

Dueling hidden Markov models for virus analysis

Ashwin Kalbhor · Thomas H. Austin · Eric Filiol ·
Sébastien Josse · Mark Stamp

Received: 2 July 2014 / Accepted: 11 November 2014 / Published online: 30 November 2014
© Springer-Verlag France 2014

Abstract Recent work has presented hidden Markov models (HMMs) as a compelling option for malware identification. However, some advanced metamorphic malware like MetaPHOR and MWOR have proven to be more challenging to detect with these techniques. In this paper, we develop the *dueling HMM Strategy*, which leverages our knowledge about different compilers for more precise identification. We also show how this approach may be combined with previous techniques to minimize the performance overhead. Additionally, we examine the HMMs in order to identify the meaning of these hidden states. We examine HMMs for four different compilers, hand-written assembly code, three virus construction kits, and two metamorphic malware families in order to note similarities and differences in the hidden states of the HMMs.

1 Introduction

Wong and Stamp [30] have shown that tools based on hidden Markov models (HMMs) are effective at detecting metamorphic computer viruses. This paper explores these tools in more depth to better understand the meaning of the hidden states in these models.

A. Kalbhor · T. H. Austin (✉) · M. Stamp
Department of Computer Science, San José State University,
San Jose, CA, USA
e-mail: thomas.austin@sjsu.edu

E. Filiol
ESIEA Laboratoire (C + V)^O, Laval, France

S. Josse
Direction générale de l'armement (DGA), Rennes, France
e-mail: sebastien.josse@polytechnique.edu

In other domains, the states of an HMM have been connected with some fundamental aspects of the problem at hand. For instance, Cave and Neuwirth [5] reveal that an HMM with two hidden states for the English (written) language corresponds to vowels and consonants. This paper attempts to reveal details about the hidden states and determine what insights they might provide about assembly code in general, and virus code in particular.

A key insight is that virus construction kits and metamorphic code are essentially another type of compiler. Our tests build models for four different compilers, for hand-written (benign) assembly code, for three virus construction kits, and for two metamorphic malware families. We identify salient points of our models, noting how hand-written assembly differs from compiled code and how benign code differs from virus code.

We leverage this understanding of different models to more effectively detect computer viruses. The traditional approach uses a hidden Markov model of virus code and flags a file as infected if it exceeds a given threshold [30]. Instead, we test the file against several different HMMs and flag the file as a virus only if the virus HMM reports the highest probability of observing the given file. We dub this approach *the dueling HMM strategy*, evoking the notion that the different HMMs are competing against one another. Our results show that the dueling HMM strategy achieves superior results to the threshold-based technique, and is often effective at identifying viruses. While multiple HMMs have been leveraged in other areas such as intrusion detection [8], this approach has not previously been applied to virus identification.

This paper expands upon a previous conference paper [3] to include analysis of additional sources of benign code and additional virus families, including the MWOR worm [22] that is specifically designed to evade the detection technique used by Wong and Stamp. We also show how the threshold

approach may be combined with the dueling HMM strategy to reduce the performance overhead of the dueling HMM strategy with no reduction in the accuracy of the results.

The contributions of this paper are as follows:

- We explore the semantic meaning behind the hidden states of the hidden Markov model.
- We demonstrate the effectiveness of HMMs in distinguishing between different compilers.
- We develop the dueling HMM strategy, a novel technique for using multiple HMMs in virus identification.
- We develop a tiered HMM strategy that combines the threshold approach with the dueling HMM strategy, gaining the benefits of both techniques.

1.1 Polymorphic viruses, metamorphic viruses, and virus construction kits

Signature-based detection is the primary method of identifying computer viruses [29]. However, virus makers have been resourceful, and have developed a variety of countermeasures. One early approach used by virus writers was to encrypt the body of the virus code. However, this technique could often be defeated by looking for the signature of the encryptor itself [29]. Polymorphic code defeats this detection technique by mutating the code responsible for encryption. Antivirus detection can still identify these programs by scanning decrypted data for the virus signature.

Metamorphic viruses extend polymorphic techniques to transform the entire virus, thereby defeating signature-based detection approaches. Compounding the danger, virus construction kits have been created that make it easy for people with limited technical ability to create sophisticated viruses. Other threats such as evolvable malware [17] still remain theoretical, but might further complicate virus detection.

Research shows that better tools for virus detection are needed to handle these threats. Christodorescu and Jha [9] test different malware detectors and show that many commercial products are ill-equipped to handle code obfuscation techniques. Kruegel et al. [19] use control-flow graphs to detect polymorphic/metamorphic worms. Bruschi et al. [4] use this technique to normalize programs and compare the results, testing their technique against the MetaPHOR virus. Mohammed [24] uses *zeroing transformations*, which perform a series of transformations on a program to convert it to a “zero form”. Signature-based methods can then be used on the zero form program.

Leder et al. [20] use *value set analysis*, performing a static flow analysis and check for values that are characteristic of a piece of malware. Zhang and Reeves [31] statically analyze programs to compare semantics based on the pattern of library calls. Christodorescu et al. [10] identify polymor-

phic/metamorphic malware by considering the semantics of programs.

Hidden Markov models use a statistical approach to identify these viruses. Wong and Stamp [30] use HMMs to identify viruses from different virus construction kits (VCKs) with a high degree of accuracy. Attaluri et al. [2] consider the application of profile hidden Markov models, which consider positional information. Their results show that positional HMMs can be effective for detecting certain types of metamorphic viruses, but do not perform well when viruses shift blocks of code far apart. Annachhatre et al. [1] use HMM analysis for classifying malware. Filiol and Josse [14] discuss the application of Bayesian techniques to detecting metamorphic viruses, considering both naive Bayes and HMMs.

Chess and White [7] show that there are computer viruses that no algorithm can detect. Song et al. [26] highlight the challenges that polymorphic techniques present to signature-based approaches and any generative approaches to producing malicious code. Filiol and Josse [13] discuss statistical testing simulability and show how attackers can evade detection by exploiting the defender’s detection model. Lin [21] explores this idea further by creating viruses specifically designed to avoid HMM-based detection. In short, a metamorphic virus can be designed to select mutations only if the mutations will make the program appear to be more like a benign program. Madenur Sridhara and Stamp [22] utilize this strategy in the design of MWOR, a metamorphic worm that uses dead code insertion to evade an HMM-based virus detection tool.

2 Background on hidden Markov models

Markov models are state machines where the current state depends on some number of the previous states in a probabilistic way. A Markov model of order one depends only on the previous state, a Markov model of order two depends on the previous two states, etc. A *hidden* Markov model is a Markov model where the state is not directly observable. To better illustrate how these models work, we begin with a motivating example.¹

2.1 Hidden Markov model example

Suppose that we want to go see a movie. For simplicity, we will consider movies to either be “Good” (G) or “Uninteresting” (U).

From experience, we know that the probability of a given director releasing a good movie after another good movie is 0.7, and the probability of a bad movie after another bad

¹ An expanded version of this section discussing hidden Markov models is available at <http://www.cs.sjsu.edu/~stamp/RUA/HMM>.

movie is 0.6. Summarizing this information gives us the following table:

$$\begin{matrix} & G & U \\ G & \begin{bmatrix} 0.7 & 0.3 \end{bmatrix} \\ U & \begin{bmatrix} 0.4 & 0.6 \end{bmatrix} \end{matrix} \tag{1}$$

If we have seen all of the previous movies, and hence could observe the states, we could use this information to directly build a Markov model. Alas, academic life has kept us busy, so instead we must rely on the opinions of movie critics.

The *observations* available to us are the reviews of the movie critics. The movie critics may despise (D) a movie, think that a movie is mediocre (M), or love (L) a movie. Based on our experience, we determine the probability of the critics’ reviews given the quality of the movie:

$$\begin{matrix} & D & M & L \\ G & \begin{bmatrix} 0.1 & 0.4 & 0.5 \end{bmatrix} \\ U & \begin{bmatrix} 0.7 & 0.2 & 0.1 \end{bmatrix} \end{matrix} \tag{2}$$

With the information from (1), we can construct the state transition matrix:

$$A = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \tag{3}$$

Similarly from (2), we produce the following observation matrix:

$$B = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{bmatrix} \tag{4}$$

We also need to know the initial state distribution. In this example, suppose that is is

$$\pi = [0.6 \quad 0.4] \tag{5}$$

The matrices π , A and B are *row stochastic*, meaning that each element is a probability and the elements of each row sum to 1, that is, each row is a probability distribution.

