Opcode Graph Similarity and Metamorphic Detection

Neha Runwal[*] Richard M. Low[†] Mark Stamp[‡]

Abstract

In this paper, we consider a method for computing the similarity of executable files, based on opcode graphs. We apply this technique to the challenging problem of metamorphic malware detection and compare the results to previous work based on hidden Markov models. In addition, we analyze the effect of various morphing techniques on the success of our proposed opcode graph-based detection scheme.

Keywords: malware, metamorphic detection, hidden Markov model, similarity detection, opcode graph, directed graph

1 Introduction

Practical detection of metamorphic malware is a difficult challenge. A detection technique using a hidden Markov model trained on opcode sequences was studied in [33]. While this approach was highly successful at detecting hacker-produced metamorphic malware, in [17] it was shown that the detector can be defeated by a properly designed metamorphic generator.

The paper [1] discusses a malware detection strategy based on weighted directed graphs derived from opcode sequences. The authors of [1] show that their technique is effective on many types of malware, including polymorphic viruses. However, metamorphic viruses are not considered.

In this paper, we develop and analyze a technique that is based on the same opcode graphs, but our overall approach is simpler and more efficient than that in [1]. In addition, we apply our technique to the challenging problem of metamorphic malware detection.

Various similarity measures have previously been applied to the metamorphic detection problem [21, 33]. In addition, control flow graph analysis has been considered [5, 8]. But such graph techniques often lead to difficult graph isomorphism problems. Simpler call graph techniques have also been studied in the context of

[*] Department of Computer Science, San Jose State University

[†] Department of Mathematics, San Jose State University

[‡] Department of Computer Science, San Jose State University: stamp@cs.sjsu.edu

metamorphic detection [16]. Yet another example of a graph-based malware detection strategy can be found in [4]. However, there does not appear to be any previous work involving similarity techniques or graph analysis for metamorphic detection that is analogous to the approach we discuss here.

The remainder of the paper is organized as follows. In Section 2, we briefly discuss relevant background material related to malware and malware detection, including a brief overview of hidden Markov models (HMMs) and other similarity-based detection methods. We also outline the graph-based malware detection method developed in [1]. Section 3 covers the design and implementation of our graph based similarity measure and its relevance to metamorphic detection. In Section 4, we present experimental results showing the potential utility of our technique for metamorphic detection. Then in Section 5, we analyze the effectiveness of morphing strategies that a virus writer might employ in an effort to evade our graph-based detection. Finally, Section 6 gives our conclusions and mentions some directions for future work.

2 Background

In this section, we briefly discuss several topics that are relevant to the remainder of the paper. First, we consider malware detection in general, with the emphasis on metamorphic malware. Then we we discuss hidden Markov models and their potential role in malware detection. We also briefly mention a few other opcode-based similarity measures that have been applied to the metamorphic detection problem. Finally, we turn our attention to the graph-based detection technique in [1].

2.1 Malware detection

Malware is software "whose intent is malicious, or whose effect is malicious" [3]. Examples of malicious activity include damaging data, stealing information, or stealing computing resources. Increasingly, malware appears to serve as a source of revenue for the malware writer [19].

The purpose of anti-virus (AV) software is to detect and, ideally, remove malware. Next, we briefly discuss a few types of malware in the context of the ongoing "arms race" between malware writers and malware detection. Here, we only provide a very brief overview with the aim of highlightign the development of metamorphic malware. Note that throughout this paper, we use the terms virus and malware interchangeably.

Since the dawn of time,[1] signature scanning has been the most widely used AV technique [3]. A virus signature can be as simple as a string of bits (possibly, including wildcards) that appear in a particular piece of malware. Of course, these bits represent instructions and/or data in the particular virus. Ideally, the signature does not occur in any other software. However, once a signature has been detected, additional

[1] Or, at least since the development of the first AV systems.

testing is likely required to be certain that the code is indeed the indicated malware. Signature scanning is efficient and effective against common types of malware; thus its popularity.

Of course, virus writers are aware that signature scanning is the most prevalent form of AV scanning. Consequently, virus writers have developed various techniques designed to help their creations evade signature detection. As a first line of defense against signature detection, a virus can be encrypted or "packed" (i.e., compressed). Here, we consider encryption, but similar comments hold for packed code. The advantage of encryption is that a signature that is present in the unencrypted code will not appear in the encrypted version. Furthermore, different keys will yield different ciphertext, so AV software cannot directly scan for a signature in the ciphertext. From the virus writers perspective, the disadvantage of encryption is that the code must be decrypted before it can execute, and the decryption code itself cannot be encrypted. Consequently, AV software can scan for the signature of the decryption code.

Polymorphic malware is the next logical step in the arms race between signature-based detection and malware writers. As with encrypted malware, polymorphic malware is encrypted. However, polymorphic malware adds a new twist—the decryption code is morphed and consequently there is no fixed decryption code to search for in a scan [18]. Ideally, new decryptor code would be generated for each infection. Robust detection of polymorphic malware is difficult. One approach is to let the code decrypt itself (via emulation), then scan for a signature in the decrypted code [3, 10].

Metamorphic malware is sometimes said to be "body polymorphic." That is, instead of encrypting the virus body and morphing the decryptor (as in polymorphic malware), metamorphic code does away with encryption and morphs the entire virus body [15, 29, 31]. Note that the function of the code remains the same, but the internal structure of the code changes. If the morphing is sufficiently thorough, no common signature will exist and therefore, no encryption is necessary. It could be argued that well designed metamorphic code presents the ultimate challenge in malware detection.

A wide variety of techniques can be employed to create metamorphic software. Such techniques include register swapping, general code obfuscation, equivalent instruction substitution, code shuffling, and subroutine permutation [15]. Predictably, virus writers have developed metamorphic "engines" that can be used by the unskilled to create metamorphic malware. Often, existing malware is morphed to create equivalent code that can evade signature detection. Examples of metamorphic engines include the Next Generation Virus Construction Kit (NGVCK), Phalcon/Skism Mass-Produced Code generator (PS-MPC), Second Generation virus generator (G2), and the Mass Code Generator (MPCGEN) [32]. According to [33], NGVCK is by far the most effective of these at creating highly metamorphic code. Therefore, we focus our attention on NGVCK [28].

Metamorphic detection is a challenging research problem. In [33] hidden Markov models (HMMs) were used to successfully detect highly morphed NGVCK viruses.

However, in [17] a metamorphic generator was developed that can successfully evade the HMM-based detector in [33]. Therefore, research into more robust detection mechanisms is warranted.

The immediate motivation for the research presented here is the paper [1], where an interesting graph-based technique was applied to the malware detection problem. Our goal is to analyze a similar (albeit simpler) opcode graph-based detection algorithm in the context of metamorphic detection. We also compare the results obtained using our proposed technique to previous research involving HMM-based detection [17, 33].

In the next subsection, we first discuss HMMs in general. We then consider the use of HMMs for malware detection.

2.2 Hidden Markov models

Hidden Markov models can be viewed as a machine learning technique and as a discrete hill climb [30]. An HMM is a machine learning technique in the sense that the user only needs to specify some basic parameters. The HMM can then be trained on a given data set and, if successful, the trained model can be used to detect similar data. That is, data can be scored against a trained model, with higher scores indicating a higher degree of similarity to the training data.

The HMM training process can be viewed as a discrete hill climb on a the HMM parameter space. Training is an iterative process, and during the process, the parameters of the HMM "climb" towards an optimal model.

HMMs have been successfully applied in many fields, including language analysis [30], speech recognition [23], and metamorphic virus detection [33]. Here we present a classic example that illustrates the strength of HMMs. Specifically, we train an HMM on English text, as discussed in [30]. But first, we require some notation.

A generic view of an HMM is given in Figure 1. We assume that there is a Markov process that is "hidden," that is, the Markov process is not directly observable. In Figure 1, the states of the hidden Markov process are denoted X_i. For each state X_i of the underlying Markov process, we observe $\mathcal{O}_i$. As with any standard Markov process, the matrix A "drives" the hidden Markov process. The unique feature of a hidden Markov model is that the states are hidden. But we do have some information on the hidden states via the series of observations (the $\mathcal{O}_i$), since these observations are probabilistically related to the hidden states by the matrix B.

