Exploring Hidden Markov Models for Virus Analysis: A Semantic Approach

Thomas H. Austin
UC Santa Cruz
Santa Cruz, California 95064
Email: taustin@ucsc.edu

Eric Filiol and Sébastien Josse
ESIEA Laboratoire $(C + V)^O$
Laval, France
Email: {filiol,josse}@esiea.fr

Mark Stamp
San José State University
San Jose, California 95192
Email: stamp@cs.sjsu.edu

Abstract

Recent work has presented hidden Markov models (HMMs) as a compelling option for virus identification. However, to date little research has been done to identify the meaning of these hidden states. In this paper, we examine HMMs for four different compilers, hand-written assembly code, three virus construction kits, and a metamorphic virus in order to note similarities and differences in the hidden states of the HMMs. Furthermore, we develop the dueling HMM Strategy*, which leverages our knowledge about different compilers for more precise identification. We hope that this approach will allow for the development of better virus detection tools based on HMMs.*

1. Introduction

Wong and Stamp [24] 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.

In other domains, the states of an HMM have been connected with some fundamental aspects of the problem at hand. Cave and Neuwirth [3] 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 a metamorphic virus. 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 [24]. 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 [6], this approach has not previously been applied to virus identification.

Signature-based detection is the primary method of identifying computer viruses [23]. However, virus makers have been resourceful, and have developed a variety of counter-measures. 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 [23]. 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 build on the 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 [14] still remain theoretical, but might further complicate virus detection.

Research shows that better virus detection tools are needed to handle these threats. Christodorescu and Jha [7] test different malware detectors and show that many commercial products are ill-equipped to handle code obfuscation techniques. Kruegel et

1530-1605/12 $26.00 © 2012 IEEE
DOI 10.1109/HICSS.2013.217

al. [17] use control-flow graphs to detect polymorphic/metamorphic worms. Bruschi et al. [2] use this technique to to normalize programs and compare the results, testing their technique against the MetaPHOR virus. Mohammed [21] 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. [18] use *value set analysis*, performing a static flow analysis and check for values that are characteristic of a piece of malware. Zhang and Reeves [25] statically analyze programs to compare semantics based on the pattern of library calls. Christodorescu et al. [8] consider the semantics of programs in order to identify polymorphic/metamorphic malware.

Hidden Markov models use a statistical approach to identify these viruses. Wong and Stamp [24] use HMMs to identify viruses from different virus construction kits (VCKs) with a high degree of accuracy. Attaluri et al [1] 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. Josse and Filiol [16] discuss the application of Bayesian techniques to detecting metamorphic viruses, considering both naive Bayes and HMMs.

Chess and White [5] show that there are computer viruses that no algorithm can detect. Song et al. [22] highlight the challenges that polymorphic techniques present to signature-based approaches and any generative approaches to producing malicious code. Filiol and Josse [11] discuss *s*tatistical testing simulability and show how attackers can evade detection by exploiting the defender's detection model. Lin [19] 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.

2. Dueling HMM Strategy

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

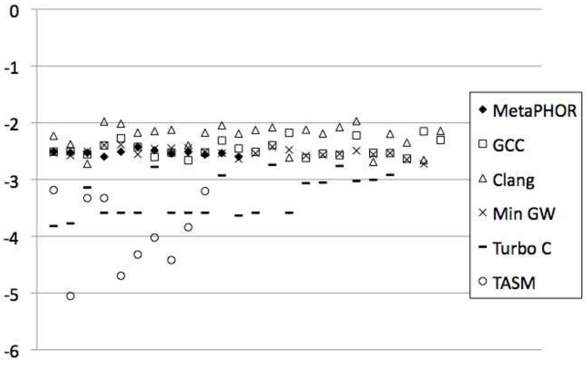
There are several benefits to this approach. Since only a single HMM is required, the analysis can be performed more efficiently. Also, it is straightforward to adjust the threshold value in order to set the desired balance between false positives and false negatives.

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 Section 7. The diagram below shows the distribution of 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 Section 3. 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. For instance, if the threshold value is set to -2.55, 75% (9/12) infected files are correctly identified. However, 57% (53/92) benign files are mistakenly flagged as viruses.

The dueling HMM strategy includes five additional HMMs, representing benign code compiled with the four different compilers and hand-written assembly code built with Turbo Assembler. The dueling HMM strategy correctly identifies 83% (10/12) of the infected files, without a single false positive.

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 could provide this flexibility.

