Profile Hidden Markov Models and Metamorphic Virus Detection

Srilatha Attaluri, Scott McGhee, and Mark Stamp
Department of Computer Science
San Jose State University
San Jose, California

Abstract

Metamorphic computer viruses "mutate" by changing their internal structure and, consequently, different instances of the same virus may not exhibit a common signature. With the advent of construction kits, it is easy to generate metamorphic strains of a given virus.

In contrast to standard hidden Markov models (HMMs), profile hidden Markov models (PHMMs) explicitly account for positional information. In principle, this positional information could yield stronger models for virus detection. However, there are many practical difficulties that arise when using PHMMs, as compared to standard HMMs.

Profile hidden Markov models are widely used in bioinformatics. For example, PHMMs are the most effective tool yet developed for finding family-related DNA sequences. In this paper, we consider the utility of PHMMs for detecting metamorphic virus variants generated from virus construction kits. PHMMs are generated for each construction kit under consideration and the resulting models are used to score virus and non-virus files. Our results are encouraging, but several problems must be resolved for the technique to be truly practical.

Keywords: metamorphic engine, malware, virus, profile hidden Markov model, bioinformatics, virus detection

1 Introduction

Computer viruses and other malware present an ongoing security threat. The most popular virus detection technique used today is signature detection, which is generally highly effective on known viruses. Of course, virus writers are aware of this "problem", and they often go to great lengths to hide virus signatures.

Metamorphism, which can be viewed as an advanced form of code obfuscation, is a potentially powerful means of evading signature detection. Today, it is easy for the aspiring virus writer to generate metamorphic variants by using construction kits [36].

Detecting metamorphic viruses is a challenge. For a properly designed metamorphic generator, it can be proved [4] that signature detection is not effective[1]. It has also been shown that the problem of metamorphic virus detection is, for a properly designed metamorphic generator, undecidable [13].

Hidden Markov models (HMMs) are widely used in speech recognition [30], as well as a variety of other applications. In [39], HMMs are employed to detect metamorphic viruses, with very promising results. The work presented in this paper can be viewed as a continuation of [39]. Here, our goal is to study the strengths and weaknesses of so-called profile hidden Markov models (PHMMs) with respect to metamorphic virus detection.

PHMMs are a substantially modified form of the standard HMM approach. PHMMs are widely used in bioinformatics to find distantly-related sequences of a given protein sequence family [8]. Analogous to [39], in this paper, we use PHMMs to model metamorphic virus families and we use the resulting models to score virus and non-virus files. Our PHMM models are created using opcode alignments generated from a collection of related metamorphic virus variants. Intuitively, there should be classes of viruses for which PHMMs are superior to standard HMMs, and it is also likely that there are classes of viruses where standard HMMs are superior.

This paper is organized as follows. Section 2 contains background information, including a brief discussion of metamorphic viruses, some details on the metamorphic virus construction kits studied in this paper, a quick look at some of the code obfuscation techniques that are often used in metamorphic viruses, and a very brief overview of popular anti-virus technologies. Section 3 provides an overview of standard hidden Markov models while Section 4 describes the algorithms and theory behind profile hidden Markov models. Section 5 contains a detailed discussion of our test data, implementation details related to the training of our PHMMs, and a discussion of the scoring of virus and non-virus files. Then is Section 6 we give our results concerning detection rates. In Section 7, we draw conclusions based on our findings and discuss future work.

[1]To oversimplify, the idea is to slice the program into pieces that are smaller than the signature window, then randomly rearrange the slices, inserting jump instructions so that the code is executed in its original order. By also including garbage code insertion and, say, "opaque predicates" [7], this approach yields a relatively simple metamorphic generator which is resistant to signature scanning and basic modifications thereof.

2 Background

2.1 Metamorphism and Metamorphic Viruses

Virus writers have long used encryption as a means to obscure virus signatures. However, encrypted viruses are relatively easy to detect, either by finding a signature for the decryption code or via emulation (i.e., let the virus decrypt itself, then look for a signature). To make detection more difficult, virus writers developed so-called polymorphic viruses, where the decryption code varies [34].

Metamorphic viruses take polymorphism to the limit by mutating the entire viral code, not just a decryptor. If the code is sufficiently mutated, no common signature will exist and emulation will not yield a viable signature. In this historical context, metamorphism can be viewed as an anti-emulation technique.

Consider the snippet of assembly code that appears in Table 1. Tables 2 and 3 provide morphed versions of the code in Table 1. The morphed code in Table 2 employs code reordering and equivalent code substitution, whereas the code in Table 3 employs these techniques as well as garbage code insertion. The hexadecimal representations of the corresponding executables are sufficiently different so that signature scanning is not feasible. This simple example illustrate the potential utility of metamorphism in evading signature detection.

	call Delta
Delta:	pop ebp
	sub ebp, offset Delta

Table 1: Original Code

	call Delta
Delta:	sub dword ptr[esp], offset Delta
	pop eax
	mov ebp, eax

Table 2: Morphed Version 1

A good metamorphic engine will likely employ multiple code-obfuscation methods. Obfuscation methods range from simple register renaming to sophisticated code-substitution techniques.

Some significant metamorphic viruses are listed in Table 4. While none of these viruses caused major damage, each was important in the evolution of metamorphic virus techniques.

	add ecx,0031751B	; junk
	call Delta	
Delta:	sub dword ptr[esp], offset Delta	
	sub ebx,00000909	; junk
	mov edx,[esp]	
	xchg ecx,eax	; junk
	add esp,00000004	
	and ecx,00005E44	; junk
	xchg edx,ebp	

Table 3: Morphed Version 2

virus name	innovation	date
Regswap	register swapping [16]	1998
Win32.Apparition	garbage insertion [16]	2000
W32.Evol	multiple techniques [26]	2000
Zmist	code integration [16]	2001
Win32.Metaphor	target-specific [16]	2002
Lexotan32	advanced techniques [27]	2002
Simile	entry point obfuscation [19]	2003
MSIL/Gastropod	parasitic insertion [11]	2004

Table 4: Notable Metamorphic Viruses

In addition to metamorphic viruses, mutation engines are available which can be used to change code structure—as opposed to creating malware per se. A wide variety of metamorphic engines are available, some of which employ relatively sophisticated techniques such as decryptor permutation, code compression, anti-heuristics, code permutation, and so on [36].

Among malware writers, interest in metamorphic viruses seems to have peaked in about 2002, and metamorphic viruses have never been a major problem "in the wild". This is most likely due to the fact that it is extremely difficult to write an effective metamorphic engine, as attested to by the infamous virus writer "Benny" [2][2]. As further evidence of the difficulty of creating truly metamorphic viruses, most self-proclaimed "metamorphic engines" studied in [39] failed to produce highly metamorphic variants. In addition, it is shown in [39] that metamorphic code that evades signature detection may still be detectable via statistical analysis (e.g., machine learning techniques). Effective metamorphic generators must, therefore, employ sufficient metamorphism to evade signature detection, and they must also produce variants

[2]Benny, formerly a member of the "29A" virus writing group, is thought to be the creator of the devastating Slammer worm [40].

that are statistically similar to "normal" code. Building such a generator is certainly challenging, but clearly not impossible, and we believe it is inevitable that potent metamorphic viruses will appear in the future. Consequently, it is critical that virus researchers continue to focus on metamorphism.

2.2 Virus Construction Kits

VXHeavens [36] is a website that provides several metamorphic virus construction kits, enabling a novice to easily develop advanced viruses. Construction kits combine features such as encryption and anti-debugging with metamorphic engines, allowing almost anyone to generate metamorphic viruses. Some of the kits are capable of generating a virtually unlimited numbers of metamorphic variants. Construction kits are available for viruses, trojans, logical bombs and worms. Since these kits can create variants with ease, they pose a challenge to anti-virus software.

Table 5 lists the virus construction kits considered in this paper. Additional information on each of these kits is given below.

name	version	year
Virus Creation Lab	VCL32	2004
Phalcon-Skism Mass Produced Code Generator	PS-MPC 0.91	1992
Next Generation Virus Creation Kit	NGVCK 0.30	2001

Table 5: Virus Construction Kits

VCL32 creates virus variants based on user-specified preferences. The first version of VCL was created by a group of virus writers called NUKE and appeared in 1992. A more recent version—developed by the "29A" virus writing group—surfaced in 2004. VCL32 provides a GUI interface for the user to choose from various preferences. Once the options are chosen, VCL32 generates assembly code for each virus variant, and these files can then be assembled to obtain exe files. It has been reported that the code generated by the earlier version had bugs and would not yield working code, but the current version of VCL32 seems to have overcome this problem. We employed the Borland Turbo Assembler and Tools (TASM), version 5.0, to assemble the VCL32 viruses. Many virus creators recommend TASM over the corresponding Microsoft assembler (MASM) for this purpose.

Phalcon and Skism were two independent groups that merged to form the Phalcon-Skism group [24]. Their metamorphic engine, PS-MPC, allows users to select from about 25 options, including parameters such as the payload type, memory resident (or not), encryption (or not), etc. The generated code depends on the month, day and time specified in the virus, as well as the minimum or maximum file sizes to infect. PS-MPC also implements obfuscation of the decryption code, but it does not implement anti-debugging or anti-emulation techniques.

NGVCK, created by “SnakeByte”, appeared in 2001 and, according to [39], it generates highly metamorphic variants. Unlike VCL32 and PS-MPC there is no need to set configuration settings as NGVCK randomly generates a new variant every time it is used. This construction kit utilizes junk code insertion, subroutine reordering, random register swapping and code-equivalent substitutions. NGVCK also implements anti-debugging and anti-emulation techniques. NGVCK was developed as a general purpose metamorphic engine and it has gone through multiple revisions. For this paper, we used NGVCK version 30, which, as of this writing, is the most recent stable version.

Construction kits and mutation engines are easy to use and they provide “personalization” of new viruses, which, among many other potential problems, makes it possible to resurrect old viruses by creating new variants that have new signatures. It is, therefore, important to consider techniques to automatically detect metamorphic variants.

2.3 Code Obfuscation and Metamorphism

The goal of code obfuscation is to produce code that is difficult to understand—essentially the opposite of good software engineering practice [32]. Code obfuscation is often employed by software developers to make reverse engineering attacks more difficult. Virus writers use similar obfuscation techniques when generating metamorphic variants of a given virus. In this section, we briefly discuss several common code obfuscation techniques.

Garbage or “do-nothing” code can be viewed as instructions that are a part of the program physically, but not logically. That is, they have no bearing on the outcome of the program. For example, a virus writer might employ register exchanging (XCHG) to slow down code emulation. Other instructions such as “NOP”, “MOV ax, ax”, “SUB ax, 0”, etc., can be used to make a virus binaries look different and thus possibly remain undetected. Garbage instructions may also be branches of code that are never executed or which perform some calculation using variables declared in other garbage blocks. So-called opaque predicates can be employed, which makes it very difficult to automatically determine the unexecuted branches [7].