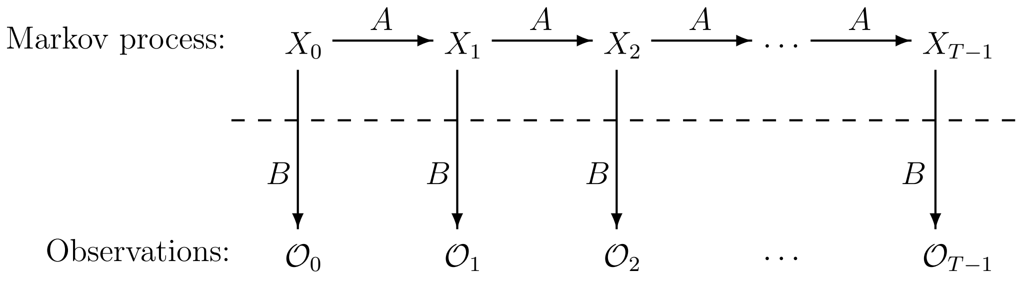

Figure 1: Generic Hidden Markov Model [30]

For our discussion of HMMs, we use the following notation:

$$\begin{aligned}
T &= \text{length of the observation sequence}\\
N &= \text{number of states in the hidden Markov process}\\
M &= \text{number of distinct observation symbols}\\
A &= \text{state transition probabilities}\\
B &= \text{observation probability matrix}\\
\pi &= \text{initial state distribution}\\
X &= (X_0, X_1, \ldots, X_{T-1}) = \text{hidden states of Markov process}\\
\mathcal{O} &= (\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{T-1}) = \text{observation sequence.}
\end{aligned}$$

The matrices A, B and π are row stochastic, that is, the elements of each row form a discrete probability distribution [24].

To illustrate the strengths of HMMs, consider the following experiment [30]. We are given a large body of English text and we remove all punctuation, numbers, special characters, etc., and convert all upper-case letters to lower-case. The text then has 27 distinct symbols, consisting of the 26 letters and space. We train an HMM with this text serving as the observation sequence.

Suppose that we select $N = 2$, that is, we assume there are two hidden states. Then $M = 27$ and the matrix π is 1×2, the matrix A is 2×2, while the matrix B is 2×27. The matrices are initialized so that each element of π and each element of A are approximately $1/N$ and each element of B is approximately $1/M$ subject to the constraint that each row sum must be 1. For technical reasons, we cannot initialize the matrices to uniform values [30].

In this case, the HMM training requires about $T = 50{,}000$ observations to converge. The training algorithm is described in detail in [30].

After training, for this example we find

$$\pi = \begin{bmatrix} 0.00000 & 1.00000 \end{bmatrix}$$

and

$$A = \begin{bmatrix} 0.25633 & 0.74367 \\ 0.71195 & 0.28805 \end{bmatrix}.$$

The π matrix has converged to the initial state probabilities, while the A matrix gives the transition probabilities between the two hidden states. Neither of these matrices tell us anything about what the hidden states might represent. However, the converged B matrix, does provide some interesting information concerning the hidden states. For this particular example, the converged B matrix appears in Table 1. Note that the matrix in Table 1 is actually the transpose of the B matrix.

Table 1: Trained HMM for English Text

Final B matrix (transpose)		
a	0.13956	0.00000
b	0.00000	0.02306
c	0.00000	0.05661
d	0.00000	0.06925
e	0.21460	0.00000
f	0.00000	0.03547
g	0.00016	0.02780
h	0.00000	0.07321
i	0.12308	0.00000
j	0.00000	0.00364
k	0.00177	0.00708
l	0.00000	0.07258
m	0.00000	0.03880
n	0.00000	0.11439
o	0.13184	0.00000
p	0.00000	0.03703
q	0.00000	0.00153
r	0.00000	0.10202
s	0.00000	0.11024
t	0.00971	0.14483
u	0.04514	0.00000
v	0.00000	0.01617
w	0.00000	0.02298
x	0.00000	0.00446
y	0.00000	0.02599
z	0.00000	0.00110
space	0.33413	0.01178

A careful examination of the B matrix in Table 1 reveals that the two hidden states essentially correspond to consonants and vowels. Note that no a priori assumption was made regarding the nature of the hidden states—we simply chose $N = 2$, which specified the number of hidden states. The training process proceeded to automatically extract a fundamental property of English from the training data. This example illustrates the strength of HMMs as a machine learning technique.

For more information on HMMs in general, the paper [30] is a readable and reasonably thorough introduction. Another standard reference is [23].

2.3 HMM-based metamorphic detection

In [33], HMMs were used for metamorphic malware detection. First, several virus generating kits that claimed to produce metamorphic code were studied. Of these, only the Next Generation Virus Construction Kit (NGVCK) was found to produce highly metamorphic code. Consequently, the paper [33] focuses on detecting NGVCK viruses.

A large number of experiments involving HMMs were conducted in [33], with a typical result reproduced here in Figure 2. For this example, the authors of [33] trained an HMM (with two hidden states) on a sequence of opcodes extracted from 160 NGVCK variants. The test results given in Figure 2 were obtained using 40 NGVCK "family" viruses and 40 "normal" (or benign) files, along with a smaller number of "non-family" viruses. The 40 NGVCK test files were not among those used for training. Also, Cygwyn utility files were used as the benign files, and the non-family viruses consisted of non-NGVCK viruses. The scores are given in the form of log likelihood per opcode (LLPO) that is, the scores are normalized per opcode so that files of different lengths can be compared.

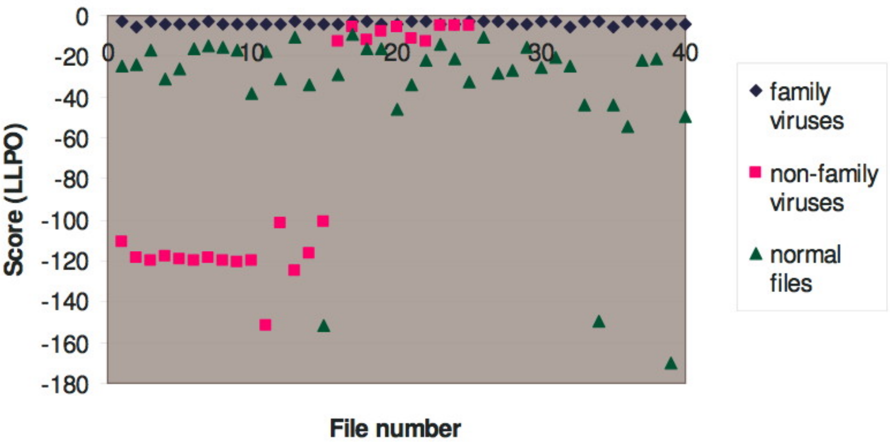

Figure 2: HMM detection results from [33]

In Figure 2, we could easily set a threshold that would provide complete separation between the family viruses and the benign files. That is, we can distinguish between the NGVCK family viruses and the benign files using an HMM trained on opcode sequences. The only misclassifications come from the set of non-family viruses, some of which would be classified as family viruses, and some of which would be classified as benign. The fact that some non-family viruses are also detected could be considered a feature—as opposed to a flaw—of the technique.

In the paper [17] a metamorphic generator was developed for the purpose of defeating the HMM-based detection presented in [33]. The metamorphic generator in [17] relies primarily on inserting dead code extracted from normal files. In effect, this makes the resulting viruses look more like benign files, thereby making it more difficult to distinguish the viruses from the benign files.

Figure 3 gives typical test results from the paper [17]. For these results, the same procedure as in [33], was used, that is, an HMM was trained on extracted opcode sequences, and viruses and benign files were scored using the resulting model.