3. 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 [13], the Clang [9] front-end for the LLVM project, the MinGW port of GCC to Windows, and the Turbo C compiler. We use the JAHMM toolkit [12] and code from http://www.c.happycodings.com to both train and test our models.

3.1. 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 OS X laptop running version 10.6.8.

Following Wong and Stamp [24], we consider only the x86 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 1 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. 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 antivirus 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.

3.2. Using disassembled assembly

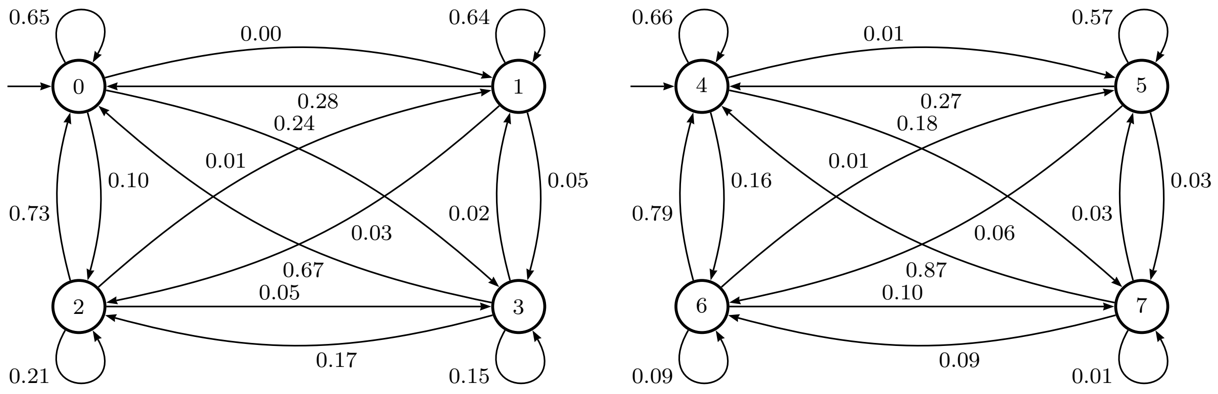
As Wong and Stamp observe [24], a more realistic model for generating assembly in antivirus detection compiles the source code and then disassembles the resulting binaries. For these tests, we used IDA Pro version 6.2.111006 as our disassembler.

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 Figure 2.

Four states seems to be the optimal number of states, determined by our testing in Section 3.3. 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 that state: the PUSH state; the MOV state; the CALL state; and the miscellaneous state.

Figure 1: GCC HMM and Clang HMM with 4 Hidden States from Compiler Generated Assembly



GCC Observation Probabilities

0		1		2		3	
MOVQ	47%	MOVSD	95%	JMP	42%	CALL	79%
MOVL	46%	SET*	5%	J*	24%	J*	21%
Other	7%			RET	16%		
				LEAVE	16%		
				Other	2%		

Clang Observation Probabilities

4		5		6		7	
MOVQ	47%	MOVSD	95%	J*	47%	CALLQ	100%
MOVL	46%	SET*	5%	JMP	40%	Other	< 1%
Other	7%			RET	13%		

SET*: SETE, SETNE, SETL, SETLE
J*: JA, JE, JG, JGE, JL, JLE, JNE, JAE, JB, JBE, JP, JS, and JNS.
Note: Due to rounding, probabilities do not always add up to 100%

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 Figure 2. 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 Figure 2.

State 7 is dominated by the observations MOVSD, MOVSX, 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) [20] 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 Figure 2.

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, 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 [15], 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 TASM HMM in Section 5.

The model for the Turbo C compiler is in the bottom right corner of Figure 2. 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.

3.3. 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 [4] 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 Windows XP, and 23 with MinGW on Windows XP. We use HMMs built with 2 to 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.

The 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. 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.