One purpose of garbage code insertion is to confuse and exhaust a virtual machine or person analyzing the virus code. However, virus scanners are often powerful enough to get beyond do-nothing instructions. In fact, an excessive number of do-nothing instructions is itself a reasonable heuristic for flagging code as a possible virus. Another possible defense against garbage code is to employ optimizing compiler techniques to remove dead code [5]. Therefore, the utility of garbage code as a metamorphic technique may be somewhat limited.

Register renaming consists of modifying the names of variables or registers used in the code. When registers are changed the result is different opcodes that can evade elementary signature scanning. Regswap is a metamorphic virus that employs registers renaming for each variant.

Subroutine permutation is a simple obfuscation method where the subroutines are reordered. Such reordering will not affect the virus, since the order in which subroutines appear in the code is irrelevant to the program execution. Compared to most other obfuscation methods, subroutine permutation is relatively ineffective at evading signature detection, since the signature will generally still exists. Some simple metamorphic viruses, such as Win95.Ghost and Win95.Smash, employ subroutine permutation [16].

Code reordering alters the order of the instructions but maintains the original logical flow by inserting jump instructions. Reordering the code creates control flow obfuscation as the control changes depending on unconditional jumps. The use of unconditional jumps allows the code to be reordered in a virtually unlimited number of ways.

Any coding task can be implemented in many different ways. This simple fact makes it possible to vary the internal structure of viral code without affecting the function of the code. This type of obfuscation can also be used to shrink or expand the original code by substituting smaller or larger code segments. As a simple example "ADD ax, 3" can be transformed to "SUB ax, -3", since both the instructions add 3 to the content of ax register. The same effect can also be achieved with a two-step process such as "MOV bx, -3" and "SUB ax, bx". W32.Evol is an example of a metamorphic virus that makes extensive use of equivalent code substitution. Aggressive use of code substitution is a potent technique for evading signature detection. One detection technique that has been suggested as a defense against code substitution is to transform the code into a base form [14].

2.4 Antivirus Technologies

Table 6 lists the most popular virus detection techniques [28]. Of the techniques listed in Table 6, signature detection is undoubtedly the most widely used today.

technique	strength	weakness
signature detection	efficient	new malware
checksum	new malware	false positives
heuristic analysis	new malware	costly, unproven
virtual machine execution	encrypted viruses	costly

Table 6: Virus Detection Techniques

Here, we simply want to emphasize that since signature scanning is the most popular detection method today, virus writers focus most of their efforts on trying to evade signature detection.

As an aside, it has recently been suggested that we may have reached a tipping point, in the sense that malware now outnumbers "goodware". Consequently, it may be more effective to maintain signatures for goodware rather than malware [6].

3 Hidden Markov Models

3.1 Markov Chains

Consider a series—or chain—of states with probabilities associated to each transition between states. Such a chain is “Markov” if the transition probabilities depend only on the current state, not on the previous states, that is, a Markov chain has no “memory” [22]. More precisely, in a first-order Markov chain, the transition probabilities only depend on the current state, while in an nth order Markov chain, the transition probabilities depend on the current state and the $n-1$ previous states. In any case, a Markov chain has finite memory.

A Markov chain for a DNA sequence is shown in Figure 1 [8]. The DNA chemical code is represented by an alphabet of four symbols (i.e., bases) denoted A (adenosine), C (cytosine), G (guanine) and T (thymine). Each arrow in Figure 1 represents the transition probability of a specific base followed by another base. Transition probabilities could be calculated after observing several DNA sequences. The corresponding transition probability matrix provides a compact representation of these transition probabilities. This DNA Markov model is a first order Markov model since each event depends only on the previous event.

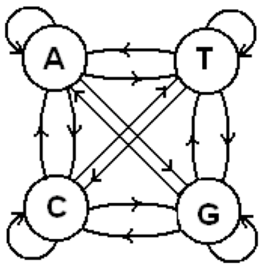


Figure 1: Markov Chain for DNA [8]

The transition probability from a state with observed symbol s to a state with observed symbol t, denoted a_{st}, is given by

$$a_{st} = P(x_i = t \,|\, x_{i-1} = s) \text{ for } 1 \leq s, t \leq N,$$

where N is the number of states and x_i represents the state at step i. Note that the sum of the transition probabilities from each state is equal to 1, since these transitions represent a probability distribution. Since there is a probability associated with each step, this model is sometimes called a probabilistic Markov model [17].

The probability of a sequence relative to a given model is calculated as [8]

$$\begin{aligned}P(x) =&P(x_L, x_{L-1}, \ldots, x_1)\\ =&P(x_L \,|\, x_{L-1}, \ldots, x_1)P(x_{L-1} \,|\, x_{L-2}, \ldots, x_1) \cdots P(x_1)\\ =&P(x_L \,|\, x_{L-1})P(x_{L-1} \,|\, x_{L-2}) \cdots P(x_2 \,|\, x_1)P(x_1)\\ =&P(x_1) \prod_{i=2}^{L} a_{x_{i-1}x_i}\end{aligned}$$

which follows by Bayes' Theorem. Note that $P(x_1)$ is the probability of starting at the state x_1. Generally, we include a "begin" state, and an "end" state to accommodate the first and last symbols of the output sequence.

3.1.1 Higher Order Markov Chains

As mentioned above, higher order Markov chains are those in which the current event depends on more than one previous event. An nth order Markov process over an alphabet of m symbols can be represented as a first order markov chain with an alphabet of mn symbols. For example, consider a two-symbol alphabet $\{A, B\}$. Then the sequence $ABAAB$ can be viewed as consisting of the consecutive pairs (AB, BA, AA, AB), which can be represented by a four-state first-order Markov model, with states AB, BB, BA and AA, or as a second-order Markov process.

3.2 Hidden Markov Models

Given a series of observations (i.e., an output sequence) from a Markov process, we might want to determine which state generated each observation. Consider the following urn and ball model [30]. There are N glass urns with a given distribution of colored balls in each, as illustrated in Figure 2. We know the distribution of balls in each urn and the rule used for determining which urn to select from. Since the underlying process is Markov, this rule can depend on the previous selection. Suppose we are given a sequence of colors corresponding to the balls that were selected, but we do not know from which urns the balls were selected. That is, the Markov process itself is "hidden". We would like to gain information about this hidden process via the observations—the colors of the balls selected.

So far, we have only outlined the basic structure of a hidden Markov model (HMM). Below, we discuss the problems that can be solved using the HMM approach. But first we present the standard HMM notation [31], and we consider a simple example.

- O is the observation sequence
- T is the length of the observation sequence
- N is the number of states in the (hidden) Markov process

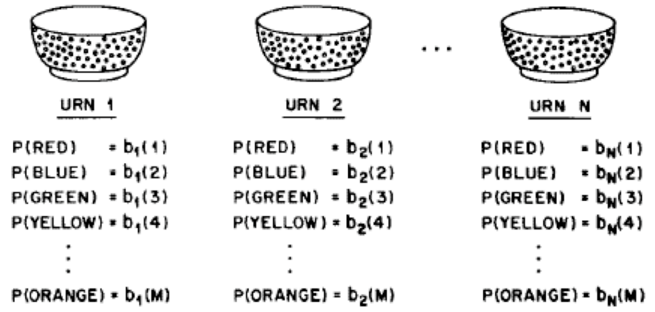


Figure 2: Urns and Ball Model [30]

- α is the alphabet for the model
- M is the number of symbols in the alphabet
- π is the initial state probability distribution
- A is the state transition probability matrix
- a_{ij} is the probability of a transition from state i to j
- B contains the N probability distributions for the observations (one distribution for each state of the Markov process)
- $b_i(k)$ is the probability of observing symbol k in state i
- $\lambda = (A, B, \pi)$ represents the HMM

Note that the HMM is completely specified by $\lambda = (A, B, \pi)$.

To illustrate an HMM, we consider an example where two coins—one biased and one normal (or fair)—are tossed T times to generate an observation sequence O. We toss one coin at a time, and we occasionally switch between the coins. Suppose that the alphabet is $\{H, T\}$ (which implies $M = 2$), where H stands for heads and T for tails, and we observe the sequence $O = \{H, T, H, T, H, H\}$. There are two hidden states (i.e., $N = 2$), corresponding to the biased and normal coins. Figure 3 illustrates the model.

Suppose that for the example in Figure 3, the transition probability matrix is

$$A = \begin{bmatrix} 0.95 & 0.05 \\ 0.20 & 0.80 \end{bmatrix}$$

where row (and column) 1 represents the normal coin, and row (and column) 2 represent the biased coin. Then, for example, the probability that the Markov process transitions from the normal state to the biased state is 0.05, since $a_{12} = 0.05$. That

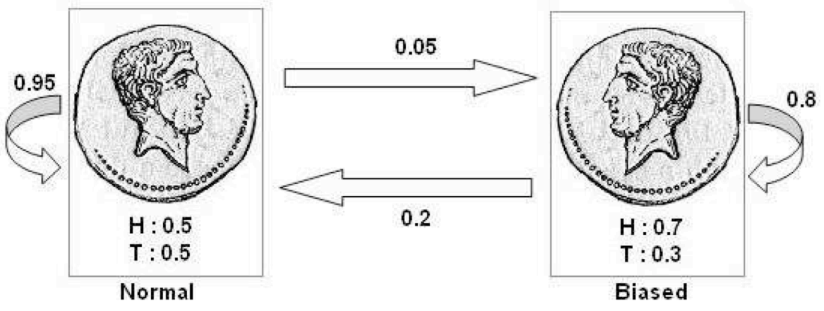


Figure 3: Example of HMM

is, if the normal coin is flipped, the probability the the biased coin is flipped next is 0.05.

The symbol distribution matrix B gives the probability distribution of H and T for both the normal and biased states. Suppose that in this example we have

$$B = \begin{bmatrix} 0.5 & 0.5 \\ 0.7 & 0.3 \end{bmatrix}$$

where first row gives the probability of H and T, respectively, when the normal coin is flipped, and second row is the corresponding distribution for the biased coin. The term $b_2(H)$ represents the probability of H when the biased coin is flipped—in this example, $b_2(H) = 0.7$. There is also an initial distribution, π, which specifies the probability that the Markov process begins with the normal and biased coins, respectively. In this example, we take

$$\pi = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}$$

Note that the matrices A, B and π are all row-stochastic, that is, each row is a probability distribution.

Again, we emphasize that the series of states in the underlying Markov process is "hidden". We observe the sequence of heads and tails that result from the process, and we assume that N and M are known. The HMM is denoted as $\lambda = (A, B, \pi)$, where the matrices A, B and π may or may not be known, depending on the particular problem that we are trying to solve.

The practical utility of HMMs derives largely from the fact that there exist efficient algorithms to solve each of the following problems [30].

- Problem 1: Given a model $\lambda = (A, B, \pi)$ and an observation sequence O, compute $P(O \mid \lambda)$. That is, we can compute the probability that a given model produced a given observation sequence.

- Problem 2: Given a model $\lambda = (A, B, \pi)$ and an observation sequence O, determine the most likely sequence of states $X = (x_1, \ldots, x_T)$ that could have produced the observed sequence. In other words, we can uncover the "hidden" part of the HMM.