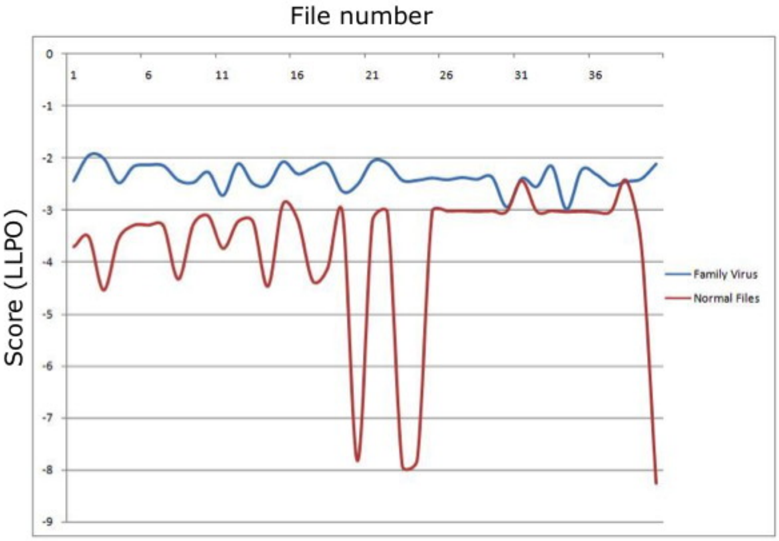

Figure 3: HMM detection defeated [17]

The results in Figure 3 show that an HMM-based detector can be defeated by a properly constructed metamorphic generator. Consequently, there is a need to consider additional detection techniques for metamorphic malware. In this paper, we consider an approach based on opcode graphs and compare it to the HMM-based technique discussed in this section.

2.4 Similarity and metamorphic detection

Software similarity is a potentially useful means for detecting metamorphic malware. If we can determine a characteristic common to all members of a metamorphic family, then we can potentially use this characteristic to detect metamorphic code belonging to this family. Note that such an approach is not limited to metamorphic malware, but it is one of the few viable options available when dealing with well-designed metamorphic code.

Various opcode-based similarity measures have been proposed specifically for metamorphic detection, including n-gram similarity [33], chi-squared (statistical) similarity [11], edit distance [21], and pairwise sequence alignment [2, 21]. In addition, opcode-based machine learning techniques have been applied to the metamorphic detection problem—examples include hidden Markov models (as discussed above in Section 2.3) and profile hidden Markov models [2]. These techniques are aimed at deriving a model that captures the similarity of a given metamorphic family. Yet another approach to opcode-based metamorphic detection is to employ data mining techniques, such as cosine similarity [14]. In this paper, we consider a metamorphic detection strategy based on opcode graph similarity.

2.5 Opcode graph similarity

The paper [1] proposes an interesting graph-based technique for virus detection. Given an executable file, the sequence of opcodes is extracted and a weighted directed graph is constructed as follows. Each distinct opcode that appears in the program opcode sequence is a node in directed graph. A directed edge is inserted from a node to each possible successor node, that is, each successor opcode. Edge weights give the probability of the corresponding successor node. To illustrate the process, consider the program trace in Table 2, where some lines have been abbreviated to save space (as indicated by "...").

From the program in Table 2, we extract the opcode sequence and tabulate counts for pairs of consecutive opcodes. That is, we are interested in opcode digram frequencies. For this particular program, we obtain the counts in Table 3. For example, in the sample program, the opcode `MOV` is immediately followed by the opcode `CALL` three times (see lines `17`, `35`, and `43` in Table 2). Consequently, there is a 3 in the `MOV` row and `CALL` column in Table 3.

Using the digram frequency counts in Table 3, we convert the counts to probabilities by dividing the count in each cell by its corresponding row sum. The resulting matrix appears in Table 4. For example, `MOV` occurs 18 times, while (`MOV`, `CALL`) occurs 3 times. Therefore, the (`MOV`, `CALL`) cell in Table 4 contains the probability $3/18 = 1/6$.

The entries in Table 4 give the corresponding edge weights in the opcode directed graph. The opcode graph for the program in Table 2 appears in Figure 4. However, for all computations involving the opcode graph, we use the array of edge weights. That

Table 2: Assembly language instruction trace

1	PUSH	ebp	24	MOV	ebp, esp
2	MOV	ebp, esp	25	PUSH	edi
3	SUB	esp, 8	26	PUSH	esi
4	AND	esp, 0FFFFFFF0h	27	PUSH	ebx
5	MOV	eax, ds:dword_404000	28	SUB	esp, 7Ch
6	TEST	eax, eax	29	MOV	edi, [ebp+arg_0]
7	JZ	Short loc_401013	30	MOV	esi, [ebp+arg_4]
8	INT	3	31	AND	esp, 0FFFFFFF0h
9	FNSTCW	[ebp+var_2]	32	CALL	sub_401930
10	MOVZX	eax, [ebp+var_2]	33	CALL	main
11	AND	eax, 0FFFFF0C0h	34	MOV	[ebp+var_4C], 0
12	MOV	[ebp+var_2], ax	35	MOV	[esp+88h+var_88], ...
13	MOVZX	eax, [ebp+var_2]	36	CALL	CORBA_exception_init
14	OR	eax, 33Fh	37	MOV	dword ptr ...
15	MOV	[ebp+var_2], ax	38	XOR	edx, edx
16	FLDCW	[ebp+var_2]	39	MOV	eax, offset ...
17	MOV	[esp+8+var_8], ...	40	MOV	[esp+88h+var_78], edx
18	CALL	sub_401960	41	MOV	[esp+88h+var_7C], eax
19	LEAVE		42	MOV	dword ptr ...
20	RETN		43	MOV	[esp+88h+var_88], ...
21	ALIGN	10h	44	CALL	poptGetContext
22	PUSH	ebp	45	MOV	ebx, eax
23	MOV	eax, 10h	46	LEA	esi, [esi+0]

is, we use the digram probability array in Table 4, not the graphical representation in Figure 4. This opcode graph is the basis for the graph scoring technique in [1] as well as the metamorphic similarity measure discussed in this paper.

Next, we give a very brief summary of the opcode graph technique in [1]. First, the opcode sequence is extracted and the weighted directed opcode graph is constructed, as discussed above. Graph kernel techniques [9] are then used to obtain a score corresponding to the graph. Finally, a support vector machine (SVM) is used for classification. That is, based on training data, an SVM determines whether a given score corresponds to a virus or a benign file.

The authors of [1] test their approach on a large sample of known malware and benign files, and they compare their classification results with popular anti-virus software. Some of the results from [1] are summarized here in Table 5. These results are based on 615 instances of benign software and 1615 instances of malware. The graph kernel technique is clearly superior to a standard n-gram model and, in the sense of overall accuracy, it is far superior to all of the AV software tested. However, it is apparent from these results that AV software companies designed their products so as to avoid false positives at all cost—even if that cost includes a large number of

Table 3: Assembly language counts for Table 2

	PUSH	MOV	SUB	AND	TEST	JZ	INT	FNSTCW	MOVZX	OR	FLDCW	CALL	LEAVE	RETN	ALIGN	XOR	LEA
PUSH	2	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV	1	7	1	1	1	0	0	0	1	0	1	3	0	0	0	1	1
SUB	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
AND	0	2	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
TEST	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
JZ	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
INT	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
FNSTCW	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
MOVZX	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
OR	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
FLDCW	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CALL	0	3	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
LEAVE	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
RETN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
ALIGN	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
XOR	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LEA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

false negatives. While interesting in their own right, direct comparisons of research techniques to AV software may not be particularly informative, since any AV software could certainly improve its accuracy by accepting more false positives. In any case, the opcode graph technique yields impressive results.

Our proposed graph technique is similar to that in [1], but simpler, and hence more efficient. In addition, we apply our technique to metamorphic detection and compare the results to previous related work. In the next section, we discuss the design of our graph technique in detail.

3 Opcode graph-based similarity

Here, we consider a graph technique base on the same opcode graph used in the research presented in [1]. However our technique is considerably simpler and somewhat more efficient, and we are focused on the problem of metamorphic detection. Our goal is to develop a similarity measure—based on extracted opcode sequences—that can be used to compare executable files. In Section 4 we apply this similarity measure to the problem of metamorphic detection.