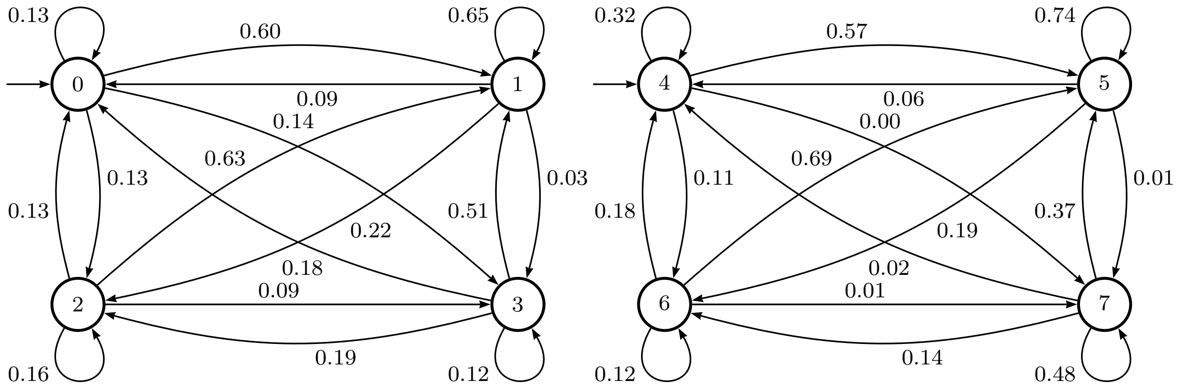
State		Observation probabilities
0	:	SAR(0.02) SHL(0.22) SHR(0.00) SUB(0.24)
	:	TEST(0.15) XOR(0.05) LEAVE(0.13)
	:	CWDE(0.00) MOVSD(0.08) MOVSX(0.02)
	:	MOVZX(0.07) SETNZ(0.00) SETZ(0.00)
1	:	JNS(0.00) JNZ(0.03) JS(0.00) JZ(0.01) JMP(0.09)
	:	LEA(0.12) **MOV(0.62)** POP(0.02) PUSH(0.09)
	:	REP(0.00) 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.

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)

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.

Figure 2: HMMs for GCC, Clang, MinGW, and Turbo C Compilers from Disassembled Code

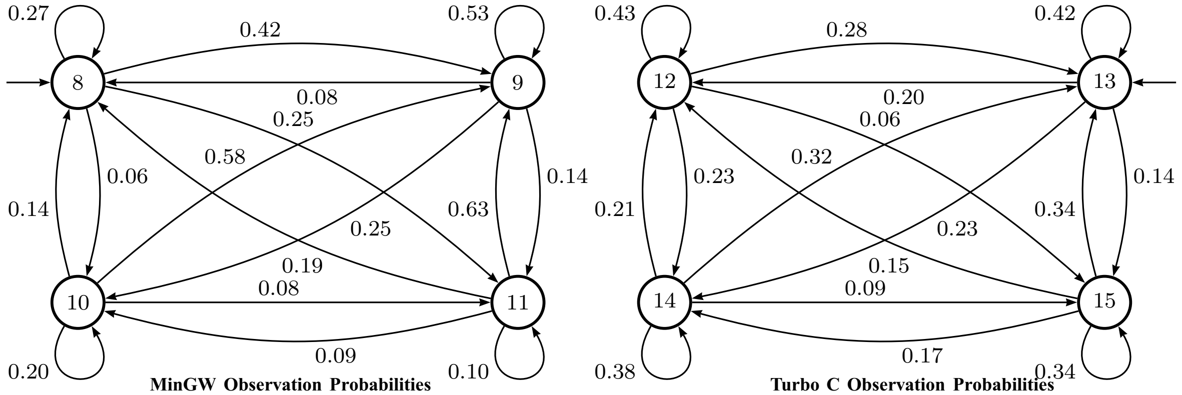


GCC Observation Probabilities

0		1		2		3	
PUSH	58%	MOV	71%	CALL	47%	SUB	32%
SHL	16%	LEA	14%	ADD	19%	MOV*	22%
POP	15%	JMP	10%	CMP	15%	TEST	20%
RETN	10%	J*	5%	J*	8%	LEAVE	18%
Other	1%			INC	7%	XOR	7%
				Other	4%	Other	1%

Clang Observation Probabilities

4		5		6		7	
PUSH	47%	MOV	81%	CALL	46%	MOVSD	76%
POP	21%	JMP	8%	CMP	22%	MOVSX	14%
SUB	16%	LEA	7%	ADD	21%	SET*	5%
RETN	8%	J*	4%	J*	7%	MOVZX	4%
Other	8%			Other	4%	Other	1%



MinGW Observation Probabilities

8		9		10		11	
PUSH*	43%	MOV	70%	CALL	44%	TEST	37%
POP*	21%	J*	17%	CMP	27%	SUB	27%
RETN	19%	JMP	8%	ADD	12%	LEAVE	19%
OR	11%	LEA	5%	J*	7%	XOR	8%
Other	6%			Other	10%	Other	7%

