 70%. The similarity decreases with higher generations. 9th generation virus is about 10% similar to the base virus. After applying the metamorphic engine to the base virus, the number of opcodes in morphed copies increases. The dissimilar length of the compared files may affect similarity test. So we compared a pair of viruses from the same generation. The viruses from the same generation are of similar length. 1st generation viruses are about 50% similar whereas 9th generation viruses are about 2.5% similar as shown in figure 10. Note that, the viruses generated by Next Generation Virus Creation Kit (NGVCK) were found to be about 10% similar with default settings [2]. Based on these similarity tests, decide to model HMM on highly dissimilar generation which is 9th generation.



	1st Gen	2nd Gen	3rd Gen	4th Gen	5th Gen	6th Gen	7th Gen	8th Gen	9th Gen
Similarity with Base	67.79%	41.54%	25.57%	20.12%	18.90%	17.14%	16.71%	12.81%	10.82%

Figure 9: Similarity results of the base virus v/s 9 different generations



	1st Gen	2nd Gen	3rd Gen	4th Gen	5th Gen	6th Gen	7th Gen	8th Gen	9th Gen
Similarity	50.18%	25.26%	13.55%	9.98%	7.08%	5.16%	4.95%	3.51%	2.46%

Figure 10: Graph of Similarity of two N generations

6.3.2 The Base virus against the morphed virus model

After then model HMM for odd generations of viruses. The base virus was tested against these modes and scores are listed in table 5. Results shows the statistical pattern of the base virus can still be detected by different generation of viruses shown in figure 11.

Table 5: The base virus tested against N Generation Model

Model	Score
1st Generation Model	-2.26519095918038
3rd Generation Model	-2.5616088296304
5th Generation Model	-2.7804691006756
7th Generation Model	-6.53547571903687
9th Generation Model	-9.36420192759975

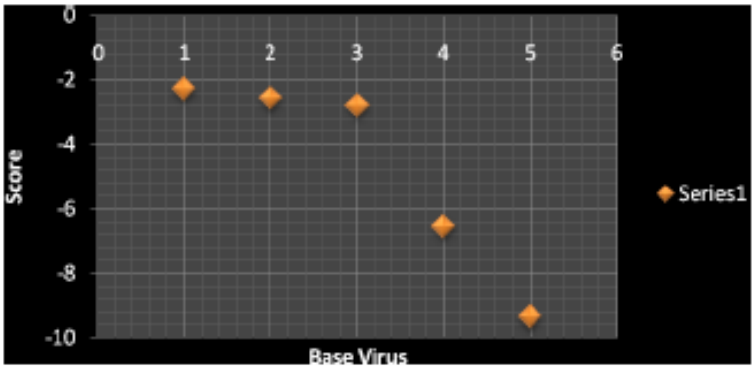


Figure 11: Base virus tested against N generation models

6.3.4 Morphed viruses against normal file model

The collected 40 cygwin files as a set of normal files. The generated HMM model on a set of normal files. Then 9th generation viruses are tested against this model. The threshold for normal files is -180.5254.All 9th generation viruses scored higher than the threshold. The maximum score of 9th generation viruses is -37.2978. So the 9th generation viruses are considered as normal files. This is 100% false positives, result show in figure 12 and 13 .

Normal model with N = 2			
Normal Files		9th Generation Viruses	
N0	-21.9658	G9_0	-173.3586
N1	-5.20571	G9_1	-160.9587
N2	-180.5254	G9_2	-154.1496
N3	-4.53708	G9_3	-159.1445
N4	-1.7961	G9_4	-168.9089
N5	-1.7246	G9_5	-169.4739
N6	-1.7961	G9_6	-164.7176
N7	-2.0771	G9_7	-37.2978
N8	-2.0542	G9_8	-169.2335
N9	-1.7599	G9_9	-158.5317
Min Score = -180.5254		Max Score = -37.2978	