- Problem 3: Given an observation sequence and parameters N and M, determine the model $\lambda = (A, B, \pi)$ that best fits the observed sequence. That is, we can "train" a model to fit the data. Remarkably, this training requires no a priori assumptions about the model, other than the parameters N and M, which specify the "size" of the model.

Of course, "most likely" and "best" have precise meanings in the HMM context; see [31] for more details.

4 Profile Hidden Markov Models

Profile HMMs (PHMMs) are a particular formulation of the standard HMM approach that are designed to deal with fundamental problems in bioinformatics. One crucial difference between HMMs and PHMMs is that the latter make explicit use of positional (or alignment) information contained in the observation sequences, whereas standard HMMs do not. Another difference is that unlike standard HMMs, PHMMs allow null transitions, which are necessary so that the model can match sequences that include insertions or deletions. In the case of DNA, such differences naturally occur during evolution [8]. Metamorphic viruses are often created in a process that is somewhat analogous to evolutionary change, so there is reason to believe that PHMMs may be effective in this context.

In DNA sequencing, we can align multiple sequences of genes that are known to have some significant biological relationship. The resulting multiple sequence alignment (MSA) can then be used to determine whether an unknown sequence might be related to the sequences that comprise the MSA. For our purposes, we would like to apply this to the case where the MSA consists of opcode sequences from a specific metamorphic generator, and then use the resulting PHMM to score strings of opcodes. Our goal is to determine whether a given opcode sequence might belong to a virus from the same family as the MSA.

The structure of a PHMM is illustrated in Figure 4. In Figure 4, the circles are delete states (which allow for null transitions), the diamonds are insert states (which allow gaps in a sequence alignment), and the rectangles are match states (which, essentially, correspond to the states in a standard HMM). Match and insert states are "emission states" since a symbol is emitted (i.e., an observation is made) whenever the PHMM passes through one of these states. Emission probabilities are calculated based on the frequency of the symbols that can be emitted at a particular state in the model. Note that the emission probabilities—which correspond to the B matrix in a standard HMM—are position-dependent, in contrast to a standard HMM. Furthermore, the emission probabilities are derived from the MSA and, therefore, creating the MSA is essentially equivalent to the training phase in a standard HMM.

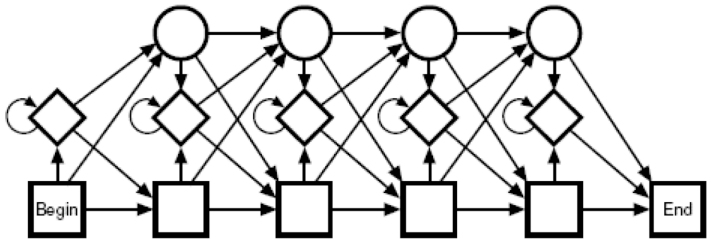


Figure 4: Structure of Profile HMM [18]

Finally, the delete states allow the model to pass through gaps, which invariably exist in an MSA to reach other emission states. Such gaps are necessary to prevent the model from over-fitting the training data.

The arrows in Figure 4 represent the possible transitions. Each transition has an associated probability and these transition probabilities determine the likelihood of the subsequent state, given the current state. Together, these transition probabilities correspond to the A matrix in a standard HMM.

The PHMM includes a begin state and an end state. The begin state incorporates the initial probability distribution into the PHMM.

The following notation is used in a PHMM.

- $X = (x_1, x_2, \ldots, x_i)$ is the sequence of emitted symbols (i.e., the observation sequence)
- N is the total number of states
- α is the alphabet for the model (the possible observation symbols)
- M represents the match states, $M_1, M_2, \ldots, M_N$
- I represents the insert states, $I_1, I_2, \ldots, I_N$
- D represents the delete states, $D_1, D_2, \ldots, D_N$
- π is the initial state probability distribution
- A is the state transition probability matrix
- A_{kl} is the transition frequency from state k to state l, as determined from the given MSA
- $a_{M_1 M_2}$ is the transition probability from match state M_1 to match state M_2 (transitions between different types of states are also allowed, as indicated in Figure 4).
- E is the emission probability matrix (for match and insert states)

- $E_{M_1}(k)$ is the emission frequency of symbol k at state M_1
- $e_{M_1}(k)$ is the emission probability of symbol k at state M_1 (emissions also occur at insert states)
- $\lambda = (A, E, \pi)$ represents the PHMM model

Below we give a brief example of a PHMM, but first we outline the process that we used to generate an MSA from a family of metamorphic computer viruses. Generating the MSA is the most challenging part of the entire process.

4.1 Multiple Sequence Alignment

To generate a multiple sequence alignment, we first create pairwise alignments, i.e., we align pairs of sequences. Then these pairwise alignments are combined to produce the desired MSA. An alignment can be created for any pair of sequences, but for our purposes we align pairs of sequences from the same virus family.

To visualize the alignment, the sequences can be considered rows in a matrix, where the positions are the columns. All symbols in one sequence will then be aligned with symbols in the other sequence so that related symbols or subsequences will align to the same column. In order to accomplish this, gaps can be inserted into either sequence. We represent a gap by a dash, "-".

The example in Table 7 shows an alignment of two sequences. These two sequences were derived from opcodes contained in a common subroutine. Note that we have substituted a single letter or number for each opcode—the precise conversion between opcodes and symbols is not needed for the discussion here; see [21] for the details.

Unaligned Sequences:
`AABNBAFCDBAAEAABCEDAEQCDABABBAF4NBBMBTYBAAAAABBCD`
`AABBAFCDBAAEA0ACEDAEQAABCDBALF4BBASBAAAAFBABCCD`
Alignment With Gaps:
`AABNBAFCDBAAEA-ABCEDAEQCD-ABABBA-F4NBBMBTY--BAAAA--ABB-CD`
`AAB-BAFCDBAAEA0A-CEDAEQ--AABCDBALF4-BB----ASBAAAAFBAB-CCD`

Table 7: Alignment of two NGVCK Virus Subroutines

The alignment in Table 7 contains several small matched subsequences consisting of 3 to 10 opcodes, which is fairly typical for the pairwise alignment of opcodes from the metamorphic generators we analyzed. We used a dynamic programming approach to generate this alignment. This is discussed in more detail below.

In bioinformatics applications, the purpose of aligning sequences is to look for evidence that the sequences diverged from a common ancestor by a process of mutation and selection [8]. In the case of proteins and DNA sequences, the basic mutational processes which are normally considered are the following:

- Substitution — a subsequence has been substituted for a subsequence in the original
- Insertion — a subsequence was inserted into the original sequence
- Deletion — a subsequence was removed from the original

In the case of metamorphic viruses, these same processes can also occur. However, there is another basic process which would not normally be considered in biological sequences, namely,

- Permutation — a re-ordering of the original sequence

A permutation could be stated in terms of a series of insertions and deletions, but it is important to make a distinction between the mutational processes of substitution, insertion and deletion, as opposed to the arbitrary permutations that are possible with metamorphic viruses.