Table 4: Probabilities from Table 3

	PUSH	MOV	SUB	AND	TEST	JZ	INT	FNSTCW	MOVZX	OR	FLDCW	CALL	LEAVE	RETN	ALIGN	XOR	LEA
PUSH	$\frac{2}{5}$	$\frac{2}{5}$	$\frac{1}{5}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOV	$\frac{1}{18}$	$\frac{7}{18}$	$\frac{1}{18}$	$\frac{1}{18}$	$\frac{1}{18}$	0	0	0	$\frac{1}{18}$	0	$\frac{1}{18}$	$\frac{1}{6}$	0	0	0	$\frac{1}{18}$	$\frac{1}{18}$
SUB	0	$\frac{1}{2}$	0	$\frac{1}{2}$	0	0	0	0	0	0	0	0	0	0	0	0	0
AND	0	$\frac{2}{3}$	0	0	0	0	0	0	0	0	$\frac{1}{3}$	0	0	0	0	0	0
TEST	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
JZ	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
INT	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
FNSTCW	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
MOVZX	0	0	$\frac{1}{2}$	0	0	0	0	0	0	$\frac{1}{2}$	0	0	0	0	0	0	0
OR	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
FLDCW	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CALL	0	$\frac{3}{5}$	0	0	0	0	0	0	0	0	$\frac{1}{5}$	$\frac{1}{5}$	0	0	0	0	0
LEAVE	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
RETN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
ALIGN	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
XOR	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LEA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5: Summary of results from [1]

technique	accuracy	false positives	false negatives
Graph kernel	96.41	47	33
n-gram	82.15	300	98
AV0	73.32	0	595
AV1	53.86	1	1028
AV2	49.60	0	1196
AV3	43.27	1	1264
AV4	42.96	1	1271

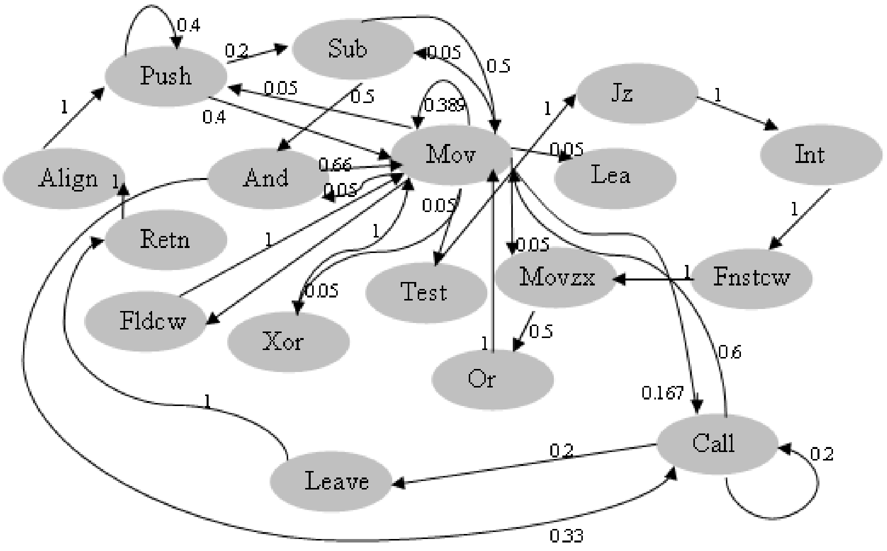

Figure 4: Weighted directed graph for code in Table 2

As discussed in Section 2.5, given an executable file, we extract the opcode sequence and generate a weighted directed graph based on opcode digrams. To this point, our technique parallels the approach used in [1]. But, instead of using graph kernels to generate scores and SVMs for classifications, we directly compare the opcode graphs.

Let N be the number of distinct opcodes under consideration and map the opcodes to $\{0, 1, 2, \ldots, N-1\}$. Let $A = \{a_{ij}\}$ and $B = \{b_{ij}\}$ be the edge-weight matrices corresponding to executable files. Recall that an example of such a matrix appears in Table 4. Note that both A and B are $N \times N$ and the opcode numbering is the same for both. That is, a_{ij} and b_{ij} represent the probability that opcode i is followed by opcode j in programs A and B, respectively.

Now, to compare these matrices, we compute the score

$$\texttt{score}(A,B) = \frac{1}{N^2}\Bigg(\sum_{i,j=0}^{N-1} |a_{ij} - b_{ij}|\Bigg)^2. \tag{1}$$

If $A = B$, then the minimum score, namely, $\texttt{score} = 0$, is achieved. Suppose that $a_{ij} = 1$ and $b_{ik} = 1$, with $j \neq k$. Then we obtain the maximum possible row sum,

$$\sum_{j=0}^{N-1} |a_{ij} - b_{ij}| = 2.$$

If this maximum row sum is achieved for each row, then we obtain the maximum possible score of 4. Consequently,

$$0 \leq \texttt{score}(A, B) \leq 4$$

for all A and B.

Other scoring functions were tested, but none proved superior to (1) for our purposes. In addition, various graph comparison techniques were considered, such as those in [12, 20], but most were costly to compute and none offered a clear advantage to the straightforward calculation in (1).

To use the score function in (1) for metamorphic detection, we first need to determine a score threshold. To do so, the following process is used:

1. Determine the opcode graphs for a collection of metamorphic family viruses.
2. Determine the opcode graphs for a representative sample of benign files.
3. Using (1), compute the scores for all pairs of metamorphic family viruses from step 1.
4. Using (1), compute the scores for all pairs consisting of one family virus from step 1 and one benign file from step 2.
5. Set a threshold based on the scores in steps 3 and 4.

Once we have set a threshold, we can use any randomly selected metamorphic file from the set in step 1 for scoring. That is, given a file that we want to score, we first determine its opcode graph, then score the resulting graph against the opcode graph from a known metamorphic file. If the resulting score is below our threshold, we classify the file as belonging to the metamorphic family; otherwise it is classified as benign. Figure 5 shows the flow of our graph technique implementation.

4 Results

Our test set of metamorphic viruses consists of 200 NGVKC files [32]. That is, all of our metamorphic viruses belongs to the NGVCK family. In [33], these viruses are shown to be the most highly metamorphic of any of the virus construction kits tested. In addition, a wide variety of metamorphic detection techniques have been applied to this set of viruses [2, 7, 11, 21, 22, 27, 33], so we have a basis for comparing the effectiveness of our technique to previous work.

Our set of benign files consists of 41 cygwin utility files [6]. The cygwin utility files were used as representative benign files in several previous studies, including [33]. Finally, we also consider a third set containing 25 non-family virus files.

The results in [33], show that this set of metamorphic files can be distinguished from the benign files using an HMM-based technique. Our initial tests are aimed are determining whether our graph-based technique is competitive with HMM-based detection.

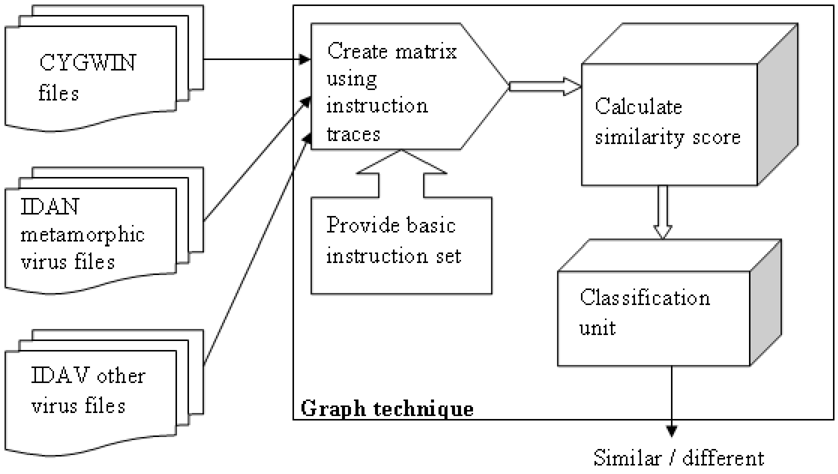
CYGWIN files
IDAN metamorphic virus files
IDAV other virus files
Create matrix using instruction traces
Provide basic instruction set
Calculate similarity score
Classification unit
Graph technique
Similar / different

Figure 5: Flow of the graph technique

Using the approach outlined in Section 3 we determine a threshold. The viability of our proposed technique is dependent on there being a useful separation between scores for the following two cases:

- Metamorphic virus versus metamorphic virus
- Benign file versus metamorphic virus

Although not strictly necessary for malware detection, we also consider the following cases:

- Benign file versus non-family virus
- Benign file versus benign file

The graphs in Figures 6 and 7 give typical results corresponding to the first two case listed above, that is, the "metamorphic versus metamorphic" case and the "benign versus metamorphic" case, respectively. Note that the highest score in Figures 6 is 0.525 while the lowest score in Figures 7 is 0.588, and hence we have significant separation as required for ideal detection. These results show that our graph-based detection technique is on par with the HMM-based approach in [33], which also yielded no false positives or false negatives.

Figure 8 gives results for the case where benign files are scored against other benign files. As mentioned above, this is not directly relevant for malware detection, but it does illuminate some properties of our similarity measure.