Now consider the release of 4 consecutive movies by the same director. Reading the reviews, we observe D, M, D, L . Letting 0 represent D , 1 represent M and 2 represent L , this observation sequence is

$$\mathcal{O} = (0, 1, 0, 2). \tag{6}$$

We might want to determine the most likely state sequence of the Markov process given the observations (6). In other words, what is the most likely quality of recent movie releases. “Most likely” has a couple of possible interpretations. In this context, we will define “most likely” to mean

Table 1 HMM notation

Notation	Explanation
T	Length of the observation sequence
N	Number of states in the model
M	Number of observation symbols
Q	Distinct states of the Markov process, q_0, q_1, \dots, q_{N-1}
V	Possible observations, assumed to be $0, 1, \dots, M - 1$
A	State transition probabilities
B	Observation probability matrix
π	Initial state distribution
\mathcal{O}	Observation sequence, $\mathcal{O}_0, \mathcal{O}_1, \dots, \mathcal{O}_{T-1}$

the state sequence that maximizes the expected number of correct states.²

2.2 Notation

The notation used in an HMM is summarized in Table 1. Note that the observations are assumed to come from the set $\{0, 1, \dots, M - 1\}$, which simplifies the notation with no loss of generality. That is, we simply associate each distinct observation with one of the elements $0, 1, \dots, M - 1$, so that $\mathcal{O}_i \in V = \{0, 1, \dots, M - 1\}$ for $i = 0, 1, \dots, T - 1$.

For the movie example of the previous section—with the observations sequence given in (6)—we have $T = 4, N = 2, M = 3, Q = \{H, C\}, V = \{0, 1, 2\}$ (where we let 0, 1, 2 represent “despised”, “mediocre”, and “loved” reviews, respectively). In this case, the matrices A, B , and π are given by (3), (4), and (5), respectively.

The matrix $A = \{a_{ij}\}$ is $N \times N$ with

$$a_{ij} = P(\text{state } q_j \text{ at } t + 1 \mid \text{state } q_i \text{ at } t)$$

and A is row stochastic. Note that the probabilities a_{ij} are independent of t . The matrix $B = \{b_j(k)\}$ is an $N \times M$ with

$$b_j(k) = P(\text{observation } k \text{ at } t \mid \text{state } q_j \text{ at } t).$$

As with A , the matrix B is row stochastic and the probabilities $b_j(k)$ are independent of t . The unusual notation $b_j(k)$ is standard in the HMM world.

An HMM is defined by A, B , and π (and, implicitly, by the dimensions N and M). The HMM is denoted by $\lambda = (A, B, \pi)$.

Consider a generic state sequence of length four

$$X = (x_0, x_1, x_2, x_3)$$

² Alternately, we could reasonably define “most likely” as the state sequence with the highest probability from among all possible state sequences. Dynamic programming (DP) can be used to efficiently find this particular solution. Note that the DP solution and the HMM solution are not necessarily the same.

Table 2 State sequence probabilities

State	Probability	Normalized probability
<i>GGGG</i>	0.000412	0.042787
<i>GGGU</i>	0.000035	0.003635
<i>GGUG</i>	0.000706	0.073320
<i>GGUU</i>	0.000212	0.022017
<i>GUGG</i>	0.000050	0.005193
<i>GUGU</i>	0.000004	0.000415
<i>GUUG</i>	0.000302	0.031364
<i>GUUU</i>	0.000091	0.009451
<i>UGGG</i>	0.001098	0.114031
<i>UGGU</i>	0.000094	0.009762
<i>UGUG</i>	0.001882	0.195451
<i>UGUU</i>	0.000564	0.058573
<i>UUGG</i>	0.000470	0.048811
<i>UUGU</i>	0.000040	0.004154
<i>UUUG</i>	0.002822	0.293073
<i>UUUU</i>	0.000847	0.087963

with corresponding observations

$$\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3).$$

Then π_{x_0} is the probability of starting in state x_0 . Also, $b_{x_0}(\mathcal{O}_0)$ is the probability of initially observing \mathcal{O}_0 and a_{x_0,x_1} is the probability of transiting from state x_0 to state x_1 . Continuing, we see that the probability of the state sequence X is given by

$$P(X) = \pi_{x_0} b_{x_0}(\mathcal{O}_0) a_{x_0,x_1} b_{x_1}(\mathcal{O}_1) a_{x_1,x_2} b_{x_2}(\mathcal{O}_2) a_{x_2,x_3} b_{x_3}(\mathcal{O}_3). \tag{7}$$

Consider again the movie example in Sect. 2.1 with observation sequence $\mathcal{O} = (0, 1, 0, 2)$, as given in (6). Using (7) we can compute, say,

$$\begin{aligned} P(GGUU) &= 0.6(0.1)(0.7)(0.4)(0.3)(0.7)(0.6)(0.1) \\ &= 0.000212 \end{aligned}$$

Similarly, we can directly compute the probability of each possible state sequence of length four, assuming the given observation sequence (6). We have listed these results in Table 2, where the probabilities in the last column are normalized so that they sum to 1.

To find the optimal³ state sequence, we choose the most probable symbol at each position. We sum the probabilities in Table 2 that have an *G* in the first position. Doing so, we find the (normalized) probability of *G* in the first position

³ In the dynamic programming (DP) sense, we would simply choose the sequence with the highest probability, namely *UUUG*. Note that this differs from the optimal solution in the HMM sense.

Table 3 HMM probabilities

	Element			
	0	1	2	3
$P(G)$	0.188182	0.519576	0.228788	0.804029
$P(U)$	0.811818	0.480424	0.771212	0.195971

is 0.18817, and hence the probability of *U* in the first position is 0.81183. The HMM therefore chooses the first element of the optimal sequence to be *U*. We repeat this for each element of the sequence, obtaining the probabilities in Table 3.

From Table 3 we find that the optimal sequence is *UGUG*.

2.3 Applying hidden Markov models

There are three fundamental problems that we can solve using HMMs:

1. Given an HMM and a sequence of observations, determine the likelihood of the observed sequence. Returning to our movie example from Sect. 2.1, this might be useful for determining the director of a series of movies.
2. Given an HMM and an observation sequence, find the optimal state sequence. In this case, we are trying to uncover the hidden part of our HMM. We discuss this type of problem in some detail in Sect. 2.1.
3. Given an observation sequence \mathcal{O} and the dimensions N and M , find the model $\lambda = (A, B, \pi)$ that maximizes the probability of \mathcal{O} . This can be viewed as training a model to best fit the observed data.

Item 1 is the problem we are trying to solve with virus detection, so we focus our discussion here; for more details on the other problems, we refer the interested reader to [27].

Let $\lambda = (A, B, \pi)$ be a given model and let \mathcal{O} be a series of observations where $\mathcal{O} = (\mathcal{O}_0, \mathcal{O}_1, \dots, \mathcal{O}_{T-1})$. We want to find $P(\mathcal{O} | \lambda)$.

Let $X = (x_0, x_1, \dots, x_{T-1})$ be a state sequence. Then by the definition of B we have

$$P(\mathcal{O} | X, \lambda) = b_{x_0}(\mathcal{O}_0) b_{x_1}(\mathcal{O}_1) \dots b_{x_{T-1}}(\mathcal{O}_{T-1})$$

and by the definition of π and A it follows that

$$P(X | \lambda) = \pi_{x_0} a_{x_0,x_1} a_{x_1,x_2} \dots a_{x_{T-2},x_{T-1}}.$$

Since

$$P(\mathcal{O}, X | \lambda) = \frac{P(\mathcal{O} \cap X \cap \lambda)}{P(\lambda)}$$

and

$$\begin{aligned}
 P(\mathcal{O} | X, \lambda)P(X | \lambda) &= \frac{P(\mathcal{O} \cap X \cap \lambda)}{P(X \cap \lambda)} \cdot \frac{P(X \cap \lambda)}{P(\lambda)} \\
 &= \frac{P(\mathcal{O} \cap X \cap \lambda)}{P(\lambda)}
 \end{aligned}$$

we have

$$P(\mathcal{O}, X | \lambda) = P(\mathcal{O} | X, \lambda)P(X | \lambda).$$

By summing over all possible state sequences we obtain

$$\begin{aligned}
 P(\mathcal{O} | \lambda) &= \sum_X P(\mathcal{O}, X | \lambda) \\
 &= \sum_X P(\mathcal{O} | X, \lambda)P(X | \lambda) \\
 &= \sum_X \pi_{x_0} b_{x_0}(\mathcal{O}_0) a_{x_0, x_1} b_{x_1}(\mathcal{O}_1) \dots \\
 &\quad a_{x_{T-2}, x_{T-1}} b_{x_{T-1}}(\mathcal{O}_{T-1}).
 \end{aligned}$$

However, this direct computation is generally infeasible, since it requires about $2TN^T$ multiplications. The strength of HMMs derives largely from the fact that there exists an efficient algorithm to achieve the same result.

To find $P(\mathcal{O} | \lambda)$, the so-called *forward algorithm*, or α -pass, is used. For $t = 0, 1, \dots, T-1$ and $i = 0, 1, \dots, N-1$, define

$$\alpha_t(i) = P(\mathcal{O}_0, \mathcal{O}_1, \dots, \mathcal{O}_t, x_t = q_i | \lambda). \tag{8}$$

Then $\alpha_t(i)$ is the probability of the partial observation sequence up to time t , where the underlying Markov process is in state q_i at time t .