Since certain metamorphic generators may make heavy use of permutations, we must consider the impact that permutations can have on pairwise alignments. It is easy to see that a permutation can have a large effect. For example Table 8 gives a pairwise alignment of two sequences where a simple permutation has been applied—the second half of the sequence has been placed at the beginning. In this example, the resulting alignment has a large number of gaps which effectively reduces the number of meaningful positions in the alignment. By extension, it is easy to see that many other permutation (e.g., reversing the sequence) would degrade the quality of a pairwise alignment even further.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ-------------
-------------NOPQRSTUVWXYZABCDEFGHIJKLM
```

Table 8: Effect of Permutation on Pairwise Alignment

To deal with the problems caused by permutations, it may be possible to preprocess the sequences being aligned in such a way that some of the effect of the permutation is negated. For example, it may be possible to put subroutines into a common order. However, any such preprocessing step will increase the complexity of creating alignments, as well as increasing the complexity of scoring. We will consider preprocessing again below.

To align sequences, we must have a means for scoring a putative alignment. Aligned opcodes is the ideal case; therefore, the real question is how much to penalize mismatches. Not all mismatches are equally bad since some opcodes can be considered closely related, while other opcodes are not. To deal with this scoring issue, it is standard practice to employ a substitution scoring matrix. This matrix contains all of the possible scores when any symbol is aligned with any other symbol. Consequently, if a given sequence has an alphabet with, say, 100 symbols, the scoring matrix will be 100×100 in size. Note that values on the diagonal of the substitution matrix correspond to no substitution at all, which is clearly a desirable case. In general, the value on the diagonal will be the largest number in a column (or row). In addition, our scoring matrices will be symmetric, since we assume that the substitution of opcode "A" for "B" carries the same penalty as substituting "B" for "A".

We need to determine a substitution matrix to apply to virus opcodes. After considerable experimentation, we settled on the following simple criteria to fill our substitution matrix.

- Aligning two symbols that are the same is a high positive score
- Aligning two "rare" symbols with each other is a medium positive score
- Aligning two different symbols is a low negative score
- Aligning two "markers" (i.e., subroutine boundaries) is a low positive score
- Aligning a marker with a non-marker is a high negative score (i.e., not aligning subroutines is penalized)

A sophisticated criteria that more thoroughly takes individual opcode relationships into account would likely improve the resulting models somewhat. In particular, we could negate much of the effect of equivalent code substitution by not penalizing such substitutions in our substitution matrix. Ideally, the substitution matrix would be tailored to a specific metamorphic generator. However, we found that the simple criteria above yielded strong PHMMs for all but one of the metamorphic generators under consideration and for the exceptional generator (NGVCK), we found that modifications to the scoring matrix appear to be of little value. We have more to say about this situation in Section 6.

To obtain better alignment of subsequences, gaps must be allowed. However, gaps tend to make the resulting model more generic, so we want to penalize the creation of gaps to some degree, and the penalty will be length-dependent, i.e., the longer the gap, the higher the penalty. Let g be the length and $f(g)$ the corresponding gap penalty. There are two basic types of gap penalty models commonly used in sequence analysis:

- Linear gap penalty — The gap penalty is the product of the size of the gap and the gap cost: $f(g) = dg$, where d is the gap-cost.
- Affine gap penalty — Opening a gap has an initial cost to start the gap, and a fixed cost for each subsequent gap: $f(g) = a + e(g - 1)$, where a is the gap opening cost, and e is the gap extension cost.

Note that the linear gap penalty is a special case of the affine gap penalty, where we choose the gap opening cost to equal the gap extension cost. For this research, we employed an affine gap penalty with the values a and e determined by trial and error.

Once the substitution matrix and gap penalty have been defined, we employed a fairly standard dynamic program to create pairwise alignments. A dynamic program will find the highest scoring path—as opposed to an HMM, which maximizes the expectation at each position in the path[3]. Dynamic programming is highly efficient, and generating the pairwise alignments is one-time work per PHMM.

The following definitions are used to specify our pairwise alignment dynamic program:

$$\begin{aligned}
x &= \text{first sequence to align} \\
y &= \text{second sequence to align} \\
|a| &= \text{length of sequence } a \\
a_i &= \text{the } i\text{th symbol of sequence } a \\
a_{i\ldots j} &= \text{subsequence } a_i, \ldots, a_j \text{ of } a \\
s(p,q) &= \text{score assigned to substituting symbol } p \text{ for } q \\
g(n) &= \text{cost of adding a gap to a sequence with } n-1 \text{ gaps} \\
F, G &= \text{matrices of size } |x|+1 \times |y|+1 \text{ (indices are 0 based)} \\
F(i,j) &= \text{optimal score for aligning } x_{1\ldots i} \text{ with } y_{1\ldots j} \\
G(i,j) &= \text{number of subsequent gaps used to generate } F(i,j).
\end{aligned}$$

The dynamic program recursion is initialized by

$$\begin{aligned}
&G(i,0) = F(i,0) = 0 \\
&G(0,j) = j \\
&F(0,j) = \sum_{n=1}^{j} g(n).
\end{aligned}$$

Note that $F(0,j)$ is simply the cost (i.e., penalty) associated with aligning j gaps. Finally, the recursion is given by

$$F(i,j) = \max \begin{cases} F(i-1,j-1) + s(x_i, y_j) & \text{case 1} \\ F(i-1,j) + g(G(i-1,j)) & \text{case 2} \\ F(i,j-1) + g(G(i,j-1)) & \text{case 3} \end{cases}$$

where

$$\begin{aligned}
&\text{if case 1 holds, then } G(i,j) = 0 \\
&\text{if case 2 holds, then } G(i,j) = G(i-1,j) + 1 \\
&\text{if case 3 holds, then } G(i,j) = G(i,j-1) + 1.
\end{aligned}$$

[3]Note that HMMs can, in fact, be used to generate pairwise alignments [8].

The point here is that the dynamic program will find the optimal path, given the assigned scores (as specified by s) and the gap penalties (as specified by g).

Given a collection of pairwise alignments, we would like to construct a multiple sequence alignment (MSA). The resulting MSA will contain a great deal of statistical information based on the various frequencies of symbols at each position. In effect, the MSA provides us with a probability distribution for each column of the data, as well as various transition probabilities. This statistical information is then directly used to create the PHMM that is, in turn, used to score opcode sequences.

There are many possible approaches to creating an MSA. If the number and length of sequences being aligned is small, it is not too difficult to create a plausible alignment by hand, but this is not practical for the opcode sequences that we consider here. One of the simplest means to automatically create an MSA is to use a so-called progressive alignment algorithm. This type of algorithm begins with an initial pairwise alignment and then builds on it by incorporating other pairwise alignments one by one until all pairwise alignments are included. Unfortunately, gaps tend to proliferate using such an approach, since gaps that appear in any of the newly-included pairwise alignment tend to also appear in the resulting MSA.

Another more sophisticated approach, is the Feng-Doolittle progressive alignment algorithm [10], in which we pre-calculate all possible alignment scores between pairs of n sequences, and then select $n - 1$ alignments which "connect" all sequences and maximize the pairwise alignment scores. Once the scores are calculated, one way to represent this data is as an undirected fully-connected graph in which the vertices represent the sequences and the edges are assigned distance values equal to the alignment scores between the sequences. When the data is represented in this way, the objective is to choose the alignments (i.e. the edges in the graph) that maximize the score. This problem can be reduced to the standard graph theory problem of producing a minimum spanning tree for the given graph. The only difference from a standard spanning tree scenario is that we are trying to maximize the score, as opposed to minimizing the cost, but this is easily remedied by multiplying all of our scores by -1.

In the Feng-Doolittle algorithm, the spanning tree is referred to as a "guide tree", and it is calculated using a clustering algorithm due to Fitch and Margoliash [8]. For simplicity, we have chosen to use Prim's algorithm [29] to construct the spanning tree. This simplification introduces a potential stability issue in the construction of the spanning tree, but our experimental results indicate that the use of Prim's algorithm does not appear to create any significant problems in practice; see [21] for more details.

After calculating the minimum spanning tree, the MSA is constructed by selecting pairwise alignments in the order that they arise when traversing the tree, starting from the alignment with the highest score. Next, we provide an example of this MSA construction process.

To demonstrate our MSA construction algorithm, we begin with 10 opcode sequences taken from NGVCK virus variants. The sequences have been trimmed to a single subroutine from each of the variants to simplify the example.