Note that the benign versus benign scores in Figure 8 lie within a similar range as the metamorphic versus metamorphic scores in Figure 6. These graphs tell us that, according to our graph similarity measure, NGVCK viruses are typically about as different from each other as any two randomly chosen benign files. While this

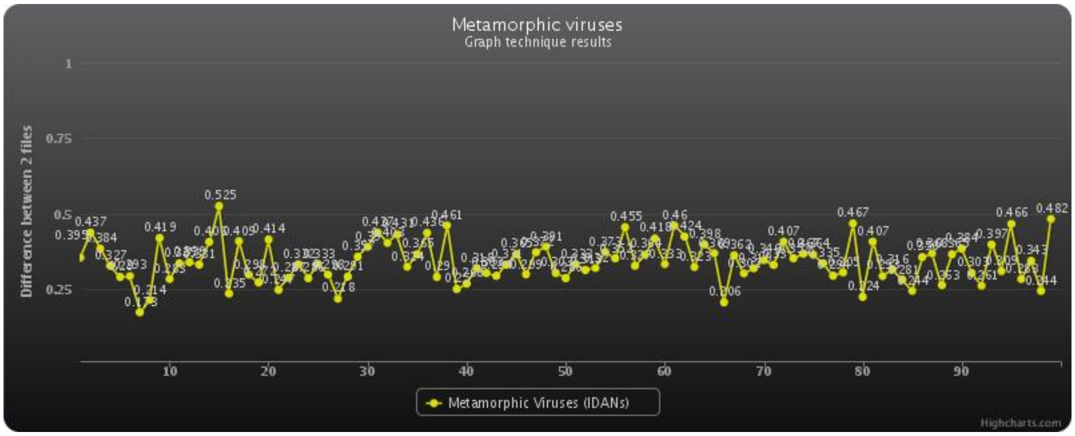

Figure 6: Similarity scores for metamorphic versus metamorphic

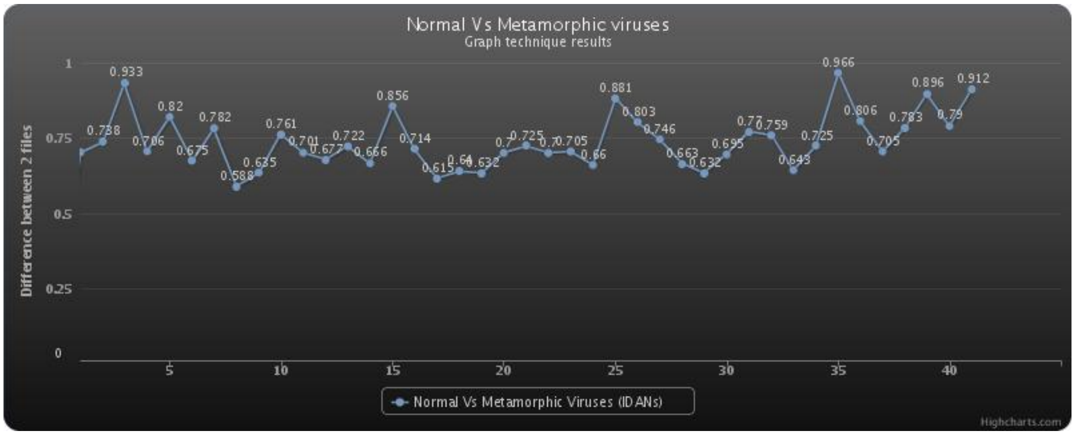

Figure 7: Similarity score for benign versus metamorphic

does indicates a high degree of metamorphism, it is actually considerably less than some previously studied similarity scores. In other words, according to previous similarity measures, NGVCK viruses are significantly more different from each other than benign files are different from each other. For example, for an n-gram based similarity measure analyzed in [33], pairs of NGVCK viruses are far more dissimilar than pairs of benign files. It could be considered a strength of our graph-based approach that the NGVCK viruses appear to be less metamorphic than with other similarity measures.

Figure 9 gives similarity results when benign files are compared to non-family viruses. The results are comparable to those in Figure 7 and indicate that our similarity measure is not restricted to metamorphic detection.

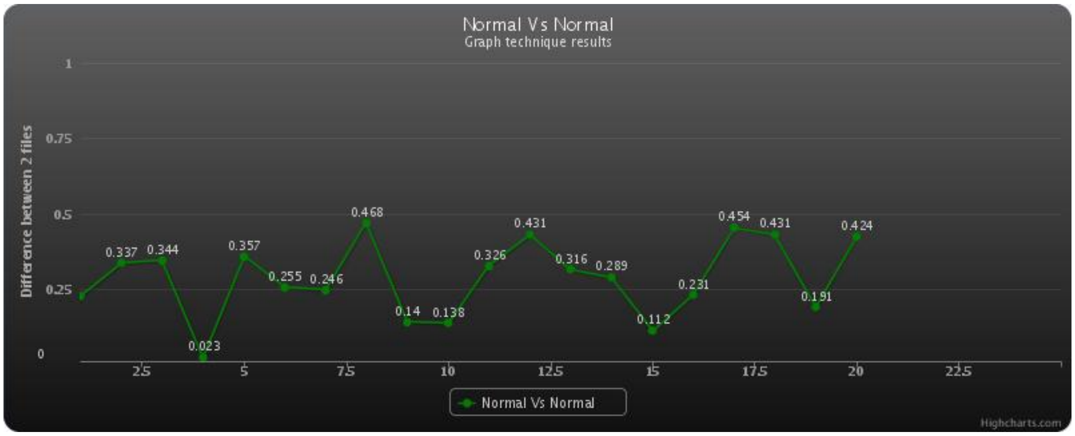

Figure 8: Similarity scores for benign versus benign

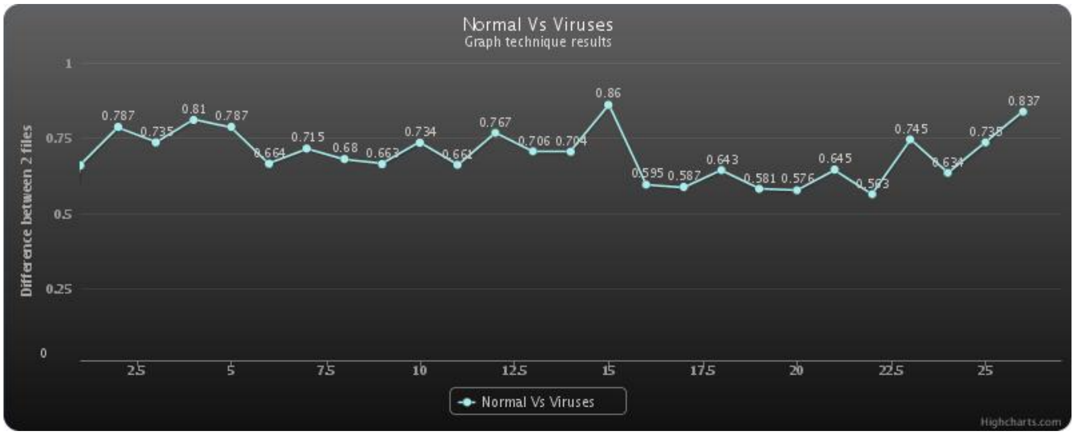

Figure 9: Similarity scores for benign versus non-family viruses

Figure 10 shows all four of the graphs from Figures 6, 7, 8, and 9 plotted on the same axes. Also, we note that many additional graphs of results can be found in [25].