The crucial insight here is that the $\alpha_t(i)$ can be computed recursively as follows.

1. Let $\alpha_0(i) = \pi_i b_i(\mathcal{O}_0)$, for $i = 0, 1, \dots, N-1$
2. For $t = 1, 2, \dots, T-1$ and $i = 0, 1, \dots, N-1$, compute

$$\alpha_t(i) = \left[\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji} \right] b_i(\mathcal{O}_t)$$

3. Then from (8) it is clear that

$$P(\mathcal{O} | \lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i).$$

The forward algorithm only requires about N^2T multiplications, as opposed to more than $2TN^T$ for the naïve approach.

3 Dueling HMM strategy

A central contribution of this paper is a novel method of applying hidden Markov models to virus identification. The *dueling HMM strategy* differs from traditional HMM-based approaches in that it leverages HMMs of benign code, rather

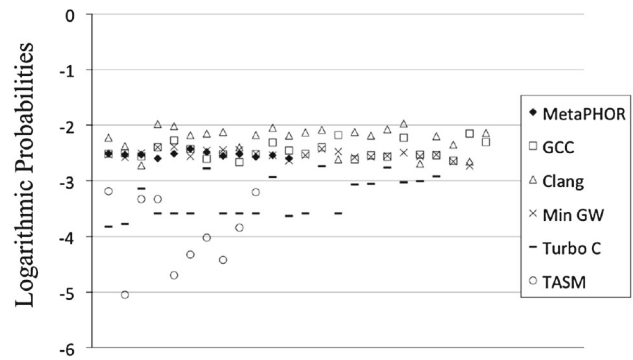


Fig. 1 Probabilities of a match with the MetaPHOR HMM

than relying on a single HMM of the target virus family. While there is an additional performance penalty, it appears to achieve more accurate results.

The standard application of HMMs to virus identification works as follows:

1. Build an HMM from virus code.
2. Determine the proper “threshold value”.
3. For any new file, determine the probability of observing the given sequence of opcodes, normalized for the length of the observation. If the probability is less than the threshold value, the file is flagged as a virus.

There are several benefits to this approach. Since only a single HMM is required, the analysis can be performed more efficiently. Also, it is straightforward to adjust the threshold value in order to set the desired balance between false positives and false negatives.

Rather than rely on threshold values, the dueling HMM strategy uses the following process:

1. Build N HMMs of benign code, representing code compiled by different compilers.
2. Build M HMMs of virus code, representing the different viruses to identify.
3. For any new file, determine the probability of observing the sequence of opcodes *for each of the $N + M$ HMMs*.
4. If the HMM reporting the highest probability represents virus code, the file is flagged as a virus.

This approach takes more overhead, but the benefit of leveraging information about different compilers allows for a more fine-grained analysis, and seems to achieve superior results.

It is illuminating to compare the two approaches in identifying MetaPHOR-infected files, discussed in Sect. 8.1. Figure 1 shows the distribution of log probabilities reported for one test of the 4-state HMM built from MetaPHOR-infected code. The black diamonds represent probabilities for

different MetaPHOR-infected files. The other shapes represent benign programs built with different compilers, outlined in Sect. 4. The traditional, threshold-based approach would draw a horizontal line across the diagram representing the threshold; ideally, all black diamonds should be above the threshold line and all other shapes should be below. The results highlight the difficulty of determining a threshold value that does not have a high number of false positives for some compiler.

The dueling HMM strategy includes additional HMMs, representing different compilers, hand-written assembly, and other models of benign code. With this approach, the accuracy is greatly improved. As shown in Sect. 8.1, the dueling HMM strategy identifies 85 % of our sample MetaPHOR-infected files, with only about a 1 % false positive rate. In contrast, setting a threshold value to detect MetaPHOR with a comparable level of false negatives results in a false positive rate in excess of 50 %.

With the dueling HMM strategy, there is no threshold value. A downside of this strategy is that it is not straightforward to adjust the balance between false positives and false negatives. However, introducing a bias to the results in favor of some HMMs provides this flexibility. In Sect. 7.1, we illustrate how to add a bias to the results.

4 Models for different compilers

A focus of our work is to identify the tools used to build a specific program. Our initial tests are designed around identifying the underlying compiler, since the vast majority of benign programs are likely to be compiled from a higher level language.

We use four different compilers for our tests. These include Gnu's venerable GCC compiler [16], the Clang [11] front-end for the LLVM project, the MinGW port of GCC to Windows, and the Turbo C compiler. We use the JAHMM toolkit [15] and code from <http://www.c.happycodings.com> to both train and test our models.

4.1 Using compiler generated assembly

Our initial models are constructed using assembly code generated directly by the compilers. In this section, we only consider the GCC and Clang models. All programs were compiled to assembly on an Apple Mac OS X laptop running version 10.6.8.

Following Wong and Stamp [30], we consider only the Intel x86 (also known as IA-32) operation codes (opcodes) for these models. The assembly generated by these two compilers is substantially different in the use of opcodes. In fact, it is sufficient to search for the presence of a few specific opcodes to conclusively identify the compiler. For instance,

the CALL opcode occurs frequently in the GCC-generated assembly code, but never in the Clang assembly, which uses CALLQ instead.

HMMs are not especially useful in this case. However, a view of some of the models is illuminating. Figure 2 shows HMMs with 4 hidden states generated for the GCC compiler on the left and the Clang compiler on the right.

HMMs do not always have a single starting state, and instead have probabilities for starting in each state [27]. However, both of these models start in a specific state with 100 % probability. This pattern held with many of the HMMs that we develop in this paper.

The HMMs show a remarkable similarity in their structure. For both models, the initial state is always dominated by the observation of MOVQ and MOVL opcodes. A second state is made up almost exclusively of MOVSD observations.

The remaining states show more variety. Both State 2 in the GCC HMM and State 6 in the Clang model have a high probability for observing JMP, RET, and conditional jump opcodes. However, State 2 also has a high probability of observing the LEAVE opcode.

State 3 of the GCC HMM is dominated by observations of the CALL opcode. However, it also contains some probability of observing conditional jump opcodes. In contrast, State 7 of the Clang HMM has almost 100 % probability of observing CALLQ.

While these models show some interesting details about how the assembly code is generated, in anti-virus detection we are unlikely to receive the original assembly code. Instead, we will be presented with executables that we will first need to disassemble before we will be able to do any significant analysis. The resulting assembly code is significantly different than that generated by the compilers themselves. In the next section, we will explore the models built from assembly generated from a disassembler.

4.2 Using disassembled assembly

As Wong and Stamp observe [30], a more realistic model for generating assembly in antivirus detection compiles the source code and then disassembles the resulting binaries. For these tests, we used IDA Pro version 6.2.111006 as our disassembler.

The resulting assembly code is markedly different from the assembly code produced by the compilers themselves. As a result, identifying the original compiler becomes somewhat more complicated.

In this section, we develop HMMs for four different compilers. The models are shown in Fig. 3.

Four states seems to be the optimal number of states, determined by our testing in Sect. 4.3. All four models seem to have the same basic structure. The four states can roughly be described by the opcode most likely to be observed when in

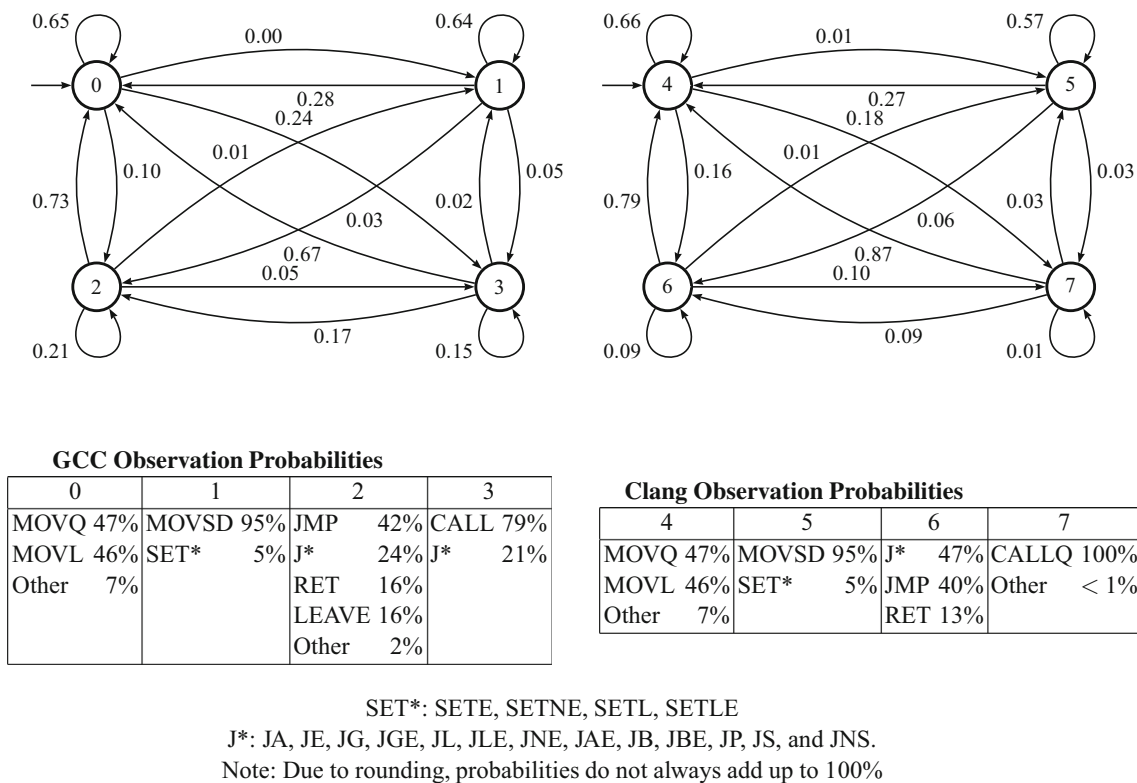


Fig. 2 GCC HMM and clang HMM with 4 hidden states from compiler generated assembly

that state: the PUSH state; the MOV state; the CALL state; and the miscellaneous state.