In Table 9, a distance matrix is given with all possible alignment scores among the 10 sequences, and a representation of the corresponding spanning tree appears in Figure 5. As discussed above, the spanning tree was constructed using Prim's algorithm.

	1	2	3	4	5	6	7	8	9	10
1	—	85	63	74	70	84	61	57	62	70
2	85	—	79	73	66	59	94	61	59	51
3	63	79	—	75	68	60	55	85	52	65
4	74	73	75	—	105	54	60	78	59	53
5	70	66	68	105	—	40	61	79	58	39
6	84	59	60	54	40	—	68	45	75	78
7	61	94	55	60	61	68	—	64	72	42
8	57	61	85	78	79	45	64	—	50	70
9	62	59	52	59	58	75	72	50	—	81
10	70	51	65	53	39	78	42	70	81	—

Table 9: Alignment Scores

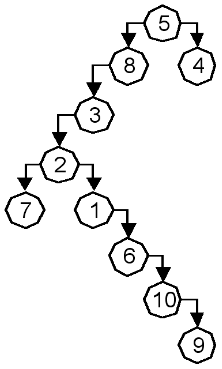


Figure 5: Spanning Tree for Table 9

An MSA based on Table 9 and Figure 5 is illustrated in Table 10. In Table 10, pairwise alignments are denoted as ordered pairs where the first number represents the index of a sequence that has already been incorporated into the MSA, and the second index represents a sequence which is new to this alignment (except, of course, for the first selected pair, in which case both sequences are not yet in the MSA). Once the spanning tree is calculated, the MSA is initialized with the highest scoring alignment; for the example in Table 10, initial alignment $(5, 4)$ was chosen. After the initial alignment, the following eight alignments (eight iterations needed to align ten sequences) are added to the MSA in order: $(5,8)$, $(8,3)$, $(3,2)$, $(2,7)$, $(2,1)$, $(1,6)$, $(6,10)$, $(10,9)$. Table 10 provides a snapshot of the third iteration. Note that in this example "+" is used to represent a "neutral" character that is inserted to better align the sequences; these neutral characters will represent gaps in the final alignment.

MSA Before New Alignment	
5)	CDABBAFCDB1AAEAA+CEDA+EQ+CDABABABALF4LBBAFBSBAAAAA
4)	2AABBAFCDABA+EAABCEDCDEQFCDABA+APALF4+BBA++SBAAAAA
8)	++AABA+CDB+AAEAA+CEDCDEQ+CDABPBA+ABF4+BBAFBSBMAAAA
3)	A+ABBAFCDABA+EAA+CEDCDEQA++ABFBAN++F4+BBAFBTYBAAAA
New Alignment	
2)	A-ABNBAFCD-BAAEAABCEDA-EQ-CDABAB--BAF4NBBM-BTYBAAAA
3)	A+AB-BAFCDABA+EAA+CEDCDEQA++ABFBAN++F4+BBAFBTYBAAAA
MSA After New Alignment	
5)	CDAB+BAFCDB1AAEAA+CEDA+EQ+CDABABABALF4LBBAFBSBAAAAA
4)	2AAB+BAFCDABA+EAABCEDCDEQFCDABA+APALF4+BBA++SBAAAAA
8)	++AA+BA+CDB+AAEAA+CEDCDEQ+CDABPBA+ABF4+BBAFBSBMAAAA
3)	A+AB+BAFCDABA+EAA+CEDCDEQA++ABFBAN++F4+BBAFBTYBAAAA
2)	A+ABNBAFCD+BAAEAABCEDA+EQ+CDABAB++BAF4NBBM+BTYBAAAA
Final alignment	
1)	A-AB-BAFCD-B-AAEA0ACEDA-EQ---A-ABCDBALF4-BBASB---AAAAFB
2)	A-ABNBAFCD-B-AAEAABCEDA-EQ-CDABAB--BA-F4NBBM-BTYBAAAA--
3)	A-AB-BAFCDAB-A-EAA-CEDCDEQA--ABFBAN---F4-BBAFBTYBAAAA--
4)	2AAB-BAFCDAB-A-EAABCEDCDEQFCDABA-APAL-F4-BBA--SBAAAAA--
5)	CDAB-BAFCDB1-AAEAA-CEDA-EQ-CDABABABAL-F4LBBAFBSBAAAAA--
6)	CDABAAA----B-A-EA-ACEDCDEQ---A-ABCD-A-F4-BBASB---AAAAFB
7)	CDAB--A-CDAB-A-EAA-CEDA-EQ-CDABCDCDAA-F4MBB--ATYBAAAA--
8)	--AA-BA-CDB--AAEAA-CEDCDEQ-CDABPBA-AB-F4-BBAFBSBMAAAA--
9)	CDAB--RBAFABPAAEA-ACEDCDEQAABCDAFAL---F4NBBASB---AAAAMB
10)	A-ABAA-----B-AAEA-ACEDCDEQAABAFA------F4BNBASB---AAAAFB

Table 10: Snapshots of MSA Construction

In summary, we employed the following steps to generate an MSA based on metamorphic virus opcode sequences:

1. Create pairwise alignments
   (a) Generate a substitution scoring matrix
   (b) Use a dynamic program to generate pairwise alignments
2. Use pairwise alignments to construct the MSA

(a) From pairwise alignments, generate a spanning tree using Prim's algorithm

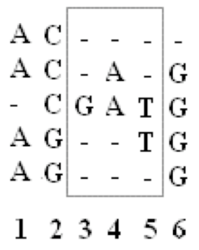
(b) Add sequences to the MSA in the order determined by the spanning tree, working from the highest scoring sequence to the lowest—inserting gaps as necessary—as illustrated in Table 10.

Note that gap penalties are explicitly included in the dynamic program. This is crucial, since the number of gaps tends to grow during the construction of the MSA. An excessive number of gaps in the MSA makes scoring ineffective, since the more gaps that are present, the easier it is for a random sequence to "match" the MSA, which results in a high match score from the corresponding PHMM.

4.2 PHMM Example

In this section we construct a PHMM from a given MSA. For the sake of brevity, we have chosen a much simpler MSA than the one presented in the previous section.

Consider the multiple sequence alignment (MSA) in Figure 6. Note that these sequences are merely illustrative and are not intended to represent biological sequences.



A	C	-	-	-	-
A	C	-	A	-	G
-	C	G	A	T	G
A	G	-	-	T	G
A	G	-	-	-	G
1	2	3	4	5	6

Figure 6: Multiple Sequence Alignment Example

The first step in creating a PHMM is to determine which columns in the MSA form the match and insert states. The more "conservative" columns are used as match states (i.e., the columns where more than half of the characters are symbols, as opposed to gaps), while the positions with more gaps than characters are insert states [8]. In the MSA in Figure 6, columns 1, 2, and 6 correspond to match states M_1, M_2, and M_3, respectively, while columns 3, 4, and 5 together correspond to the insert state I_2.

Next, we calculate the emission probabilities for column 1 of Figure 6. By simply counting the frequency of each symbol, we find

$$e_{M_1}(A) = 4/4,\ e_{M_1}(C) = 0/4,\ e_{M_1}(G) = 0/4,\ e_{M_1}(T) = 0/4. \tag{1}$$

All but one of these probabilities is zero. However, zero probabilities are highly undesirable in a PHMM, since they will eliminate "nearby" sequences from consideration, effectively over fitting the training data. To avoid this problem, one standard approach is to use the "add-one rule" [8], where we add 1 to each numerator and we add the total number of symbols to each denominator. Since there are four distinct symbols in our example, using the add-one rule, the probabilities in equation (1) become

$$e_{M_1}(A) = (4+1)/(4+4) = 5/8,\ e_{M_1}(C) = 1/8,\ e_{M_1}(G) = 1/8,\ e_{M_1}(T) = 1/8.$$

Without the add-one rule, the emission probabilities for the insert state I_2 would be

$$e_{I_2}(A) = 2/5,\ e_{I_2}(C) = 0/5,\ e_{I_2}(G) = 1/5,\ e_{I_2}(T) = 2/5$$

since these are the ratios of the 5 emitted symbols that appear in the "box" in Figure 6. Using the add-one rule, these probabilities become

$$e_{I_2}(A) = 3/9,\ e_{I_2}(C) = 1/9,\ e_{I_2}(G) = 2/9,\ e_{I_2}(T) = 3/9.$$

From Figure 6, using the add-one rule, we obtain the emission probabilities in Table 11.

$e_{M_1}(A) = 5/8$	$e_{I_1}(A) = 1/4$
$e_{M_1}(C) = 1/8$	$e_{I_1}(C) = 1/4$
$e_{M_1}(G) = 1/8$	$e_{I_1}(G) = 1/4$
$e_{M_1}(T) = 1/8$	$e_{I_1}(T) = 1/4$
$e_{M_2}(A) = 1/9$	$e_{I_2}(A) = 3/9$
$e_{M_2}(C) = 4/9$	$e_{I_2}(C) = 1/9$
$e_{M_2}(G) = 3/9$	$e_{I_2}(G) = 2/9$
$e_{M_2}(T) = 1/9$	$e_{I_2}(T) = 3/9$
$e_{M_3}(A) = 1/8$	$e_{I_3}(A) = 1/4$
$e_{M_3}(C) = 1/8$	$e_{I_3}(C) = 1/4$
$e_{M_3}(G) = 5/8$	$e_{I_3}(G) = 1/4$
$e_{M_3}(T) = 1/8$	$e_{I_3}(T) = 1/4$

Table 11: Emission Probabilities for the MSA in Figure 6

Note that the emission probability matrix E of the PHMM corresponds to the matrix B in a standard HMM. However, E differs in the fact that the probabilities are position-dependent and it also differs since in the PHMM case, we have more than one way that a symbol can be emitted at each position (i.e., match or insert).

Next, we consider the transition probabilities for our PHMM. Intuitively, we want [8]

$$a_{mn} = \frac{\text{Number of transitions from state } m \text{ to state } n}{\text{Total number of transitions from state } m \text{ to any state}}.$$

Let B represent the begin state. Then, from Figure 6 we would have

$$a_{BM_1} = 4/5$$

since 4 of the 5 transitions from B to column 1 are matches. Furthermore, we would have

$$a_{BD_1} = 1/5 \text{ and } a_{BI_0} = 0/5$$

since one element in column 1 represents a delete state (D_1), and there are no insert states (I_0).

As with the emission probability calculations, we want to avoid over fitting the data, so we use the analog of the add-one rule. However, instead of adding one for each symbol, we add one for each of the possible transitions, match, insert, and delete. For example, using the add-one rule, we have

$$a_{BM_1} = (4+1)/(5+3) = 5/8, \quad a_{BD_1} = 2/8, \quad \text{and} \quad a_{BI_0} = 1/8.$$

In cases where there is no data we set the probabilities equal to 1/3. For example, we have no transitions from insert state 1, and consequently we set

$$a_{I_1M_2} = a_{I_1I_1} = a_{I_1D_2} = 1/3.$$

As a final example of transition probabilities, consider the delete state D_1, which corresponds to the "dash" in column 1 of Figure 6. From Figure 6, we see that the only possible transition is to a match state in column 2, which, without the add-one rule, would imply

$$a_{D_1M_2} = 1/1 = 1, \quad a_{D_1I_1} = 0/1 = 0, \quad \text{and} \quad a_{D_1D_2} = 0/1 = 0.$$

In this case, utilizing the add-one rule yields

$$a_{D_1M_2} = (1+1)/(1+3) = 2/4, \quad a_{D_1I_1} = 1/4, \quad \text{and} \quad a_{D_1D_2} = 1/4.$$

The transition probabilities for the example in Figure 6 appear in Table 12, where we have used the add-one rule.

Finally, it is worth noting that there is nothing sacrosanct about the add-one rule. In fact more advanced techniques are often used in bioinformatics. Any technique that makes use of the MSA data, eliminates zero probabilities, and yields a row-stochastic matrix could be used to create the A matrix. Here, we have adopted the add-one rule because it is the simplest approach

The PHMM corresponding to the MSA in Figure 6, with beginning and ending states included, appears in Figure 7, where the probabilities of the edges are given in Table 12. Note that the desired PHMM model is fully specified by E (the emission probability matrix) and A (the transition probability matrix).

4.3 Forward Algorithm