5 Attacks on graph-based detection

In this section, we consider the robustness or our proposed metamorphic detection technique. In particular, we consider two different approaches that a metamorphic virus writer might follow to try to evade our detection technique. First, we consider the effect of removing uncommon opcodes from the malware. Second, we turn our attention to modifying the morphing technique so that the resulting malware is more

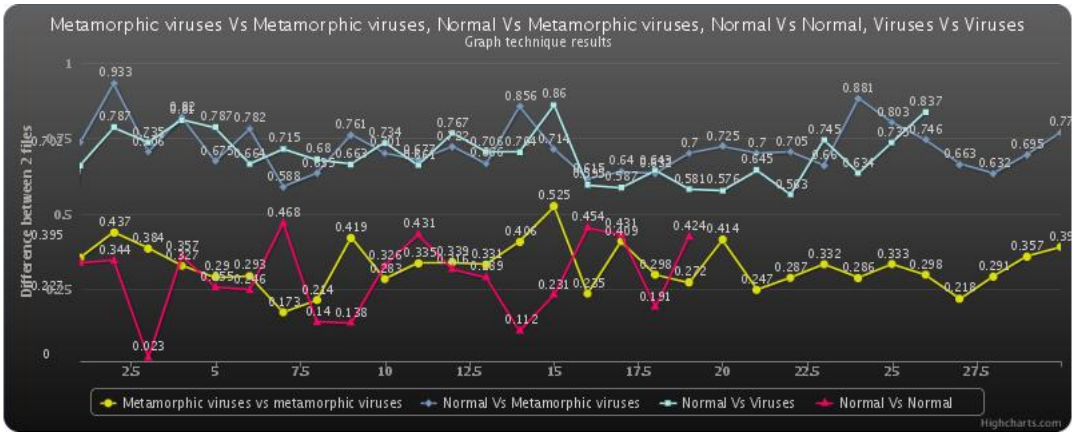

Figure 10: Combined graph

similar to benign files. This latter approach was used in [17] to effectively defeat the HMM-based detection.

5.1 Uncommon opcode removal

Consider again the opcode graph in Figure 4 or, equivalently, the opcode digram probability matrix in Table 4. By construction, the outgoing probabilities for each node sum to one. From the scoring formula in (1), we see that all opcodes are weighted the same so that, for example, MOV carries no more weight than INT, in spite of the fact that the former is typically the most common opcode, while the latter is rare.

Consequently, it could be argued that our graph-based similarity score gives excessive weight to uncommon opcodes. That is, it might seem that the score is little more than a glorified heuristic that detects malware based primarily on a relatively few rare opcodes that do not typically occur in benign code. If this is indeed the case, a virus writer who could remove these rare opcodes would evade detection by our opcode graph similarity score.

To test this hypothesis, we removed about 80% of all opcodes that appear in the virus files, but do not occur in the benign files. Figure 11 shows the before and after results—the upper line is the "metamorphic versus metamorphic" case before opcode removal (this is the same graph that appears in Figure 6) while the lower line shows the same case after these rare opcodes have been removed. The scores are only slightly weaker and Figure 12 confirms that even after removing the vast majority of rare opcodes, we still have clear separation between the "metamorphic versus metamorphic" and the "benign versus metamorphic" cases. That is, we still have an effective detection strategy based on opcode graph similarity.

Note that to remove the rare opcodes, we simply expunged the opcodes from the directed graph. However, a malware writer would have to modify the virus code so

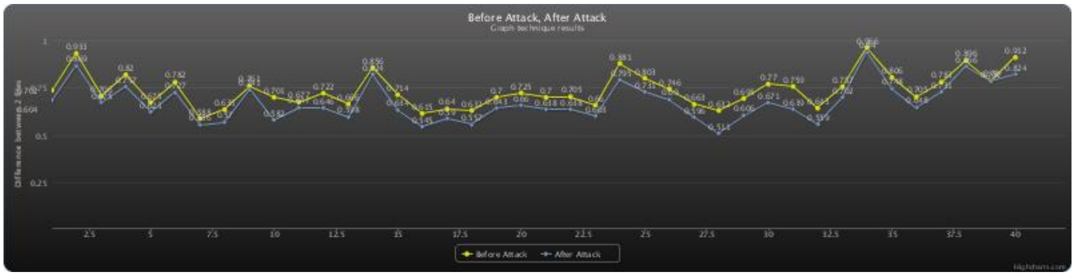

Figure 11: Metamorphic versus metamorphic: Before and after opcode removal

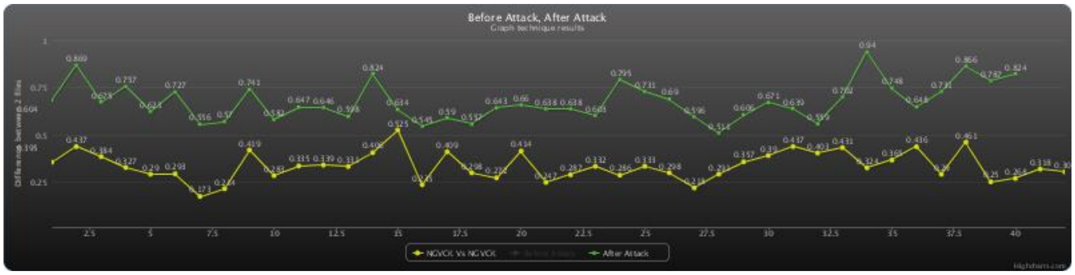

Figure 12: Metamorphic versus metamorphic and benign versus metamorphic after rare opcode removal

as to avoid these opcodes, which is a much more involved task. In any case, given the results in Figures 11 and 12, a virus writer appears to have little to gain from such an effort.

5.2 Modified morphing engine

In [17], a morphing engine was developed for the sole purpose of evading HMM-based detection. The morphing consists primarily of inserting code from benign files directly into malware files and the inserted code is largely "dead code," in the sense that it is not executed.

The paper [17] shows that inserting dead code from benign files into viruses can be an effective strategy for evading HMM-based detection. However, it is also shown that it is much more effective to insert the dead code in blocks, as opposed to having the dead code more evenly disbursed throughout the morphed virus.

We conducted similar experiments for our proposed graph-based similarity measure. Two types of morphing are used. In the first case, which we refer to as "block morphing," we insert the dead code as a single block. In the second case, which we refer to as "random morphing," we distribute the dead code approximately uniformly throughout the morphed virus. These strategies represent the extreme cases for inserting dead code. As in Section 4, for all experiments in this section, NGVCK viruses were selected as our base virus files.

We conducted experiments for both block morphing and random morphing with various dead code insertion percentages. These percentages are given in terms of the size of the original virus file. For example, if the source virus file has 100 lines, to morph it at 20% we extract 20 lines from a randomly-selected benign file and insert these lines into the virus file. Note that these steps all occur at the assembly code level, and we make no effort to create a functioning program. To actually use this technique, a virus writer would have to work much harder, taking care that none of the dead code was executed. A virus writer would also want to make some attempt to disguise the fact that the dead code is actually dead code, otherwise it could be ignored in the scanning process. In addition, unless a virus writer is careful, excessive dead code insertion might provide a simple heuristic for detecting the morphed viruses (e.g., an excessive number of `JMP` instructions used to avoid dead code). By ignoring these important practical issues, we are, in effect, considering the worst-case scenario from the virus detection point of view.

Results for block morphing in the "benign versus morphed viruses" case appears in Figure 13. Here, the morphing percentage ranges from 10% to 100%. It is important to note that for each score computation, we scored the morphed virus against the benign file from which the morphing code was extracted. This is certainly a worst-case scenario, since we expect these programs to be, in general, much more similar than if we score the morphed virus against a randomly selected benign file. As expected, the general trend is that the more code that is copied from a benign file into a morphed file, the more closely the score approaches that of the "benign versus benign" case. That is, as the morphing increases, the scores decrease, making detection more difficult. This is as expected.

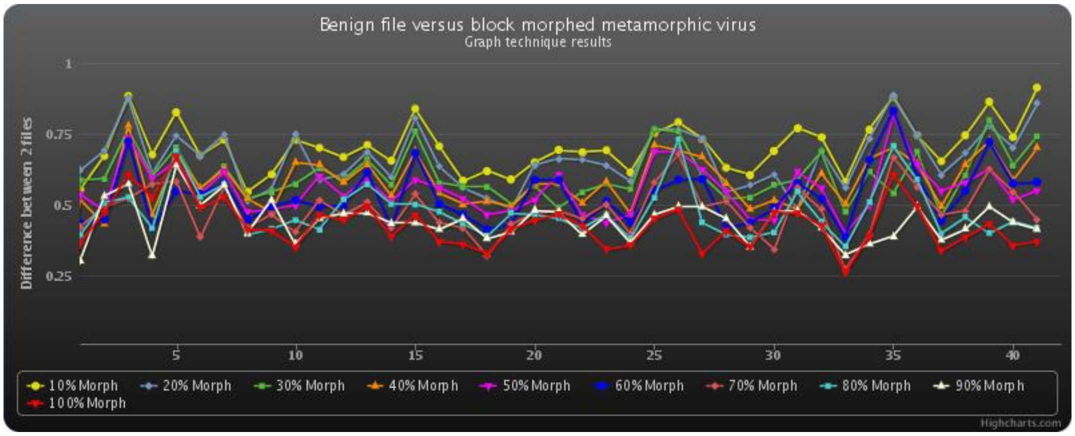

Figure 13: Block morphing: Benign versus morphed virus

In Figure 14 we give detection results when 30% block morphing is used. That is, we compare the "benign versus morphed virus" results with block morphing of 30% (as given in Figure 13) to the "morphed virus versus morphed virus" results, again with 30% block morphing. In this case, we clearly have some misclassifications, regardless of how we set the threshold. For example, it we set the threshold at 0.5, there are 4 false positives and 6 false negatives.