The PUSH state always includes POP and RETN as significant opcodes. The odds of starting in this state are 100% with the GCC, Clang, and MinGW HMMs.

The MOV state always includes a significant amount of JMP and conditional jump operations, and usually has a high probability of observing the LEA opcode.

The CALL state observes CMP and ADD and conditional jump opcodes with a high probability.

The final state is not dominated by any observation, though TEST, SUB, and XOR are common.

The model for GNU’s Compiler Collection (GCC), version 4.2.1 on OS X, is shown in the top left corner of Fig. 3. GCC is used in a variety of open-source projects, making it an important tool to consider.

State 0 is unusual in that it has a high percentage of observing SHL. SHL is the second most likely observed opcode (16% probability), but is not frequently observed in the other HMMs. There is also a low probability of staying in this state, combined with the highest probability of transitioning to the MOV state for any of our models, suggesting that state 0 is more transitional than the PUSH states of the other HMMs.

Our second compiler is the Clang compiler front end for LLVM, using version 2.0. Clang is a more recent tool than GCC, but it has also been used for a number of open-source

projects. The model for the Clang compiler is in the top right corner of Fig. 3.

State 7 is dominated by the observations of MOVSD, MOVZX, and MOVZX. Collectively, these operations are observed with a 94% probability. In contrast, these three operations are only observed with a combined 22% probability in state 3 of the GCC HMM, and do not occur with any great frequency in the other models. Transitions to state 7 are lower than equivalent transitions for the other HMMs. However, the probability of staying in this state is noticeably higher.

Another unusual characteristic of the Clang model is that SUB is a common observation in state 4, its PUSH state. For the other HMMs, SUB is usually a significant observation in the miscellaneous state.

The Minimalist GNU for Windows (MinGW) [23] is a port of GCC to Windows. We use version 4.6.1. We are particularly interested in MinGW since it allows to compare the models generated by the same compiler on two different platforms. The model for the MinGW compiler is in the bottom left corner of Fig. 3.

The most unusual aspect of the MinGW code is the use of PUSHF and POPF. These opcodes are never observed in the data for the other compilers; their presence alone strongly suggests that the code was compiled with MinGW. Another difference is the high probability of OR opcodes being used,

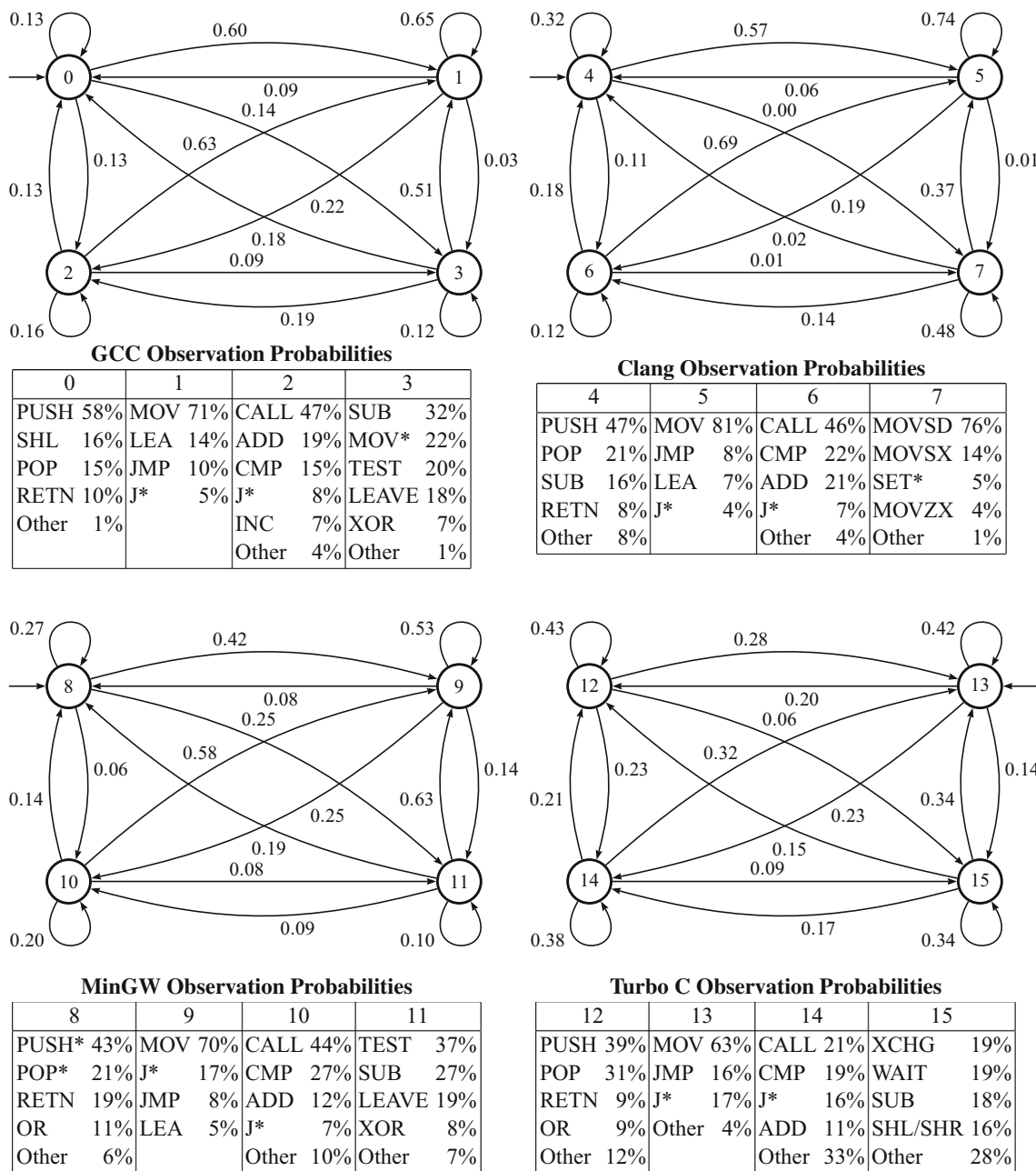


Fig. 3 HMMs for GCC, clang, MinGW, and Turbo C compilers from disassembled code

reflected in the high probability of that opcode in state 8. This quality is shared with the Turbo C compiler, perhaps indicating this feature is characteristic of Windows executables.

Borland's Turbo C [18], version 2.01, is popular for Windows. Additionally, Borland's Turbo Assembler (TASM) is

a common choice for hand-written assembly programs. We contrast the HMM for Turbo C with the Turbo ASM HMM in Sect. 6.

The model for the Turbo C compiler is in the bottom right corner of Fig. 3. The HMM for Turbo C is unusual in that

its MOV state, state 13, has 100% probability of being the initial state. In our data, a MOV operation is the first opcode in all programs.

The Turbo C compiler also seems to use a much greater variety of opcodes, reflected in the high observations of ‘other’ opcodes in the different states. Furthermore, state 15 includes XCHG and WAIT as two of its most likely opcodes, which did not appear at all in the disassembled code for the other compilers.

4.3 Identifying compiler

While the HMMs for each of the 4 compilers have a similar structure, they nonetheless can identify the compiler used with a high degree of accuracy. Our tests use additive smoothing [6] on the probabilities for each observation. No smoothing is applied to the transition probabilities or to the initial state probabilities. Probabilities are not normalized for length, since it is not necessary with the dueling HMM strategy.

Test data consists of 92 separate programs: 24 were compiled with GCC on OSX, 24 with CLANG on OSX, 21 with Turbo C on Microsoft Windows XP, and 23 with MinGW on Microsoft Windows XP. We use HMMs built with 2–11 states. More states get more accurate results, but with a significant performance penalty. When scoring (i.e., the forward algorithm) the work is on the order of $N^2 * T$ multiplications, where N is the number of states and T is the number of observations. Therefore, we would like to use as few states as possible.