The forward algorithm enables us to efficiently computer $P(X \mid \lambda)$, that is, we can score a given observation sequence to determine how well it matches a given PHMM.

$a_{BM_1} = 5/8$	$a_{I_0M_1} = 1/3$	
$a_{BI_0} = 1/8$	$a_{I_0I_0} = 1/3$	
$a_{BD_1} = 2/8$	$a_{I_0D_1} = 1/3$	
$a_{M_1M_2} = 5/7$	$a_{I_1M_2} = 1/3$	$a_{D_1M_2} = 2/4$
$a_{M_1I_1} = 1/7$	$a_{I_1I_1} = 1/3$	$a_{D_1I_1} = 1/4$
$a_{M_1D_2} = 1/7$	$a_{I_1D_2} = 1/3$	$a_{D_1D_2} = 1/4$
$a_{M_2M_3} = 2/8$	$a_{I_2M_3} = 4/8$	$a_{D_2M_3} = 1/3$
$a_{M_2I_2} = 4/8$	$a_{I_2I_2} = 3/8$	$a_{D_2I_2} = 1/3$
$a_{M_2D_3} = 2/8$	$a_{I_2D_3} = 1/8$	$a_{D_2D_3} = 1/3$
$a_{M_3E} = 5/6$	$a_{I_3E} = 1/2$	$a_{D_3E} = 2/3$
$a_{M_3I_3} = 1/6$	$a_{I_3I_3} = 1/2$	$a_{D_3I_3} = 1/3$

Table 12: Transition Probabilities for the MSA in Figure 6

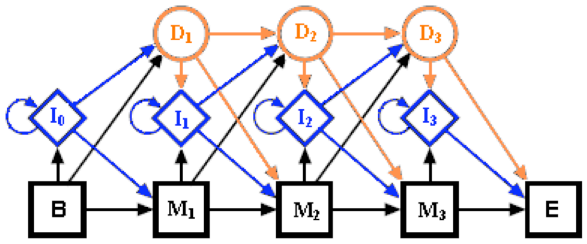


Figure 7: PHMM with Three Match States

Note that this corresponds to HMM “problem 1”, as discussed at the end of Section 3.2. There also exist PHMM algorithms that can be used to solve the other two problems mentioned at the end of Section 3.2: The Viterbi algorithm is used to solve “problem 2”, while Baum-Welch re-estimation—which is itself a special case of the expectation maximization (EM) algorithm—is used to solve “problem 3”. For the work presented here, we only require the forward algorithm; see [8] or [1] for information on the other PHMM algorithms.

Before presenting the forward algorithm, we show how $P(X \mid \lambda)$ can be calculated in an intuitively simple, but computationally inefficient manner. The brute-force approach to calculate $P(X \mid \lambda)$ is to take the sum of the probabilities of all possible paths that emit the sequence X. For example, a sequence $X = (A, B)$ emitted by a 4-state PHMM model has 13 possible paths—all 13 paths are listed in Table 13. Recall that a symbol is emitted each time the model passes through an insert or a match state. Figure 8 provides a graphical illustration of the paths listed in Table 13, where the begin and end states have been omitted.

Calculating probabilities for each possible case is clearly not efficient. The forward algorithm computes the desired probability recursively, by reusing scores calculated

	I_0	I_1	I_2	M_1	M_2
1	A,B	—	—	—	—
2	A	B	—	—	—
3	A	—	B	—	—
4	A	—	—	B	—
5	A	—	—	—	B
6	—	A,B	—	—	—
7	—	A	B	—	—
8	—	A	—	—	B
9	—	—	A,B	—	—
10	—	B	—	A	—
11	—	—	B	A	—
12	—	—	—	A	B
13	—	—	B	—	A

Table 13: Possible Paths for 4-state PHMM

for partial sequences. For a PHMM the forward algorithm recursive relation is [8]

$$\begin{aligned}
F_j^M(i) =& \log \frac{e_{M_j}(x_i)}{q_{x_i}} + \log \big(a_{M_{j-1}M_j} \exp(F_{j-1}^M(i-1)) + a_{I_{j-1}M_j} \exp(F_{j-1}^I(i-1)) \\
&+ a_{D_{j-1}M_j} \exp(F_{j-1}^D(i-1))\big) \\
F_j^I(i) =& \log \frac{e_{I_j}(x_i)}{q_{x_i}} + \log \big(a_{M_j I_j} \exp(F_j^M(i-1)) + a_{I_j I_j} \exp(F_j^I(i-1)) \\
&+ a_{D_j I_j} \exp(F_j^D(i-1))\big) \\
F_j^D(i) =& \log \big(a_{M_{j-1}D_j} \exp(F_{j-1}^M(i)) + a_{I_{j-1}D_j} \exp(F_{j-1}^I(i)) + a_{D_{j-1}D_j} \exp(F_{j-1}^D(i))\big)
\end{aligned}$$

where the indices i and j specify columns in the MSA (as discussed above), x_i is the ith observation symbol, and the base case for the recursion is $F_0^M(0) = 0$. Here, q_{x_i} is the background distribution, i.e., the distribution of the symbol x_i in the random model. Then $F_j^M(i)$ represents the score for the subsequence $x_1, \ldots, x_i$ up to state j (note that unlike the standard HMM, the indices i and j need not coincide, due to insertions and/or deletions). Finally, in this recursion, some insert and delete terms are not defined, such as $F_0^I(0)$, $F_0^D(0)$, and so on. These undefined terms are simply ignored when calculating the scores.

Note that the value of $F_j^M(i)$ depends on $F_{j-1}^M(i-1)$, $F_{j-1}^I(i-1)$ and $F_{j-1}^D(i-1)$, along with their respective transition probabilities. Similar statements hold for $F_j^I(i)$ and $F_j^D(i)$. The emission probabilities are also used when calculating $F_j^M(i)$ and $F_j^I(i)$, but not for $F_j^D(i)$, since delete states do not emit symbols. The states M_0 and M_{N+1} represent the "begin" and "end" states, respectively, and, as with delete states, they do not emit symbols.

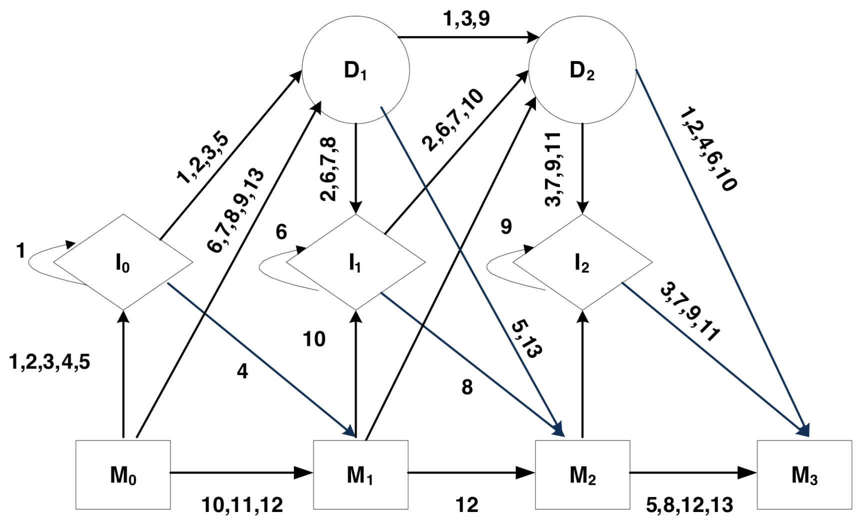


Figure 8: 4-State PHMM for Paths in Table 13

5 Implementation

Given a multiple sequence alignment (MSA) of opcodes, our objective is to generate a profile hidden Markov model (PHMM). We will then score sequences of both virus and non-virus code using the model, and tabulate the results.

A PHMM model was "trained" based on an MSA generated using opcodes sequences from virus files. These virus opcodes were generated using one of three virus construction kits: Virus Creation Laboratory (VCL32), Phalcon/Skism Mass-Produced Code Generator (PS-MPC) and the Next Generation Virus Creation Kit (NGVCK) (descriptions of these kits appear in Section 2.2). Each of these kits was used to generate multiple variants and grouped under a family.

As discussed above, a PHMM is specified by its emission and transition probabilities, on a per-state and per-opcode basis. The number of state probabilities depends on the gaps and symbols in a given MSA. A model can only be as strong as the given MSA, and an MSA with many gaps would be considered weak, since it will result in a model containing relatively few emission states.

The forward algorithm is used to score assembly files against a given PHMM. For our non-virus files, we have used "normal" programs—as specified below—which are available on many systems. These files are disassembled and all non-opcodes are filtered out before they are scored.

5.1 Test Data

Using three different construction kits we generated multiple variants for each. Our test data consisted of the following.

- 10 virus variants from VCL32 (labeled vcl32_01 to vcl32_10)
- 30 virus variants from PS-MPC (psmpc_01 to psmpc_30)
- 200 different variants from NGVCK (ngvck_001 to ngvck_200)
- For the "normal" files we used 40 disassembled cygwin version 1.5.19 dynamic link libraries (DLLs) (cygwin_01 to cygwin_40) and 30 disassembled DLLs from other non-virus programs such as Microsoft Office, Adobe, Internet Explorer, etc. (non_virus_01 to non_virus_30)

These construction kits were downloaded from VXHeaven. There are several versions of each of the kits available and we have used the latest and most stable version for our test data generation. Table 5 contains the release date and version of each of the kits used.

VCL32, PS-MPC and NGVCK all produce assembly code (asm) files depending on their settings and configurations. Although PS-MPC is capable of generating thousands of variants with different payloads, we only varied the most significant configuration options (memory resident, encryption, file type, etc.) to generate the variants. Similarly, with VCL and NGVCK, test data was generated with at least one of the various settings changed. As a result, we believe our detector will have to deal with the widest possible variations for each generator.

We used IDA Pro to disassemble the "normal" files (i.e., cygwin and other non-viruses in our test set) and to maintain consistency, we also used IDA Pro to disassemble the virus variants. Since the output of the virus kits was generated as assembly code, we used the Turbo Assembler (TASM 5.0) to assemble the files before disassembled them with IDA Pro. A virtual machine (VMWare Workstation) was used for all virus file processing and all of the virus code was deleted after we had extracted the desired opcode sequences.

All three construction kits we used generate 32-bit Windows PE executable files and each of these files can contain any of the 250 opcodes for the x86 processor. Allowing for all of the possible opcodes would make the PHMM emission and transition probability matrices unwieldy. In any case, only 14 opcodes make up the vast majority of the opcodes seen in programs—malware or normal code [3]. In our code samples, we found that an alphabet containing 36 different opcodes covered virtually all observed opcodes. We used the "*" character to represent any opcode not in our list of 36. The same 37-element alphabet—the 36 opcodes and the "*" character—was used for all of our experiments.

Each assembly file was filtered to obtain the opcode sequence; all other information was discarded. The resulting filtered files were used to generate the MSA and for scoring.

5.2 Training the Model

The multiple sequence alignments we used for our PHMMs were generated using the method discussed in Section 4.1. A PHMM model was created from the MSA, which contains data about opcode sequences for the virus family. As mentioned in Section 4, creating the MSA is essentially the training phase of constructing the PHMM. We then used the resulting PHMM to score opcode sequences.

Several models were generated for each virus family using distinct subsets of the available virus variants. For generating the MSAs (and consequently, the resulting PHMMs), we grouped the viruses as follows:

- VCL32 — 2 groups with 5 files in each group
- PS-MPC — 3 groups with 10 files in each group
- NGVCK — 10 groups with 20 files in each group

Note that these groups were selected at random from the available viruses.

The percentage of gaps in the MSAs is shown in Table 14. High gap percentages indicate that the resulting PHMM is unlikely to be effective, since the more gaps in the MSA, the more generic the resulting PHMM. A more generic model will not contain as much family-specific information, which will make scoring less reliable.

Virus Family	Gap percentage
VCL32	7.453
PS-MPC	23.555
NGVCK	88.308

Table 14: Gap Percentages for Virus Families