Figure 12: Results of 9th generation viruses tested against normal model

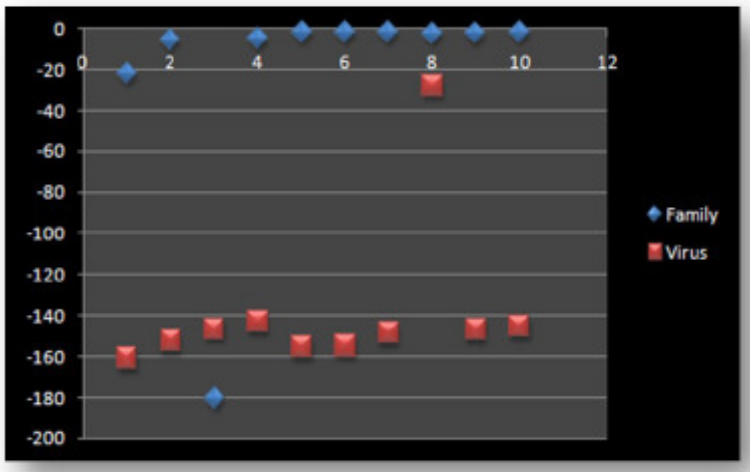


Figure 13: Family viruses and 9th generation viruses tested against normal model

HMM model of normal files has very low threshold. The reason behind this low threshold is less similarity within a set of normal files. With less similarity, generating most probable model is difficult. And this is causing false positives.

7. CONCLUSION

The developed the metamorphic engine producing morphed copies of the base virus that are highly dissimilar and includes some opcodes of the normal program. These were the two main criteria described in which are required in metamorphic virus to defeat HMM. In our new engine, employ code obfuscation techniques such as equivalent instruction substitution, dead code insertion, and transpose. This Paper introduce floating point opcodes in morphed copies which are commonly found in normal programs. The similarity showed that the morphed copies are highly metamorphic with 2.5% similarity index. Even with such a high metamorphism, HMM was able to classify the morphed copies of the base virus as the family virus. The base virus was compared with model of morphed copies, HMM was still able to classify the base virus as the same family. This fact proves that even with high metamorphism, HMM is able to identify a common statistical pattern across all morphed copies and the base virus. HMM has proved very difficult to defeat.

8. FUTURE WORK

This trained our models on disassembled virus executables. The disassembling process can take some time and the results depend on the quality of the disassembler. To speed up virus pre-processing and to eliminate the reliance on a particular disassembler, could attempt to train the HMMs directly on the binary code of the viruses. Other machine learning techniques, such as data mining or neural networks, might also work directly on the binaries. Training on raw executable byte sequences is more challenging as these byte sequences are longer and contain more irrelevant parts. To more thoroughly evaluate the performance of the HMM approach, it would be useful to test on a larger set of virus variants and also test on different types of viruses. Ideally, would like to find viruses that are similar to normal programs to a degree that the similarity index alone cannot distinguish the viruses from

normal code. Only with such data can we evaluate the effectiveness of the HMM approach to detecting metamorphic viruses. However, it appears that no metamorphic kit available today is capable of producing such challenging viral code.

REFERENCES

[1] Leonard Adleman. An abstract theory of computer viruses. In *Lecture Notes in Computer Science, vol 403*. Springer-Verlag, 1990.

[2] M. Stamp, "Information Security: Principles and Practice," August 2005.

[3] Fred Cohen. *Computer Viruses*. PhD thesis, University of Southern California, 1985.

[4] Peter J. Denning, editor. *Computers Under Attack: Intruders, Worms and Viruses*. ACMPress (Addison-Wesley), 1990.

[5] Christopher V. Feudo. *The Computer VirusDesk Reference*. Business One Irwin, Homewood, IL, 1992.

[6] Harold Joseph Highland, editor. *Computer Virus Handbook*. Elsevier Advanced Technology, 1990.

[7] J. Aycock, "Computer Viruses and malware," Springer Science+Business Media, 2006.

[8] E. Daoud and I. Jebril, "Computer Virus Strategies and Detection Methods," Int. J. Open Problems Compt. Math., Vol. 1, No. 2, September 2008.

http://www.emis.de/journals/IJOPCM/files/IJOPCM(vol.1.2.3.S.08).pdf

[9] Lance J.Hoffman, editor. *Rogue Programs:Viruses,Worms, and Trojan Horses*. VanNostrand Reinhold, New York, NY, 1990.

[10] Jan Hruska. *Computer Viruses and Anti-VirusWarfare*. Ellis Horwood, Chichester, England, 1990.