There is only one false identification for HMMs with 2 or 3 states; there are no errors with additional states.

	Errors	Accuracy
2 Hidden states	1/92	0.99
3 Hidden states	1/92	0.99
4+ Hidden states	0/92	1.00

4.4 Alternate HMM construction

Previous research [30] has focused on the use of opcodes, but richer semantic information is available within the assembly code. On the other extreme, certain opcodes dominate in the model. Using less data might be as effective and more efficient.

Op codes are straightforward to use in analyzing assembly code, but they are not the only interesting source of information. Labels provide information about a program’s structure, and could potentially prove useful. For simplicity, we ignore the name of the label and treat the existence of a label as if it were another opcode. Unfortunately, considering labels does not improve the quality of our models, identifying the correct compiler with no greater probability.

	Errors	Accuracy
2 Hidden states	1/92	0.99
3 Hidden states	2/92	0.98
4–6 Hidden states	0/92	1.00

Identification of hand-written assembly and viruses is comparable as well, suggesting that considering labels is not beneficial.

For a different approach, we consider only the most frequently observed opcodes. By ignoring less common observations, our analysis can be more efficient.

With a 2-state HMM using only the MOV and CALL opcodes the correct model is chosen with 0.67 accuracy. The Turbo C code, however, is predicted with no more success than random guessing.

Compiler	Test files	Correctly identified	Accuracy
GCC	25	17	0.68
Clang	25	21	0.84
MinGW	23	20	0.87
Turbo C	21	5	0.24
All	94	63	0.67

Including more data improves the accuracy. We limit our observations to those opcodes that account for 20% or more of the observations for any state, improving the accuracy to more than 90%.

Compiler	Test files	Correctly identified	Accuracy
GCC	25	24	0.96
Clang	25	18	0.72
MinGW	23	23	1.00
Turbo C	21	21	1.00
All	94	86	0.91

5 Progression of states

An interesting aspect of HMMs lies in uncovering the hidden states to determine what fundamental properties they reveal of the thing being modeled. This section shows the break down of opcodes as the number of hidden states increases for the GCC HMM.

In all models discussed below, state 0 was the initial state with 100% probability. We ignore the transition probabilities; while this is important information, we focus on the opcodes used in order to gain a richer understanding of the semantics behind our HMMs.

With 2 states, CALL and MOV are broken into separate states as the most likely observations. The probabilities for different opcodes are shown below:

State Observation probabilities

- 0 : JNS(0.00) JNZ(0.03) JS(0.00) JZ(0.01) JMP(0.08)
- : LEA(0.11) **MOV(0.56)** POP(0.02) PUSH(0.08)
- : REP(0.00) RETN(0.01) SAR(0.00) SHL(0.02)
- : SHR(0.00) SUB(0.02) TEST(0.02) XOR(0.01)
- : LEAVE(0.01) CWDE(0.00) MOVSD(0.01)
- : MOVSX(0.00) MOVZX(0.01) SETNZ(0.00)
- : SETZ(0.00)
- 1 : ADD(0.19) AND(0.02) **CALL(0.47)** CLD(0.00)
- : CMP(0.15) DEC(0.02) IDIV(0.00) IMUL(0.00)
- : INC(0.07) JA(0.00) JB(0.00) JBE(0.00) JG(0.01)
- : JGE(0.01) JL(0.01) JLE(0.03)

With 3 states, a new state emerges with high probabilities for observing SUB, SHL, TEST, and LEAVE, though no one opcode seems to dominate. Interestingly, in most of our testing, models with 3 states often underperformed models with either 2 or 4 states. Section 4.4 shows one example.

State Observation probabilities

- 0 : SAR(0.02) SHL(0.22) SHR(0.00) SUB(0.24)
- : TEST(0.15) XOR(0.05) LEAVE(0.13)
- : CWDE(0.00) MOVSD(0.08) MOVSX(0.02)
- : MOVZX(0.07) SETNZ(0.00) SETZ(0.00)
- 1 : JNS(0.00) JNZ(0.03) JS(0.00) JZ(0.01) JMP(0.09)
- : LEA(0.12) **MOV(0.62)** POP(0.02) PUSH(0.09)
- : REP(0.00) RETN(0.01)
- 2 : ADD(0.19) AND(0.02) **CALL(0.47)** CLD(0.00)
- : CMP(0.15) DEC(0.02) IDIV(0.00) IMUL(0.00)
- : INC(0.07) JA(0.00) JB(0.00) JBE(0.00) JG(0.01)
- : JGE(0.01) JL(0.01) JLE(0.03)

With 4 states, the PUSH state emerges, separated from the MOV state. In our experiments, models with 4 states generally achieved marginally better accuracy than models with fewer states; beyond 4 states, the improvements in accuracy seem to be negligible.

State Observation probabilities

- 0 : POP(0.15) **PUSH(0.58)** REP(0.00) RETN(0.10)
- : SAR(0.02) SHL(0.16) SHR(0.00)
- 1 : JNS(0.00) JNZ(0.03) JS(0.00) JZ(0.02) JMP(0.10)
- : LEA(0.14) **MOV(0.71)**
- 2 : ADD(0.19) AND(0.02) **CALL(0.47)** CLD(0.00)
- : CMP(0.15) DEC(0.02) IDIV(0.00) IMUL(0.00)
- : INC(0.07) JA(0.00) JB(0.00) JBE(0.00) JG(0.01)
- : JGE(0.01) JL(0.01) JLE(0.03)
- 3 : **SUB(0.32)** TEST(0.20) XOR(0.07) LEAVE(0.18)
- : CWDE(0.00) MOVSD(0.11) MOVSX(0.02)
- : MOVZX(0.09) SETNZ(0.00) SETZ(0.00)

With 5 states, MOV separates from the jump instructions.

State Observation probabilities

- 0 : POP(0.15) **PUSH(0.58)** REP(0.00) RETN(0.10)
- : SAR(0.02) SHL(0.16) SHR(0.00)
- 1 : **MOV(1.00)**
- 2 : ADD(0.20) AND(0.02) **CALL(0.50)** CLD(0.00)
- : CMP(0.16) DEC(0.02) IDIV(0.00) IMUL(0.00)
- : INC(0.07) JA(0.00) JB(0.00) JBE(0.00)
- 3 : **SUB(0.32)** TEST(0.20) XOR(0.07) LEAVE(0.18)
- : CWDE(0.00) MOVSD(0.11) MOVSX(0.02)
- : MOVZX(0.09) SETNZ(0.00) SETZ(0.00)
- 4 : JG(0.01) JGE(0.01) JL(0.01) JLE(0.03)
- : JNS(0.00) JNZ(0.11) JS(0.00) JZ(0.05)
- : **JMP(0.34) LEA(0.44)**

Beyond this point, the instructions continue to break into separate states, though the meaning behind the different states seems less clear.

From our data, it appears that the two most significant operations are CALL and MOV. In all of the HMMs that we develop over the course of this paper, including the HMMs for hand-written assembly and virus code that we develop later, CALL and MOV observations are always in separate states. While we don't know why MOV and CALL are the predominant opcodes, we do have two theories. Most likely, the dominance of these opcodes is simply due to their frequency; even subtle differences in their usage might overwhelm the usage of other opcodes. Another intriguing possibility is that these codes may reflect the programming style used in the compiler; perhaps the usage of a more CALLs over MOVs suggests a more recursive, functional style rather than an imperative one, though this is pure speculation.

6 Hand-written assembly

We compare HMMs for compiled code with hand-written assembly code. While the models are noticeably different, our tool is unable to reliably distinguish between programs built from hand-written assembly and compiled code. We use Borland's Turbo Assembler (TASM) due to its use in the build processes for many of the viruses found on <http://vxheavens.com>, including the Next Generation Virus Konstruktion [sic] Kit (NGVCK) [25] and the Metamorphic Permutating High-Obfuscating Reassembler (MetaPHOR) [12]. Our model is built from 46 sample assembly programs taken from assembly programming tutorials.

Figure 4 shows the HMM for hand-written assembly code with 4 hidden states. The model is strikingly different from the HMMs for compiled code. While those HMMs always begin in the same state with 100% probability, the HMM for hand-written assembly has no single initial state. There is also a far greater variety of opcodes used in hand-written

Observation Probabilities							
	0	1	2	3			
J*	76%	MOV	29%	AND	49%	INSB	38%
INT	11%	PUSH/PUSHF	29%	CALL	26%	XOR	19%
IMUL	7%	JMP/J*	12%	CMP	14%	BOUND	14%
Other	5%	OR	9%	ADD	6%	SUB	12%
		Other	21%	Other	5%	Other	17%

Initial State Probability	
State Number	Probability
0	0%
1	87%
2	11%
3	6%

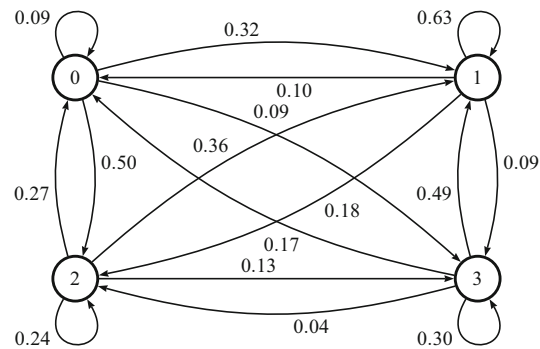


Fig. 4 HMM for hand-written assembly

assembly. While the division of the opcodes into different states follows some of the same patterns as the HMMs for the compilers, there are some notable differences.