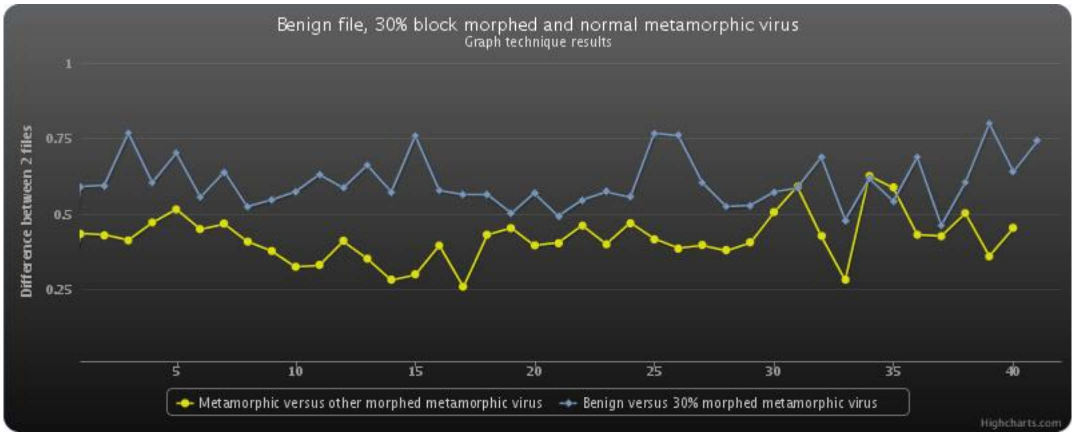

Figure 14: Morphed virus versus morphed virus and benign versus morphed virus: 30% block morphing

For comparison, we also implemented the HMM-based classification in [33] using the same 30% block morphed files used to obtain the results in Figure 14. These HMM results appear in Figure 15.

Using a threshold of -3.8, the HMM detector Figure 15 yields 1 false positive and 9 false negatives. Consequently, the results for the HMM detection are comparable to those for the graph-based similarity detection in Figure 14. However, it is worth noting that the HMM has a significant advantage in this case. Recall that for the graph similarity detection results in Figure 14, for each score computation, we compared the morphed virus with the benign file from which the morphing code was extracted. That is, we took the worst-case scenario in each and every score computation. For the HMM, no comparable worst-case scenario is possible. The HMM is trained on a set of morphed virus files, and then scored against a different set of morphed files and a set of benign files. That is, the HMM represents an average case, not a specific case.

Perhaps a fairer comparison would be to select a random benign file for the graph scoring technique, as opposed to selecting the specific benign file from which the morphing code was extracted. Figure 16 gives results for this particular case. These results demonstrate that, in practice, we can likely tolerate significantly higher levels of morphing than indicated by the results in Figure 14.

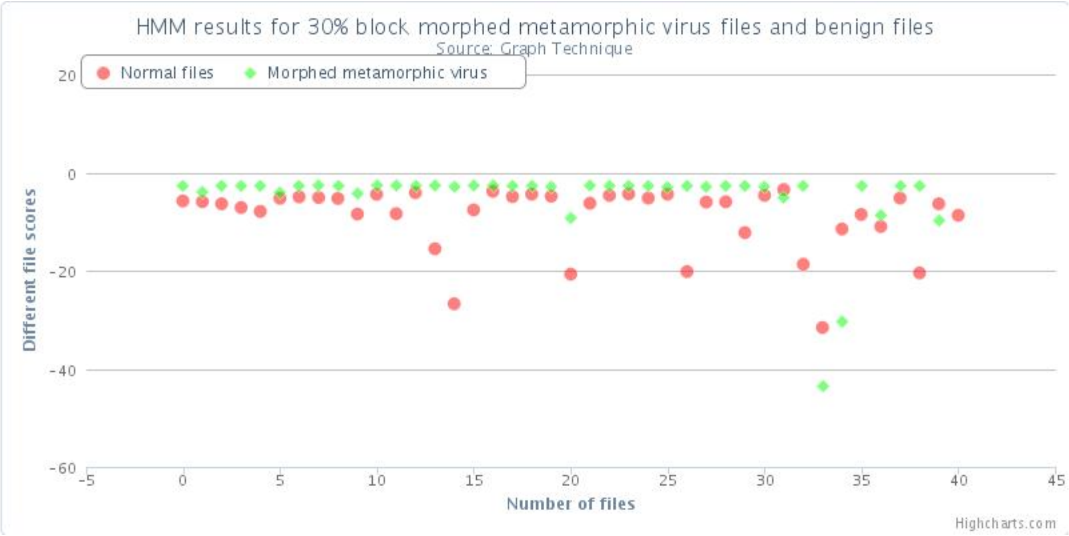

Figure 15: HMM 30% block morphed

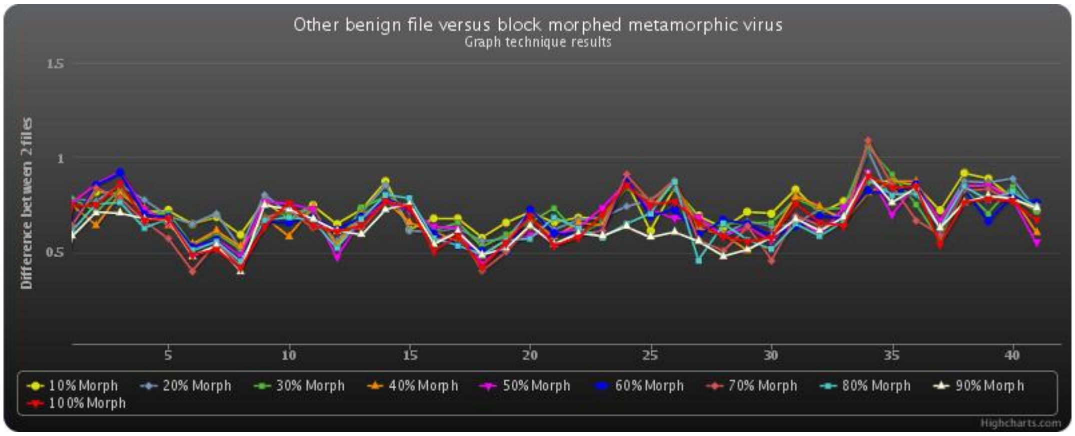

Figure 16: Metamorphic versus random benign

Next, we consider random morphing. As with block morphing, the morphing code is extracted from a randomly selected benign file. However, instead of inserting the code as a block, we disperse the morphing code approximately uniformly throughout the morphed virus. Again, we ignore the practical issues involved in making a functioning program out of the resulting morphed file—we simply modify the opcode sequence. As mentioned above, this is a worst-case scenario from the virus detection perspective. As an aside, we note that these practical issues are much more challenging in the case of random morphing than for block morphing.

Figure 17 shows the results for this case. Note that this figure is the random morphing analog of the block morphing case that appears in Figure 13.