As can be seen from the results in Table 14, the NGVCK virus variants yield MSAs that are highly gapped and, therefore, we do not expect to obtain good results from the PHMM for NGVCK. The reason that NGVCK has such a high gap percentage is that its generator tends to move similar sections of code far from each other when it generates virus variants. Consequently, the MSA must contain a large number of gaps before these similar sections can be properly aligned. We will have more to say about NGVCK below.

5.3 VCL32 Example

Our "group 1" model for VCL32 was generated from five files (denoted vcl32_01 to vcl32_05). The resulting MSA has 1820 states and, for purposes of illustration, Table 15 contains the emission probabilities for states 126, 127 and 128, as calculated

	Emission Match Probabilities			Emission Insert Probabilities		
opcodes	State 126	State 127	State 128	State 126	State 127	State 128
and	0.0238	0.025	0.025	0.0612	0.0256	0.0256
inc	0.0238	0.025	0.025	0.0204	0.0256	0.0256
xor	0.0238	0.025	0.025	0.0204	0.0256	0.0513
stc	0.0238	0.025	0.025	0.0204	0.0256	0.0256
stosb	0.0238	0.025	0.025	0.0204	0.0256	0.0256
imul	0.0238	0.025	0.025	0.0204	0.0256	0.0256
jecxz	0.0238	0.025	0.025	0.0204	0.0256	0.0256
jmp	0.0238	0.025	0.025	0.0204	0.0256	0.0256
shl	0.0238	0.025	0.025	0.0204	0.0256	0.0256
not	0.0238	0.025	0.025	0.0204	0.0256	0.0256
add	0.0238	0.100	0.025	0.0612	0.0256	0.0256
stosd	0.0238	0.025	0.025	0.0204	0.0256	0.0256
call	0.0238	0.025	0.025	0.0612	0.0256	0.0256
jnz	0.0238	0.025	0.025	0.0204	0.0256	0.0256
push	0.0238	0.025	0.025	0.0204	0.0769	0.0513
cmp	0.0238	0.025	0.025	0.0204	0.0256	0.0256
dec	0.0238	0.025	0.025	0.0204	0.0256	0.0256
xchg	0.0238	0.025	0.025	0.0204	0.0256	0.0256
test	0.0238	0.025	0.025	0.0204	0.0256	0.0256
*	0.0238	0.025	0.025	0.0204	0.0256	0.0256
jb	0.0238	0.025	0.025	0.0204	0.0256	0.0256
sub	0.0238	0.025	0.025	0.0612	0.0256	0.0256
or	0.0238	0.025	0.025	0.0204	0.0256	0.0256
jz	0.0238	0.025	0.025	0.0204	0.0256	0.0256
neg	0.0238	0.025	0.025	0.0204	0.0256	0.0256
retn	0.0238	0.025	0.025	0.0204	0.0256	0.0256
lodsb	0.0238	0.025	0.025	0.0204	0.0256	0.0256
mov	0.1429	0.025	0.100	0.1020	0.0256	0.0256
pop	0.0238	0.025	0.025	0.0204	0.0256	0.0256
jnb	0.0238	0.025	0.025	0.0204	0.0256	0.0256
shr	0.0238	0.025	0.025	0.0204	0.0256	0.0256
stosw	0.0238	0.025	0.025	0.0204	0.0256	0.0256
lodsd	0.0238	0.025	0.025	0.0204	0.0256	0.0256
cld	0.0238	0.025	0.025	0.0204	0.0256	0.0256
rep	0.0238	0.025	0.025	0.0204	0.0256	0.0256
lea	0.0238	0.025	0.025	0.0204	0.0256	0.0256
rol	0.0238	0.025	0.025	0.0204	0.0256	0.0256

Table 15: Probabilities for VCL32 (group 1, states 126, 127 and 128)

from the MSA. For these probabilities, the add-one rule was used (see Section 4) which explains the small, constant, non-zero probabilities.

The transition probabilities between states 126, 127 and 128 for our group1 VCL32 files appear in Table 16.

	M_{127}	I_{127}	D_{127}		M_{128}	I_{128}	D_{128}
M_{126}	0.500	0.375	0.125	M_{127}	0.667	0.167	0.167
I_{126}	0.067	0.733	0.200	I_{127}	0.200	0.200	0.600
D_{126}	0.333	0.333	0.333	D_{127}	0.200	0.600	0.200

Table 16: VCL32 Transition Probabilities

From Table 16 we see that $a_{M_{126}M_{127}} = 0.5$ is the probability that M_{127} is reached after M_{126} emits a symbol, and we see that this probability is greater than the probability that I_{127} or D_{127} is reached (probability 0.375 and 0.125, respectively). Note that for each state, the sum of the probabilities in a row must be 1 since these numbers form a probability distribution.

5.4 Scoring with the Forward Algorithm

The forward algorithm is used to score a given sequence against a PHMM; see Section 4.3 for more details. Suppose we want to score a sequence $X = (x_1, x_2, \ldots, x_L)$ of length L using a PHMM with $N+1$ states. The states are associated with $0, 1, \ldots, N$, where states 0 and N are the begin and end states, respectively. Then scoring consists of the following steps.

- We calculate, in order, $F^M_{N-1}(L)$, $F^I_{N-1}(L)$ and $F^D_{N-1}(L)$.
- In the recursive process used to calculate $F^M_{N-1}(L)$, many other intermediate values are computed, including $F^M_{N-2}(L-1)$, $F^I_{N-1}(L-1)$, and so on. These values are saved for later use. After $F^D_{N-1}(L)$ has been calculated, most intermediate values are known, which makes scoring efficient.
- During the scoring calculation, some terms, such as $F^I_0(0)$, $F^M_0(2)$, are not defined. Whenever an undefined term is encountered, we simply exclude it from the calculation.
- The terms $F^M_{N-1}(L)$, $F^I_{N-1}(L)$ and $F^D_{N-1}(L)$ represent the scores for the sequence X up to state $N-1$; the product of these scores with their respective end transition probabilities gives the final score, that is,

$$\begin{aligned}\text{Score} = \log\big(&a_{M_{N-1}M_N}\exp(F^M_{N-1}(L)) + a_{I_{N-1}M_N}\exp(F^I_{N-1}(L)) \\ &+ a_{D_{N-1}M_N}\exp(F^D_{N-1}(L))\big)\end{aligned} \tag{2}$$

Since we have computed a log-odds score, it is not necessary to subtract any random or null model scores.

Figures 9 and 10 illustrates this recursive scoring process.

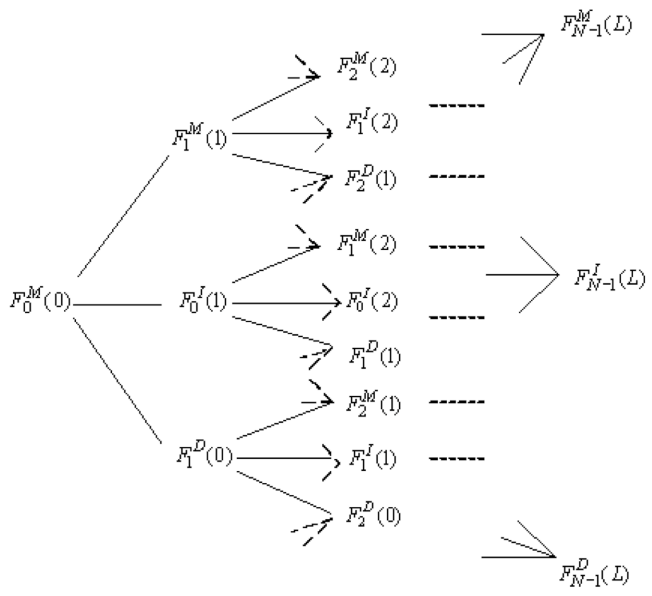


Figure 9: Forward Algorithm Recursion

$$\begin{tikzcd}
F_{N-1}^M(L) \arrow[dr, "a_{M_{N-1}M_N}"] & \\
F_{N-1}^I(L) \arrow[r, "a_{I_{N-1}M_N}"] & F_N^M(L) \\
F_{N-1}^D(L) \arrow[ur, "a_{D_{N-1}M_N}"'] &
\end{tikzcd}$$

Figure 10: Final Score

As described above, the scores depend on the length of the input sequence and, therefore, these scores cannot be used directly to compare sequence of different length. To eliminate this restriction, we simply divided the final score by the sequence length and thereby obtain a per-opcode score. Using this approach, all scores are per-opcode and we can now directly compare scores of sequences of different lengths.

There is one additional computational issue that arises when computing scores. Due to the logarithms that appear in equation (2), we did not have any underflow problems. However, the exponentiation part of the calculation leads to overflow problems. To overcome the overflow problem, we used the fact that [8]

$$\log(p+q) = \log(p) + \log(1 + \exp(\log(q) - \log(p))).$$

With this modification applied to the scoring calculations, exponentiation of large numbers does not create a problem.

The time complexity for this scoring algorithm is $O(nT)$, where n is the number of states and T is the length of the observed sequence. This complexity makes the algorithm competitive with other virus scanning techniques, such as change detection. However, this assumes that the code being analyzed has been disassembled. To be truly practical, the technique would most likely have to be applied directly to binary code, which is an area of future research.

6 Results

As discussed in the previous section, we use the forward algorithm (based on a PHMM, which is derived from an MSA), to score a given sequence of opcodes. The higher the resulting score, the more likely that the sequence of opcodes represents a virus in the same family as the PHMM. For testing purposes, we scored multiple non-viruses and multiple virus variants from each construction kit against our various PHMMs. The test data is described in Table 17.

Virus Family	Groups/Model Name	Files in Group
VCL32	vcl32_group5_1	vcl32_01 to vcl32_05
	vcl32_group5_2	vcl32_06 to vcl32_10
PS-MPC	psmpc_group10_1	psmpc_01 to psmpc_10
	psmpc_group10_2	psmpc_11 to psmpc_20
	psmpc_group10_3	psmpc_21 to psmpc_30
NGVCK	ngvck_group20_01	ngvck_01 to ngvck_020
	ngvck_group20_02	ngvck_021 to ngvck_040
	ngvck_group20_03	ngvck_041 to ngvck_060
	ngvck_group20_04	ngvck_061 to ngvck_080
	ngvck_group20_05	ngvck_081 to ngvck_100
	ngvck_group20_06	ngvck_101 to ngvck_120
	ngvck_group20_07	ngvck_121 to ngvck_140
	ngvck_group20_08	ngvck_141 to ngvck_160
	ngvck_group20_09	ngvck_161 to ngvck_180
	ngvck_group20_10	ngvck_181 to ngvck_200

Table 17: Test Data

For each model, the scoring threshold was taken as the minimum score for a virus in the same family as the model. If a log-odds score is greater than or equal to the threshold, we assume the program in question is a family virus, and if the score is below the threshold, the code is assumed to not belong to the virus family. Note that this threshold resulted in no false negative cases.

Figure 11 shows the scatter plot of scores for the vcl32_group5_1 model. No scores from any of the non-virus files exceed the minimum VCL32 family virus score of 1.0546 and, consequently, we have no errors in this particular case. The other VCL32 model performed equally well.

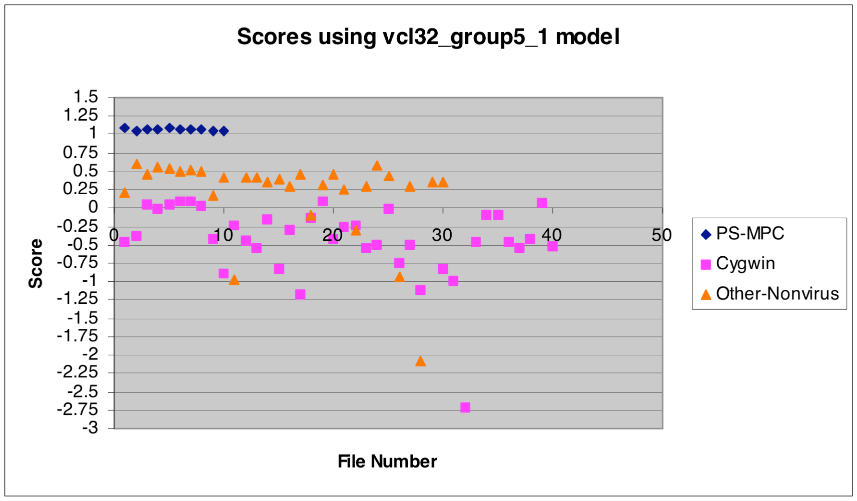


Figure 11: Scores for vcl32_group5_1 Model

The results for the group labeled psmpc_group10_1 appear in Figure 12. Again, we have no false positives or false negatives, and the same holds true for the other two models generated from the PS-MPC virus kit. The detection rate for both VCL32 and PS-MPC is 100%, regardless of the model used for scoring.

Based on the gap percentages (see Table 14), we expect NGVCK to be much more challenging for our PHMM-based detector. Figure 13 shows our scoring results using the model generated from the NGVCK virus group ngvcl_group20_01. Using the same thresholding as above, non-virus files that score greater than 0.715 are considered false positives. In this case, we have more false positives than not.

The high rate of false-positives for NGVCK is due to the aggressive subroutine permutation employed by the NGVCK construction kit. Since different variants have different subroutine order, the resulting MSA requires a high gap percentage to achieve a reasonable alignment and, as mentioned in Section 5.2, this results in a more "generic" model. The results in Figure 13 show that this NGVCK model lacks sufficient strength to be of any utility in distinguishing between family viruses and non-viruses. Similar results were obtained for the models generated from the other