The MOV state and the PUSH state are combined. A number of jump instructions instead have their own state (State 0). CALL and CMP opcodes are still in their own state, but there is also a high amount of AND instructions. State 3 roughly corresponds to the miscellaneous state of the compiler HMMs, but it includes a high number of INSB instructions.

Our test data includes 10 hand-written assembly programs along with the 92 compiled programs used in previous sections. The compiled C programs are successfully identified, even with as few as 2 states. The hand-written assembly programs are not identified as successfully as shown below:

	Errors	Accuracy
2 Hidden states	4	0.60
3 Hidden states	5	0.50
4 Hidden states	4	0.60
5–7 Hidden states	3	0.70

For the mistaken identifications, Turbo C was identified as the most likely model. With Turbo C removed, all of the hand-assembled programs were successfully identified. Given the striking difference in the HMM generated for hand-written assembly, the poor results are surprising. Perhaps some assembly programmers follow a similar pattern as compilers.

7 Identifying code generated with virus construction kits

Virus construction kits (VCKs) make it easy for anyone with minimal technical skills to create a virus, thus lowering virus creation from an art for the technical elite to a paint-by-the-numbers craft open to anyone with a malicious intent.

One of our central observations is that virus construction kits act as a specialized compiler. This section considers HMMs for identifying viruses created by VCKs. We use the Next Generation Virus Konstruktion [sic] Kit (NGVCK) for our tests due to its advanced techniques [29], performing additional tests with the Second Generation Virus Generator (G2) and the Mass Code Generator (MPC-GEN).

The Next Generation Virus Konstruktion [sic] Kit can create viruses that are automatically morphed, making it difficult to detect all variants with traditional techniques [29]. It uses several source-morphing techniques, including random function ordering, junk code insertion, and encryption [29].

The HMM for the NGVCK virus family is shown in Fig. 5. The model is built from 200 sample virus programs that have been compiled with Turbo C. It follows a pattern similar to the compiled programs, with a PUSH state, a MOV state, and a CALL state. Nonetheless, there are some noticeable distinctions. A striking difference is the high use of the NOP opcode in state 0, which hardly appeared in any of the other HMMs. Additionally, as with the Turbo ASM HMM, there is no single starting state.

We use five-fold cross validation to increase our sample size. The sample virus programs are divided into 5 equal groups; each slice is then tested against a model built from the other 4 slices. Our tests include 200 NGVCK-infected files divided up into groups of 40. Additionally, we include a set of benign files as follows:

- 74 files compiled by GCC
- 72 files compiled by Clang
- 72 files compiled by MinGW
- 64 files compiled by TurboC
- 56 hand-written Turbo Assembly files
- 16 Cygwin utilities
- 16 Linux utilities

Observation Probabilities				
0	1	2	3	
NOP 75%	MOV 68%	ADD 44%	SUB	38%
PUSH 12%	J* 20%	CALL 19%	XOR	14%
POP 6%	JMP 7%	CMP 12%	POPA	11%
Other 7%	Other 6%	Other 25%	PUSHA	9%
			Other	27%

Initial State Probability	
State Number	Probability
0	14%
1	1%
2	84%
3	1%

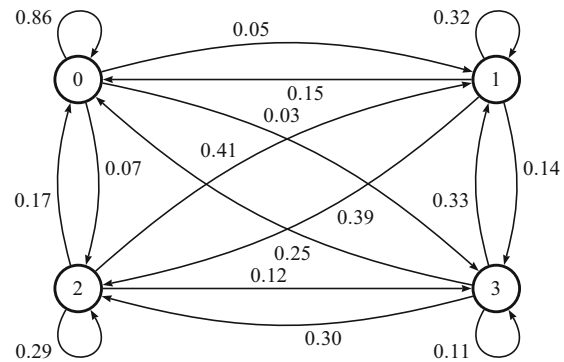


Fig. 5 HMM for NGVCK virus family

Table 4 Results for virus detection using threshold approach

Virus family	False positives	F.P. (%)	False negatives	F.N. (%)	Accuracy (%)	Total time (ms)	Threshold
G2	62/370	16.76	4/50	8.00	84.29	548.22	-2.66659
MPCGEN	89/370	24.05	0/50	0.00	78.81	495.47	-2.90398
NGVCK	166/370	44.86	16/200	8.00	68.07	550.97	-3.57961
MetaPHOR	198/370	53.51	10/60	16.67	34.88	538.67	-2.65207
MWOR (PR 1)	0/370	0.00	0/100	0.00	100.00	866.27	-2.6251
MWOR (PR 2)	1/370	0.27	0/100	0.00	99.79	964.16	-2.5394
MWOR (PR 3)	4/370	1.08	0/100	0.00	99.15	1152.99	-2.63274
MWOR (PR 4)	4/370	1.08	0/100	0.00	99.15	1341.37	-2.60028

All of the benign files were included in each test. These tests were performed on a machine running Windows 7, 64 bit operating System with an Intel Core i5-3210M CPU @ 2.5GHz processor and 8.00 GB RAM. The Linux system that was used was running Ubuntu 12.04. Ubuntu was run on the host machine using VMware player (version 5.0.2), with 1.00 GB RAM provided to the VM.

For additional validation of our approach, we also test 50 files infected with the G2 VCK and 50 files infected with the MPCGEN VCK, both divided into 5 groups of 10. Our tests were performed with 2–4 states.

Table 4 shows our results using the threshold approach, including the false positives, false negatives, and total execution time for the tests. The final column shows the threshold values used. The threshold approach detects many infected files, but suffers from a number of false positives, most notably in the case of NGVCK.

In contrast, the dueling HMM strategy fares much better. Table 5 shows that the results improve considerably in terms of the false negatives and the false positives.⁴ However, as a

side effect of each file being scored against multiple HMMs the files take significantly longer to be classified. We apply a bias to some malware models, shown in the final column of Table 5, in order to customize the false positive / false negative ratio. This technique is discussed more in Sect. 7.1.

7.1 Biasing the dueling HMM strategy

The original HICSS paper [3] did not include a technique for customizing the results to the desired false positive / false negative ratio; instead, the best match was always used to determine whether a file was benign or malicious. While this strategy achieves good results, it lacks the flexibility desired in a real-world system.

Biasing provides the necessary flexibility to fine tune the dueling HMM approach. Essentially, for a file to be classified as malicious, its score against the HMM representing the malware family must exceed the scores given by the HMMs representing benign executables by more than the bias.

We apply a biasing of -0.25 to the G2 HMM and a biasing of -0.45 to the MPCGEN HMM. Tables 6 and 7 shows how the false positives and false negatives vary with different values of biasing applied for the G2 HMM and the MPCGEN

⁴ While NGVCK remains difficult to detect, its false positive rate plummets.

Table 5 Results for virus detection Using dueling HMM approach

Virus family	False positives	F.P. (%)	False negatives	F.N. (%)	Accuracy (%)	Total time (ms)	Bias
G2	0/370	0.00	0/50	0.00	100.00	1,030.57	-0.25
MPCGEN	0/370	0.00	0/50	0.00	100.00	981.13	-0.45
NGVCK	2/370	0.54	75/200	37.50	86.49	1,155.61	0.00
MetaPHOR	4/370	1.08	9/60	15.00	96.98	1,113.47	0.00
MWOR (PR 1)	0/370	0.00	0/100	0.00	100.00	2,019.35	0.00
MWOR (PR 2)	0/370	0.00	0/100	0.00	100.00	2,413.05	0.00
MWOR (PR 3)	0/370	0.00	0/100	0.00	100.00	2,897.53	0.00
MWOR (PR 4)	0/370	0.00	0/100	0.00	100.00	3,433.05	0.00

Table 6 Biasing results for G2

Bias	False positive	F.P. (%)	False negative	F.N. (%)
0.00	2/370	0.54	0/50	0.00
-0.05	2/370	0.54	0/50	0.00
-0.10	2/370	0.54	0/50	0.00
-0.15	1/370	0.27	0/50	0.00
-0.20	1/370	0.27	0/50	0.00
-0.25	0/370	0.00	0/50	0.00

Table 7 Biasing results for MPCGEN

Bias	False positive	F.P. (%)	False negative	F.N. (%)
0.00	4/370	1.08	0/50	0.00
-0.05	4/370	1.08	0/50	0.00
-0.10	3/370	0.81	0/50	0.00
-0.15	3/370	0.81	0/50	0.00
-0.20	2/370	0.54	0/50	0.00
-0.25	2/370	0.54	0/50	0.00
-0.30	2/370	0.54	0/50	0.00
-0.35	1/370	0.27	0/50	0.00
-0.40	1/370	0.27	0/50	0.00
-0.45	0/370	0.00	0/50	0.00

HMM, respectively. ‘0.00’ represents the case where no biasing is present.