[11] Filiol, E., G. Jacob, M.L. Liard, 2007. Evaluation methodology and theoretical model for antiviral behavioral detection strategies. J. Comput. Virol., 3(1): 27-37

[12] Ye, Y., D. Wang, T. Li and D. Ye, 2008. An intelligent pe-malware detection system based on association mining. In Journal in Computer Virology,

[13] Zakorzhevsky, 2011. Monthly Malware Statistics. Available from:

http://www.securelist.com/en/analysis/204792182/Monthly_Malware_Statistics_June_ [Accessed 2 July.

[14] W. Wong, "Analysis and Detection of Metamorphic Computer Viruses," Master's thesis, San Jose State University, 2006. http://www.cs.sjsu.edu/faculty/stamp/students/Report.pdf

[15] Eugene H. Spafford. Computer viruses. In John Marciniak, editor, *Encyclopedia of Software Engineering*. JohnWiley & Sons, 1994.

[16] S. Attaluri, "Profile hidden Markov models for metamorphic virus analysis," Master's thesis, San Jose State University, 2007.

http://www.cs.sjsu.edu/faculty/stamp/students/Srilatha_cs298Report.pdf

[17] P. Szor, "The Art of Computer Virus Defense and Research," Symantec Press 2005. [18] VX Heavens, http://vx.netlux.org/

[18] Orr, "The viral Darwinism of W32.Evol: An in-depth analysis of a metamorphic engine," 2006. http://www.antilife.org/files/Evol.pdf

[19] E. Konstantinou, "Metamorphic Virus: Analysis and Detection," January 2008. [21] A. Venkatesan, "Code Obfuscation and Metamoprhic Virus Detection," Master's thesis, San Jose State University, 2008.

http://www.cs.sjsu.edu/faculty/stamp/students/ashwini_venkatesan_cs298report.doc

[20] The Mental Driller, "Metamorphism in practice or How I made MetaPHOR and what I've learnt," February 2002. http://vx.netlux.org/lib/vmd01.html

[21] M. Stamp, "*A Revealing Introduction to Hidden Markov Models*", January 2004. http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf

[22] Walenstein, R. Mathur, M. Chouchane R. Chouchane, and A. Lakhotia, "The design space of metamorphic malware," In Proceedings of the 2nd International Conference on Information Warfare, March 2007.

[23] "Benny/29A", Theme: metamorphism, http://www.vx.netlux.org/lib/static/vdat/epmetam2.htm

[24] J. Borello and L. Me, "Code Obfuscation Techniques for Metamorphic Viruses", Feb 2008, http://www.springerlink.com/content/233883w3r2652537

[25] A. Lakhotia, "Are metamorphic viruses really invincible?" Virus Bulletin, December 2005.

[26] JDB. EstiHMM: een efficiënt algoritme ter bepaling van de maximale sequenties in een imprecies hidden Markovmodel. Master Thesis at Ghent University. Supervised by GdC. June 2011.

[27] JDB & GdC. State sequence prediction in imprecise hidden Markov models. ISIPTA '11: pp. 159-168. July 2011.

Author Details

The First author is Nitesh Kumar Dixit, he is currently working as Assistance Professor, in BIET, Sikar. He has obtained his M.tech (Embedded System) degree from SRM university, Chennai and B.E. (ECE) from SEC, Dundlod (jhujhunu). His area of interest is Cryptography, Image Processing, Embedded System etc.

The Second author is Lokesh Mishra, he is currently working as Project Engineer, in NETCOM, Sikar. He has obtained his M.C.A. degree from Rajasthan university, Jaipur and B.Sc. from Rajasthan University, jaipur. His area of interest is Cryptography, Network Security, System Calls etc.

The Third author is Mahendra Singh Charan, he is currently working as Lecturer in B.I.E.T., Sikar. He is Pursuing M.tech (CS) from Gyan Vihar university, Jaipur and obtained B.E. from S.I.T. ,Rajasthan University, jaipur. His area of interest is Advanced Data Structure, Graphics Design and Network Security etc.

The Fourth author is Bhabesh Kumar Dey, he is currently working as Lecturer in B.I.E.T., Sikar. He is Pursuing M.tech (CS) from Rajasthan Technical University, Jaipur and obtained B.E. from S.I.T., Rajasthan University, jaipur. His area of interest is Computer Security and Networking, Advanced Programming in JAVA, Linux Operating System etc.