Turbo C Observation Probabilities

12		13		14		15	
PUSH	39%	MOV	63%	CALL	21%	XCHG	19%
POP	31%	JMP	16%	CMP	19%	WAIT	19%
RETN	9%	J*	17%	J*	16%	SUB	18%
OR	9%	Other	4%	ADD	11%	SHL/SHR	16%
Other	12%			Other	33%	Other	28%

SET*: SETE, SETNE, SETL, SETLE, SETZ, and SETNZ
J*: JA, JE, JG, JGE, JL, JLE, JNE, JAE, JB, JBE, JP, JS, JNS, JZ, and JNZ.
MOV*: MOVSD, MOVSX, and MOVZX.
PUSH*/POP*: PUSH/POP and PUSHF/POPF.
Note: Due to rounding, probabilities do not always add up to 100%

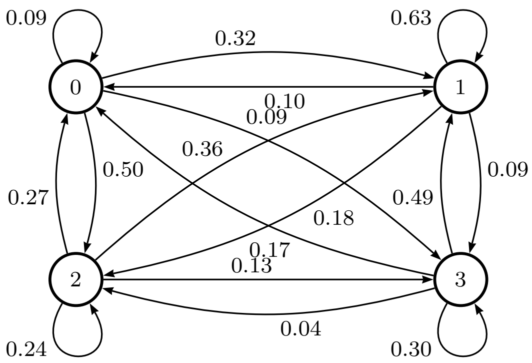
Figure 3: HMM for Hand-Written Assembly

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%



5. Hand Written Assembly

We now 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 Construktion Kit (NGVCK) and the Metamorphic Permutating High-Obfuscating Reassembler (MetaPHOR) virus [10]. Our model is built from 46 sample assembly programs taken from assembly programming tutorials.

Figure 3 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 begins 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 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

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.

6. 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. We use the Next Generation Virus Construktion Kit (NGVCK) for our tests due to its advanced techniques [23], performing additional tests with the Second Generation Virus Generator (G2) and the Mass Code Generator (MPCGEN).

The Next Generation Virus Construktion Kit creates viruses that are automatically morphed, making it difficult to detect all variants with traditional techniques [23]. It uses several source-morphing techniques, including random function ordering, junk code insertion, and encryption [23]. The HMM for the NGVCK virus family is shown in Figure 4. 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

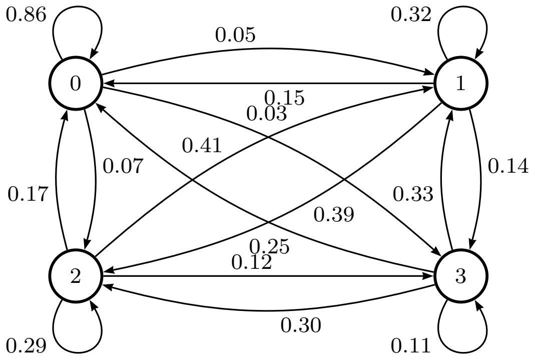
Figure 4: HMM for NGVCK virus family

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%



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 TASM HMM, there is no single starting state.

We use 5-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 225 NGVCK-infected files divided up into groups of 45, with 92 benign compiled programs and 10 hand-written assembly programs included in each test. 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. In contrast to the results in identifying compilers (Section 3.3), 2-state models identify nearly as many models, and suffer from fewer false-positives.

The results for the dueling HMM strategy used with 2-state HMMs to identify NGVCK-infected files are shown below. No benign compiled programs are flagged as infected, but some of the hand-coded assembly programs are mistakenly identified as viruses. Tests with 3-state HMMs identified fewer viruses; 4-state HMMs identified slightly more, but resulted in noticeably more false positives.

Group	Viruses identified	False positives
$NGVCK-1$	41/45	0/102
$NGVCK-2$	43/45	2/102
$NGVCK-3$	42/45	0/102
$NGVCK-4$	40/45	0/102
$NGVCK-5$	29/45	0/102
$Total$	195/225 (0.87)	2/510 (< 0.01)

The following results for G2 with 2-state HMMs show that the dueling HMM approach successfully identifies every virus with no false positives.

Group	Viruses identified	False positives
$G2-1$	10/10	0/102
$G2-2$	10/10	0/102
$G2-3$	10/10	0/102
$G2-4$	10/10	0/102
$G2-5$	10/10	0/102
$Total$	50/50 (1.00)	0/510 (0.00)

The story for MPCGEN-infected files is similar; When using 2-state HMMs, all infected files are identified; 5 benign hand-written assembly files are identified as viruses. Since both G2 and MPCGEN are somewhat less sophisticated than NGVCK [24], the strong performance of the dueling HMM model strategy is perhaps not surprising.