In Table 6, we vary the bias applied from ‘0’ to ‘-0.25’ in increments of 0.05 to see how the biasing changes the result. The false positives gradually decrease from 2 to 0 while the false negative rates stay constant at 0. Similarly, in Table 7 we see the results corresponding to the bias applied to MPCGEN. The bias in case of MPCGEN is varied from 0.0 to -0.45 in increments of 0.05. The number of false positives for MPCGEN decreases from 4 for no biasing to 0 with a biasing of ‘-0.45’.

In our experiments, biasing the other HMMs of malware families did not appear beneficial.

8 Metamorphic malware detection

Metamorphic viruses are difficult to detect with traditional scanning approaches. The virus code is obfuscated rather than merely encrypted. In this section, we examine files infected by the MetaPHOR virus [12], known for its sophisticated metamorphic techniques [29]. Additionally, we test our strategy against MWOR [22], a metamorphic worm specially designed to evade HMM-based detectors.

8.1 Detecting metaphor

The Win32/Simile virus, sometimes known as Win32/Etap, is one of the more advanced metamorphic viruses. It is built with the Metamorphic Permutating High-Obfuscating Reassembler (MetaPHOR) engine [12], created by a virus programmer known only as ‘the Mental Driller’. It infects 32-bit Windows files, though later versions also infect Linux ELF files [29]. While it does not include a destructive payload [28], it could be modified to do so. Roughly 90% of the virus code relates to its metamorphic behavior [29].

Our initial training data consists of 49 programs compiled with MinGW and infected by MetaPHOR. Figure 6 shows the model generated from the MetaPHOR-infected files. It has some noticeable similarities with the MinGW model. There is a CALL state in both models with a high probability of observing CMP and ADD opcodes. State 3 is dominated by observations of TEST, SUB, and XOR in both HMMs. Like the MinGW model, the metaphor model begins in state 0 with 100% probability. The main distinction between the two models is in the observations of jump instructions. The MinGW model has a distinct MOV state and PUSH/POP state, while the MetaPHOR model combines these two states and breaks out jump instructions into their own state. In this feature, it more closely resembles the HMM for hand-written assembly.

Our test data includes 60 programs compiled with MinGW and then infected by MetaPHOR, divided into groups of 12

Observation Probabilities

0		1		2		3	
MOV	65%	J*	59%	CALL	31%	TEST	31%
PUSH*	16%	JMP	23%	CMP	26%	SUB	27%
POP*	11%	LEA	18%	ADD	20%	XOR	19%
Other	8%			AND	14%	LEAVE	15%
				Other	9%	Other	8%

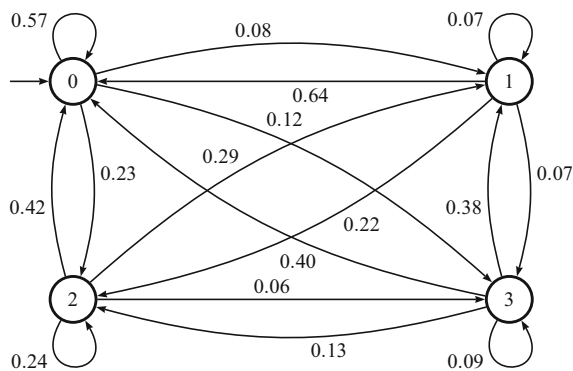
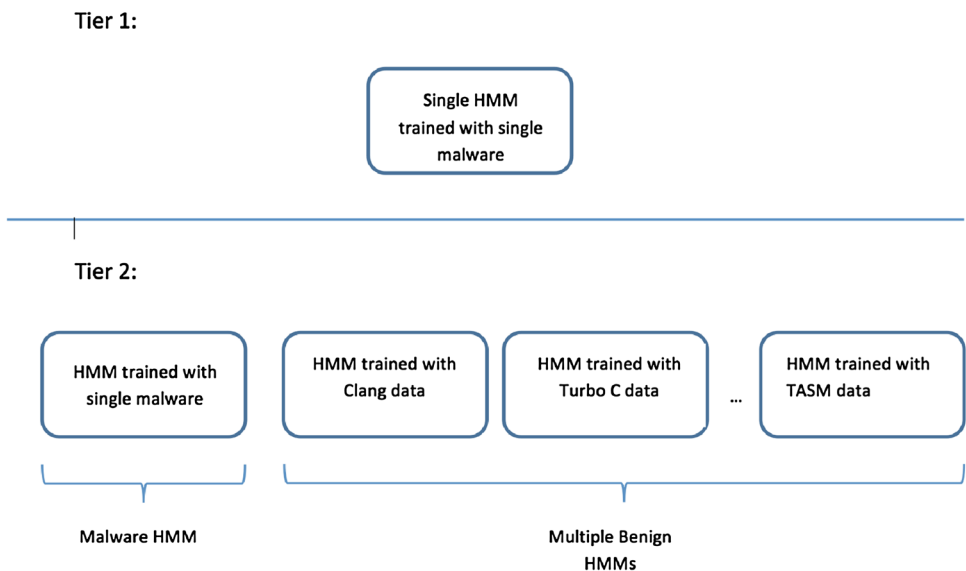


Fig. 6 HMM for MetaPHOR infected files

Fig. 7 Design of the tiered HMM approach



for use in fivefold cross validation. The set of benign files used in these tests is the same set used in Sect. 7.

Table 4 illustrates the challenge of detecting MetaPHOR-infected files; it has the highest false-negative and the highest false-positive rates of any malware that we tested. With a false positive rate exceeding 50%, the threshold approach is clearly not effective at detecting this virus family.

The results improve remarkably when tested with the Dueling HMM strategy as shown in Table 5. Only 4 out of 370 benign files are mistakenly identified as infected, and all but 9 out of 60 infected files are correctly identified. No bias was used in these tests, though it is possible to skew the results to achieve the desired balance between false negatives and false positives.

8.2 Identifying MWOR

MWOR [22] is an advanced metamorphic worm specifically designed to evade HMM-based detectors. It adds dead code

taken from Linux utilities in order to blend in with benign code. Madenur Sridhara and Stamp [22] show that MWOR is able to evade detection from threshold-based HMM detectors when the added dead code is more than 2.5 times the worm code.

In order to test the effectiveness of the dueling HMM strategy against MWOR, we use the same set of benign files used in Sect. 7. The Linux utilities in our testing data are particularly important, since MWOR is trying to masquerade as one of those files.

Our experiments test different padding ratios; that is, the amount of dead code included varies in different tests. Each test group includes 100 MWOR executables, evaluated with 5-fold cross validation. A padding ratio of 1 (PR 1) indicates a 1:1 ratio of the dead code to MWOR’s code; a padding ratio of 4 (PR 4) indicates a 4:1 ratio of the dead code to MWOR’s code;

Consistent with Madenur Sridhara and Stamp [22], the threshold approach begins to break down as the padding ratio increases, as shown in Table 4.

Table 8 Results for virus detection using tiered approach

Virus family	False positives	F.P. (%)	False negatives	F.N. (%)	Accuracy (%)	Threshold	Eliminated by Tier 1	Bias	Total time (ms)
G2	0/370	0.00	0/50	0.00	100.00	-2.67	249/420	-0.25	613.26
MPCGEN	0/370	0.00	0/50	0.00	100.00	-2.90	281/420	-0.45	603.36
NGVCK	2/370	0.54	75/200	37.50	86.49	-3.62	209/470	0.00	737.00
MetaPHOR	4/370	1.08	9/60	15.00	96.98	-2.79	152/430	0.00	860.57
MWOR (PR 1)	0/370	0.00	0/100	0.00	100.00	-2.63	370/470	0.00	1512.65
MWOR (PR 2)	0/370	0.00	0/100	0.00	100.00	-2.54	369/470	0.00	1968.83
MWOR (PR 3)	0/370	0.00	0/100	0.00	100.00	-2.61	366/470	0.00	2531.72
MWOR (PR 4)	0/370	0.00	0/100	0.00	100.00	-2.59	366/470	0.00	3187.04

Table 9 Comparison of detection strategies

Virus family	Threshold				Dueling				Tiered			
	False positives	False negatives	Accuracy	Total time (ms)	False positives	False negatives	Accuracy	Total time (ms)	False positives	False negatives	Accuracy	Total time (ms)
G2	62/370	4/50	84.29	548.22	0/370	0/50	100.00	1,030.57	0/370	0/50	100.00	613.26
MPCGEN	89/370	0/50	78.81	495.47	0/370	0/50	100.00	981.13	0/370	0/50	100.00	603.36
NGVCK	166/370	16/200	68.07	550.97	2/370	75/200	86.49	1,155.61	2/370	75/200	86.49	737.00
MetaPHOR	198/370	10/60	34.88	538.66	4/370	9/60	96.98	1,113.47	4/370	9/60	96.98	860.57
MWOR (PR 1)	0/370	0/100	100.00	866.27	0/370	0/100	100.00	2,019.35	0/370	0/100	100.00	1,512.65
MWOR (PR 2)	1/370	0/100	99.79	964.16	0/370	0/100	100.00	2,413.05	0/370	0/100	100.00	1,968.83
MWOR (PR 3)	4/370	0/100	99.15	1,152.99	0/370	0/100	100.00	2,897.53	0/370	0/100	100.00	2,531.72
MWOR (PR 4)	4/370	0/100	99.15	1,341.37	0/370	0/100	100.00	3,433.05	0/370	0/100	100.00	3,187.04

Table 9 shows that the dueling HMM strategy achieves superior results to the threshold approach. It achieves perfect accuracy, even with the highest padding ratio used (PR 4).