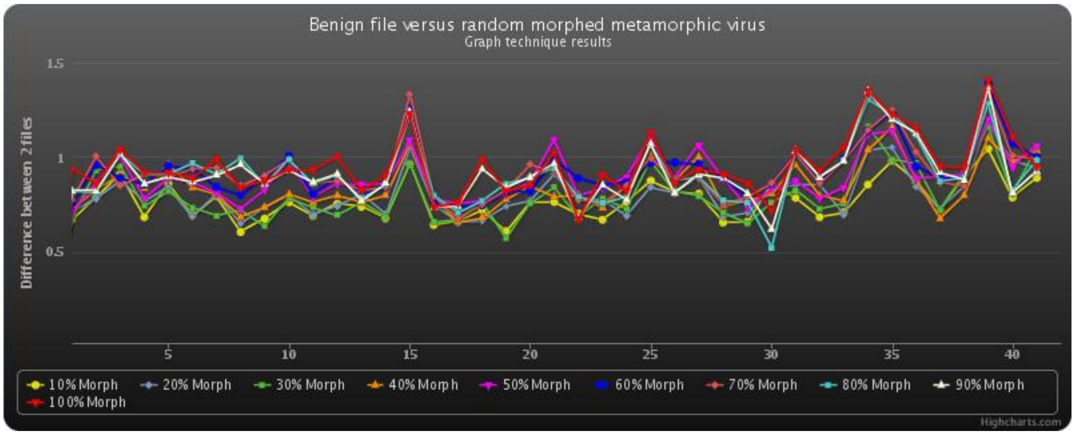

Figure 17: Random morphing: Benign versus morphed virus

The results in Figure 17 indicate that as the random morphing increases, the morphed viruses actually become more different from each other. Initially, this might seem counterintuitive. However, if we consider the effect of random morphing on the score calculation, then these result make sense. The score is based on consecutive pairs of opcodes. With random morphing, we are making essentially random changes to consecutive pairs of opcodes—increased morphing only increases this effect. Consequently, two randomly-morphed viruses are, in the sense of our score calculation, almost certainly "farther apart" than before the morphing, and they are also almost certainly farther from a given benign file. In effect, we are randomizing the weights in the opcode graph. In contrast, block morphing has the effect of merging the virus graph with (part of) a benign graph, thereby making the morphed virus more similar to the benign file, at least with respect to the opcode graph score in (1).

The bottom line is that random morphing would tend to make the detection problem easier, not more difficult. Finally, we note that a similar effect has been observed with HMM-based detection. That is, the more random the morphing (in the sense of being spread uniformly throughout the virus), the less effective the morphing is at defeating the HMM detector [17].

6 Conclusions

In this paper, we considered a similarity score based on opcode graphs extracted from executable files. We applied this score to the challenging problem of metamorphic malware detection. The score was shown to be effective—under some plausible scenarios it outperformed a previously developed technique based on hidden Markov models.

Some relevant open questions remain. The score used in the tests discussed here is given in (1). Alternative scoring functions are considered in [25], where it is noted that minor modifications to the scoring function tend to have surprisingly large effects on the results. It would be interesting to explore this more carefully, since it might be possible to find a stronger scoring function.

A more sophisticated classification scheme might offer a slight improvement in classification rates. For example, elementary techniques such as linear discriminant analysis (LDA) or quadratic discriminant analysis (QDA) could be used, or more advanced methods could be applied.

It might be useful to combine our opcode graph similarity technique with other techniques, such as the HMM-based detector in [33]. Such a combined approach could leverage the relative strengths of each of its components to yield a stronger overall detector.

Finally, it would be useful to have a standard metamorphic malware dataset so that proposed detection schemes could be compared, based on their performance on this standard data. This approach has worked well in other security-related research areas. For example, in the field of masquerade detection—a problem that arises in intrusion detection—there is a standard dataset that all proposed systems are tested against; see, for example [13]. Although this masquerade dataset, the so-called Schonlau dataset [26], is far from perfect, it provides a method for directly comparing the effectiveness of proposed systems.

References

[1] B. Anderson, et al, Graph-based malware detection using dynamic analysis, *Journal of Computer Virology*, Vol. 7, No. 4, November 2011, pp. 247–258

[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] E. Al daoud, et al, Detecting metamorphic viruses by using arbitrary length of control flow graphs and nodes alignment, *ICIT 2009 Conference – Bioinformatics and Image*, `http://www.ubicc.org/files/pdf/2_363.pdf`

[5] S. Cesare, Faster, more effective flowgraph-based malware classification, `http://www.ruxcon.org.au/2011-talks/faster-more-effective-flowgraph-based-malware-classification/`

[6] Cygwin, Cygwin utility files, `http://www.cygwin.com/`

[7] P. Desai and M. Stamp, A highly metamorphic virus generator, *International Journal of Multimedia Intelligence and Security*, Vol. 1, No. 4, 2010, pp. 402–427

[8] M. Eskandari and S. Hashemi, Metamorphic malware detection using control flow graph mining, *IJCSNS International Journal of Computer Science and Network Security*, Vol. 11, No. 12, December 2011, pp. 1–6, `http://paper.ijcsns.org/07_book/201112/20111201.pdf`

[9] T. Gartner, et al, On Graph Kernels: Hardness Results and Efficient Alternatives, Springer, pp. 129–143, 2003

[10] B. Halfpap, Artificial immune system virus detector, 2010, `http://resheth.wordpress.com/tag/virus-detection/`

[11] A. Hii, Chi-squared distance and metamorphic detection, Master's report, Department of Computer Science, San Jose State University, 2011

[12] A. Hlaoui and S. Wang, A New Algorithm for Inexact Graph Matching, `http://www.dmi.usherb.ca/~hlaoui/icpr2002.pdf`

[13] L. Huang and M. Stamp, Masquerade detection using profile hidden Markov models, *Computers and Security*, Vol. 30, Issue 8, November 2011, pp. 732–747

[14] A. Karnik, S. Goswami, and R. Guha, Detecting obfuscated viruses using cosine similarity analysis, *First Asia International Conference on Modelling & Simulation*, 2007, pp. 165–170

[15] E. Konstantinou, Metamorphic Virus: Analysis and Detection, 2008, `http://www.ma.rhul.ac.uk/static/techrep/2008/RHUL-MA-2008-02.pdf`

[16] J. Lee, K. Jeong, and H. Lee, Detecting metamorphic malwares using code graphs, *Proceedings of SAC10*, 2010

[17] D. Lin and M. Stamp, Hunting for undetectable metamorphic viruses, *Journal in Computer Virology*, Vol. 7, No. 3, August 2011, pp. 201–214

[18] C. Nachenberg, Understanding and managing Polymorphic viruses, *Symantec Enterprise Papers*, Vol. XXX, `http://www.symantec.com/avcenter/reference/striker.pdf`

[19] OECD, Malicious software (malware): A security threat to the Internet economy, `http://www.oecd.org/dataoecd/53/34/40724457.pdf`

[20] H. Ogata, et al, A heuristic graph comparison algorithm and its application to detect functionally related enzyme clusters, `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC110779`

[21] M. Patel, Similarity tests for metamorphic virus detection, Master's report, Department of Computer Science, San Jose State University, 2011, `http://www.cs.sjsu.edu/faculty/stamp/students/patel_mahim.pdf`

[22] S. Priyadarshi, Metamorphic detection via emulation Master's report, Department of Computer Science, San Jose State University, 2011, `http://www.cs.sjsu.edu/faculty/stamp/students/priyadarshi_sushant.pdf`

[23] L. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition, *Proceedings of the IEEE*, Vol. 77, No. 2, pp. 257–286, February 1989

[24] D. Radev, Lecture 13 – Eigenvectors, Eigenvalues, Stochastic Matrices, 2008
`http://www1.cs.columbia.edu/~coms6998/Notes/lecture13.pdf`

[25] N. Runwal, Graph technique for metamorphic virus detection, Master's report, Department of Computer Science, San Jose State University, 2011,
`http://www.cs.sjsu.edu/faculty/stamp/students/runwal_neha.pdf`

[26] M. Schonlau, et al, Computer intrusion: detecting masquerades, *Statistical Science*, Vol. 15, No. 1, 2001, pp. 1–17

[27] A. Shah, Approximate disassembly using dynamic programming Master's report, Department of Computer Science, San Jose State University, 2010,
`http://www.cs.sjsu.edu/faculty/stamp/students/shah_abhishek.pdf`

[28] SnakeByte, Next generation virus construction kit (NGVCK), 2002,
`http://vx.netlux.org/vx.php?id=tn02`

[29] M. Stamp, *Information Security: Principles and Practice*, second edition, Wiley 2011

[30] M. Stamp, A revealing introduction to hidden Markov models, 2011,
`http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf`

[31] P. Szor and P. Ferrie, Hunting for metamorphic, Symantec, 2001,
`http://www.symantec.com/avcenter/reference/`
`hunting.for.metamorphic.pdf`

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

[33] W. Wong and M. Stamp, Hunting for metamorphic engines, *Journal in Computer Virology*, Vol. 2, No. 3, December 2006, pp. 211–229
`http://www.cs.sjsu.edu/faculty/stamp/students/Report.pdf`