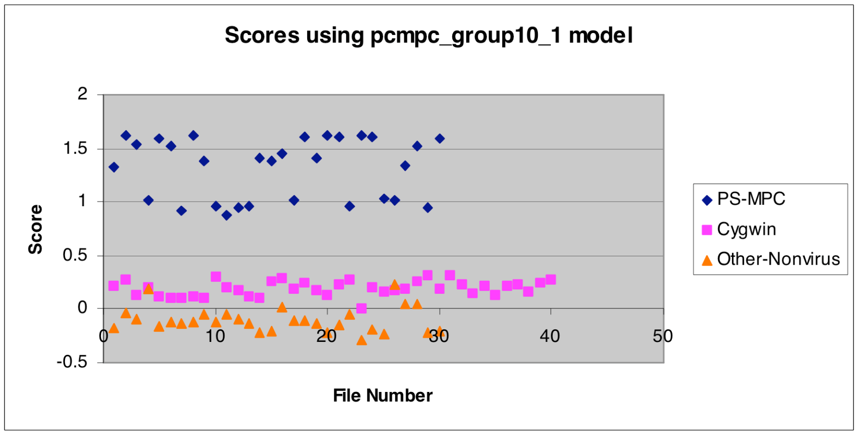


Figure 12: Scores for psmpc_group10_1 Model

NGVCK groups.

In an attempt to overcome this problem, we generated new models for NGVCK viruses using somewhat more fine-tuned MSAs. These MSAs were created by preprocessing the virus files by reordered the subroutines to reduce the number of gaps in the MSA. More details about this preprocessing step can be found in [21]. Note that any preprocessing step must also be applied when scoring files, which increases the scoring complexity.

For preprocessed NGVCK files, the MSA gap percentage decreased from 88.3% to 44.9%. We denote the models generated from these preprocessed NGVCK files as ngvck_pp_group20_01. Figure 14 shows the scores using the resulting model.

Note that by changing the threshold to allow for a few false positives, we could slightly improve the error rate. In any case, the error rate is far too high to be practical. Although our simple preprocessing step still yields an impractically high error rate, it does show that reductions in the gap percentage of the MSA can lead to dramatic improvements in scoring. For more details on all of the test cases studied, see [1].

7 Conclusions and Future Work

Hidden Markov models were developed by mathematicians in the late 1960s and since the 1980s HMMs have been applied to effectively solve many challenging computer science problems. In fact, today, HMMs are considered a standard machine learning technique. Recently, biologists have developed profile hidden Markov models to tackle some of the most difficult problems in bioinformatics. PHMMs can be viewed as a highly specialized form of HMMs designed to deal with evolutionary processes. In this paper, we have turned the tables by applying PHMMs to a challenging problem

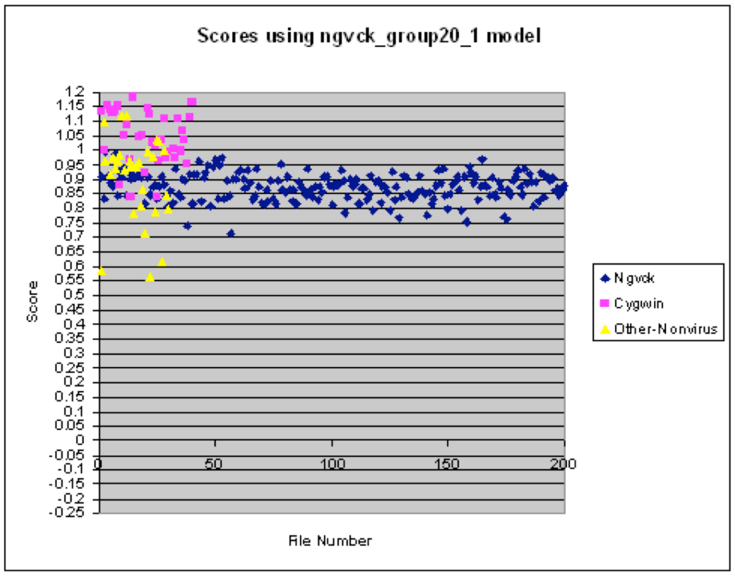


Figure 13: Scores for ngvck_group20_01 Model

in computer science, namely, the problem of metamorphic virus detection.

Profile hidden Markov models have proven to be a valuable tool for determining relations between DNA and protein sequences. In this paper, we have discussed our work aimed at quantifying the effectiveness of PHMMs for detecting metamorphic viruses. We tested our PHMM method on three virus construction kits—VCL32, PS-MPC and NGVCK. The results showed a 100% detection rate for VCL32 and PS-MPC. After fine tuning of the multiple sequence alignment phase, we were still unable to detect NGVCK viruses at a rate that would be useful in practice.

Our detection rates indicate that PHMMs are well suited for certain types of metamorphic malware but, perhaps, not well suited for others. More precisely, PHMMs can be highly effective when a virus family does not shift corresponding blocks of code "too far" apart, whereas standard HMMs appear to work well regardless of such shifting [39]. This "weakness" of PHMMs is not particularly surprising, given that PHMMs take positional information into account, while standard HMMs do not. However, this additional positional information should provide for more robust scoring in appropriate cases.

The following would be useful extensions to this study of metamorphic virus detection.

- To fine tune the models to match the given family opcode sequences, it would be useful to employ Baum-Welch re-estimation to the model obtained from the MSA.

- We trained our models using the entire opcode sequence of each virus. This

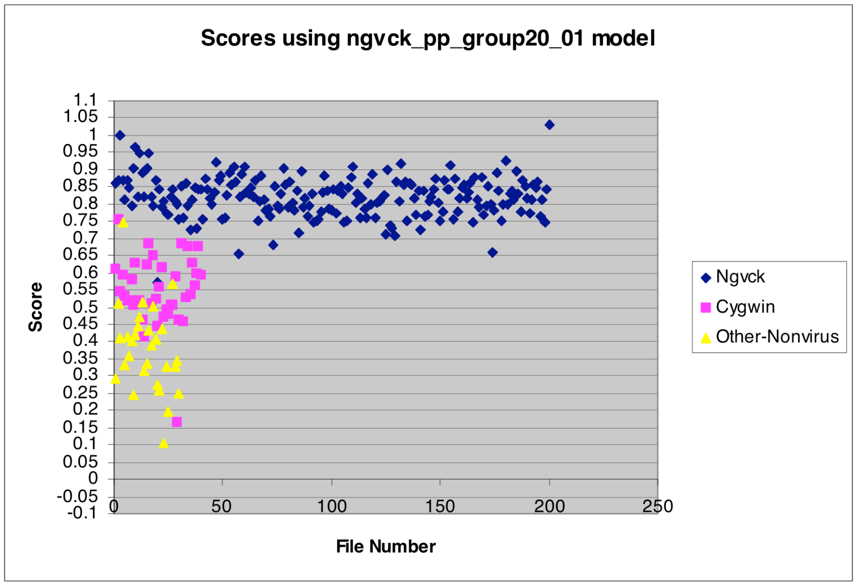


Figure 14: Scores for ngvck_pp_group20_01 Model

could be modified to model each subroutine independently. Such subroutine modeling might enable us to better detect metamorphic viruses that implement subroutine permutations (e.g., NGVCK) or more general code reordering.

- The time taken to preprocess the data (i.e., disassemble the code), makes our approach somewhat impractical. In principle, it should be possible to train on the binary executable files. It would be very interesting to see how PHMMs and standard HMMs perform when the binary code is processed directly.

Finally, we note that it has become fashionable to apply biological reasoning and modeling techniques to information security problems [15]. In this paper, we have shown that, at least in some cases, a technique developed specifically to solve problems in bioinformatics can be highly effective in an information security context. From the perspective of computer virus detection, the work presented here could be viewed as supporting evidence of the trend of looking to biology for information security solutions. However, from the virus writer's perspective, the work here could be interpreted as a cautionary tale against applying biological analogies to literally.

Acknowledgment

The authors thank the anonymous referees whose insightful and extensive comments greatly improved this paper in every respect.

References

[1] S. Attaluri, Profile hidden Markov models for metamorphic virus analysis, M.S. report, Department of Computer Science, San Jose State University, 2007. `http://www.cs.sjsu.edu/faculty/stamp/students/Srilatha_cs298Report.pdf`

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

[3] D. Bilar, Statistical structures: fingerprinting malware for classification and analysis, `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Bilar.pdf`

[4] J.-M. Borello and L. Mé, Code obfuscation techniques for metamorphic viruses, to appear in *Journal in Computer Virology.*

[5] D. Bruschi, L. Martignoni, and M. Monga, Using code normalization for fighting self-mutating malware, *Proceedings of the International Symposium of Secure Software Engineering*, ISSSE, Arlington, Virginia, USA, March 2006

[6] T.-C. Chiueh, A look at current malware problems and their solutions, `http://www.cs.sjsu.edu/~stamp/IACBP/IACBP08/Tzi-cker%20Chiueh/2008.ppt`

[7] C. Collberg, C. Thomborson and D. Low, A taxonomy of obfuscating transformations. `http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html`

[8] R. Durbin, S. Eddy, A. Krogh and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*, Cambridge University Press, 1988.

[9] S. R. Eddy, Profile hidden Markov models, *Bioinformatics*, Vol. 14, No. 9, July 1998. pp. 755–763.

[10] D.-F. Feng, and R. F. Doolittle, 1987. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Biology and Evolution* 13:93-104.

[11] P. Ferrie, Look at that escargot, *Virus Buletin*, December 2004, pp. 4–5, `http://pferrie.tripod.com/papers/gastropod.pdf`

[12] P. Ferrie, Hidan and dangerous, *Virus Bulletin*, March 2007, pp. 14–19.

[13] E. Filiol, Metamorphism, formal grammars and undecidable code mutation, *International Journal of Computer Science*, Vol. 2, No. 1, 2007, pp. 70–75.

[14] R.G. Fiñones and R. Fernandez, Solving the metamorphic puzzle, *Virus Bulletin*, March 2006, pp. 14–19.

[15] S. Forrest, Computer immune systems, `http://www.cs.unm.edu/~immsec/papers.htm`

[16] M. Jordan, Anti-virus research—dealing with metamorphism, *Virus Bulletin*, October 2002, `http://ca.com/us/securityadvisor/documents/collateral.aspx?cid=48051`

[17] S.Khuri, Hidden Markov models, lecture notes. `http://www.cs.sjsu.edu/faculty/khuri/Bio_CS123B/Markov.pdf`.

[18] A. Krogh, An introduction to hidden Markov models for biological sequences, Center for Biological Sequence Analysis, Technical University of Denmark, 1988.

[19] A. Marinescu, An analysis of Simile, SecurityFocus.com, March 2003, `http://www.securityfocus.com/infocus/1671`

[20] J. McAfee and C. Haynes, *Computer Viruses, Worms, Data Diddlers, Killer Programs and Other Threats to Your System*, St. Martin's Press, 1989.

[21] S. McGhee, Pairwise alignment of metamorphic computer viruses, M.S. report, Department of Computer Science, San Jose State University, 2007, `http://www.cs.sjsu.edu/faculty/stamp/students/mcghee_scott.pdf`

[22] D.W. Mount, Bioinformatics: sequence and genome analysis, Cold Spring Harbor Laboratory, 2004.

[23] J. Munro, Antivirus research and detection techniques, *ExtremeTech*, July 2002. `http://findarticles.com/p/articles/mi_zdext/is_200207/ai_ziff28916`

[24] Netlux, `http://vx.netlux.org/vx.php?id=tp00`.

[25] OpenRCE.org, The molecular virology of lexotan32: metamorphism illustrated, August 2007, `http://www.openrce.org/articles/full_view/29`.

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

[27] Orr, The molecular virology of Lexotan32: Metamorphism illustrated, 2007, `http://www.antilife.org/files/Lexo32.pdf`

[28] W.T. Polk, L.E. Bassham, J.P. Wack and L.J. Carnahan, *Anti-virus Tools and Techniques for Computer Systems*, Noyes Data Corporation, 1995.

[29] Prim's Algorithm, `http://en.wikipedia.org/ wiki/Prim\%27s_algorithm`

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

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

[32] M. Stamp, *Information Security: Principles and Practice*, Wiley–Interscience, August 2005.

[33] Symantec, `http://www.symantec.com/security_response/writeup.jsp?docid=2000-122010-0045-99&tabid=2`

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

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

[36] VXHeavens, `http://vx.netlux.org/`

[37] A. Walenstein, R. Mathur, M.R. Chouchane and A. Lakhotia, Normalizing metamorphic malware using term rewriting, *Proceedings of the International Workshop on Source Code Analysis and Manipulation* (SCAM), IEEE CS Press, September 2006, pp. 75–84.

[38] Wikipedia, `http://en.wikipedia.org/wiki/Timeline_of_notable_computer_viruses_and_worms`.

[39] W. Wong and M. Stamp, Hunting for metamorphic engines, *Journal in Computer Virology*, Vol. 2, No. 3, December 2006, pp. 211–219

[40] ZDNet, Ex-virus writer questioned over Slammer, `http://news.zdnet.co.uk/security/0,1000000189,39175383,00.htm`