9 The tiered HMM strategy: improving the performance of the dueling HMM strategy

While our results show that the dueling HMM strategy achieves good results in detecting malware that uses advanced metamorphic techniques, it requires a significant amount of performance overhead compared to the threshold approach.

In this section, we explore how the dueling HMM strategy can be combined with the threshold approach. This *tiered HMM strategy* significantly reduces the performance overhead of the dueling HMM strategy while maintaining the effectiveness of the original.

In the tiered approach, a potential malware file has to go through multiple layers of trained HMMs. The goal is to use the threshold approach to help reduce the number of files that are actually passed on to the dueling layer, thus increasing efficiency. With the introduction of additional HMMs in the layered approach, the efficiency in terms of time decreases as the file needs to be tested against a greater number of HMM

models to classify it as either good or bad. The challenge is to reduce the time taken to classify the file; using the threshold model, we filter out the “definitely good” and the “definitely bad” files leaving a small number of files in the gray area that need to be evaluated with the dueling HMM approach.

The tiered approach is illustrated in Fig. 7. It consists of two distinct layers: the threshold tier and the dueling tier. The file to be classified is initially processed by the threshold HMM tier to see whether it can be classified with a minimal risk of false positives. If the file cannot be eliminated in the first tier, it is then sent to the dueling HMM tier for classification. The description of the tiers is given below.

The goal of tier 1 is to quickly eliminate a good chunk of viruses by the use of the threshold technique. The log likelihood per opcode is determined for each file and is compared against the threshold. The files that cannot be satisfactorily classified are passed on to the next tier. For this approach, the threshold for Tier 1 was chosen in such a way that there were no false negatives except in the case of MetaPHOR and NGVCK, and hence they differ from the thresholds chosen in the standalone threshold approach. The results of our tests are shown in Table 8.

The files which cannot be eliminated using the threshold approach are then scored using the dueling HMM approach in

the second tier. Tier 2 contains HMMs for each virus family as well as all benign code. Thus, each suspected file can now be inspected to see whether it belongs to a particular category. The file will belong to the category of the HMM which will give it the highest score.

Table 9 shows the comparison of the three approaches. The threshold approach is the fastest method, but is less effective at detecting malware than the other strategies. The dueling approach, as expected, is the slowest of the three; The execution time for the dueling approach is on average two to three times the execution time of the threshold approach. The tiered approach lies snugly in the middle with the execution time ranging from 1.1 to 2.5 times the time taken by the threshold approach, with most of the execution times for the tiered model approaching the execution time for the threshold model. Despite this speedup, Table 9 shows that the tiered approach is as effective as the dueling HMM strategy, making the tiered strategy a clear winner.

10 Conclusions

Hidden Markov models show promise as a tool for virus identification, particularly in identifying metamorphic viruses. We illustrate how multiple HMMs may be used in the dueling HMM strategy, improving the effectiveness of HMM-based detectors. We also show how the threshold-based approach and the dueling HMM strategy can be combined in a tiered fashion to minimize the performance penalty of the dueling HMM strategy. We also reveal some of the details about the hidden states of the HMM models, allowing for a richer understanding of the critical properties of the underlying models. By leveraging the information available in these models, we hope to improve our ability to detect metamorphic malware.

References

- Annachhatre, C., Austin, T.H., Stamp, M.: Hidden markov models for malware classification. *J. Comput. Virol. Hack. Tech.* pp. 1–15 (2014). doi:10.1007/s11416-014-0215-x
- Attaluri, S., McGhee, S., Stamp, M.: Profile hidden markov models and metamorphic virus detection. *J. Comput. Virol.* **5**, 151–169 (2009). doi:10.1007/s11416-008-0105-1
- Austin, T.H., Filiol, E., Josse, S., Stamp, M.: Exploring hidden markov models for virus analysis: a semantic approach. In: *IEEE HICSS*, pp. 5039–5048 (2013)
- Bruschi, D., Martignoni, L., Monga, M.: Detecting self-mutating malware using control-flow graph matching. In: *DIMVA* (2006)
- Cave, R.L., Neuwirth, L.P.: Hidden markov models for english. In: Ferguson, J.D. (ed) *Hidden Markov Models for Speech* (1980)
- Chen, S.F., Goodman, J.: An empirical study of smoothing techniques for language modeling. In: *Association for computational linguistics* (1996). doi:10.3115/981863.981904
- Chess, D.M., White, S.R.: An undetectable computer virus. In: *Virus bulletin conference* (2000)
- Cho, S.B., Han, S.J.: Two sophisticated techniques to improve hmm-based intrusion detection systems. In: *RAID* (2003)
- Christodorescu, M., Jha, S.: Testing malware detectors. In: *ISSTA* (2004)
- Christodorescu, M., Jha, S., Seshia, S.A., Song, D.X., Bryant, R.E.: Semantics-aware malware detection. In: *Symposium on security and privacy* (2005)
- Clang: a C language family frontend for LLVM. <http://www.clang-llvm.org>. Accessed November 2011
- Driller, T.M.: Metamorphic permutating high-obfuscating reassembler source. <http://vx.netlux.org/29a/29a-6/29a-6.602>. Accessed December 2011
- Filiol, E., Josse, S.: A statistical model for undecidable viral detection. *J. Comput. Virol.* **3**, 64–74 (2007). doi:10.1007/s11416-007-0041-5
- Filiol, E., Josse, S.: Malware spectral analysis: security evaluation of Bayesian network based detection models. In: *EICAR conference* (2011)
- Francois, J.M.: JAHMM: An implementation of hidden Markov models in Java. <http://code.google.com/p/jahmm/>. Accessed October 2011
- GCC, the GNU compiler collection. <http://gcc.gnu.org/>. Accessed November 2011
- Iliopoulos, D., Adami, C., Szor, P.: Darwin inside the machines: malware evolution and the consequences for computer security. *CoRR abs/1111.2503* (2011)
- Intersimone, D.: Antique software: Turbo C version 2.01. <http://edn.embarcadero.com/article/20841>. Accessed November 2011
- Krügel, C., Kirda, E., Mutz, D., Robertson, W.K., Vigna, G.: Polymorphic worm detection using structural information of executables. In: *RAID* (2005)
- Leder, F., Steinbock, B., Martini, P.: Classification and detection of metamorphic malware using value set analysis. In: *International conference on malicious and unwanted software MALWARE* (2009)
- Lin, D., Stamp, M.: Hunting for undetectable metamorphic viruses. *J. Comput. Virol.* **7**(3), 201–214 (2011)
- Madenur Sridhara, S., Stamp, M.: Metamorphic worm that carries its own morphing engine. *J. Comput. Virol.* **9**(2), 49–58 (2013). doi:10.1007/s11416-012-0174-z
- MinGW | the minimalist GNU for Windows. <http://www.mingw.org/>. Accessed November 2011
- Mohammed, M.: Zeroing in on metaphoric computer viruses. Master's thesis, University of Louisiana at Lafayette (2003)
- SnakeByte: next generation virus konstruktion kit. <http://vxheavens.com/vx.php?id=tn02>. Accessed December 2011
- Song, Y., Locasto, M.E., Stavrou, A., Keromytis, A.D., Stolfo, S.J.: On the infeasibility of modeling polymorphic shellcode—rethinking the role of learning in intrusion detection systems. *Mach. Learn.* **81**(2), 179–205 (2010)
- Stamp, M.: A revealing introduction to hidden Markov models (2004). <http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf>. Accessed October 2011
- Symantec security response: W32.simile. http://www.symantec.com/security_response/writeup.jsp?docid=2002-030617-5423-99. Accessed December 2011
- Szor, P.: *The Art of Computer Virus Research and Defense*. Addison Wesley, Boston (2005)
- Wong, W., Stamp, M.: Hunting for metamorphic engines. *J. Comput. Virol.* **2**(3), 211–229 (2006)
- Zhang, Q., Reeves, D.S.: Metaaware: identifying metamorphic malware. In: *ACSAC* (2007)