Group	Viruses identified	False positives
$MPCGEN-1$	10/10	1/102
$MPCGEN-2$	10/10	1/102
$MPCGEN-3$	10/10	1/102
$MPCGEN-4$	10/10	1/102
$MPCGEN-5$	10/10	1/102
$Total$	50/50 (1.00)	5/510 (0.01)

7. Metamorphic Virus Detection

Metamorphic viruses are difficult to detect with traditional scanning approaches. The virus code is obfuscated rather than merely encrypted. 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 [10]. Roughly 90% of the virus code relates to its metamorphic behavior [23].

Our initial training data consists of 49 programs compiled with MinGW and infected by MetaPHOR.

Figure 5: HMM for MetaPHOR Infected Files

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%

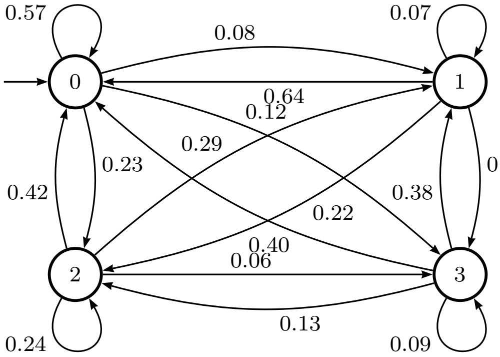


Figure 5 shows the model generated from the MetaPHOR-infected files. It has some noticeable similarities with the MinGW model. There is a CALL state in both models with a high probability of observing CMP and ADD opcodes. State 4 is dominated by observations of TEST, SUB, and XOR in both HMMs. Like the 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 divided into groups of 12 for use in 5-fold cross validation. 2-state HMMs identify 87% of the MetaPHOR-infected programs with no false positives. Additional states do not identify additional infected programs, and mark some benign MinGW files as viruses, suggesting that 2-state HMMs are ideal for virus identification.

Group	Viruses identified	False positives
$MetaPHOR - 1$	11/12	0/102
$MetaPHOR - 2$	10/12	0/102
$MetaPHOR - 3$	9/12	0/102
$MetaPHOR - 4$	11/12	0/102
$MetaPHOR - 5$	11/12	0/102
$Total$	52/60 (0.87)	0/510 (0.00)

8. Alternate HMM Construction

Previous research [24] 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.

Labels provide information about a program's structure. We treat the existence of a label as if it were another op code. Unforunately, 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
Total	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
Total	94	86	0.91

Unfortunately, identifying hand-written assembly and viruses is less successful. Of the 10 programs in hand-written assembly, only two are correctly identified. None of the NGVCK programs are correctly identified, and one of the hand-written assembly programs is flagged as an NGVCK-infected file. In one test of 12 MetaPHOR-infected programs, 10 are correctly identified but the number of false positives increases. Viruses and hand-written assembly use more opcodes than compiled programs; the additional opcodes appear to be important for accurate identification.

9. Conclusions and Future Work

Hidden Markov models show promise as a tool for virus identification, particularly in identifying metamorphic viruses. In this paper, we 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. Furthermore, we develop the dueling HMM strategy, leveraging specifics about compiled code for more precise analysis. In future work, we will explore how we may bias the dueling HMM strategy in order to fine-tune the tradeoff between false-positives and false-negatives.

References

[1] Srilatha Attaluri, Scott McGhee, and Mark Stamp. Profile hidden markov models and metamorphic virus detection. *Journal in Computer Virology*, 5:151–169, 2009. 10.1007/s11416-008-0105-1.

[2] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *DIMVA*, 2006.

[3] R. L. Cave and L. P. Neuwirth. Hidden markov models for english. In J. D. Ferguson, editor, *Hidden Markov Models for Speech*. October 1980.

[4] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Association for Computational Linguistics*, 1996.

[5] David M. Chess and Steve R. White. An undetectable computer virus. In *Virus Bulletin Conference*, 2000.

[6] Sung-Bae Cho and Sang-Jun Han. Two sophisticated techniques to improve hmm-based intrusion detection systems. In *RAID*, 2003.

[7] Mihai Christodorescu and Somesh Jha. Testing malware detectors. In *ISSTA*, 2004.

[8] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Xiaodong Song, and Randal E. Bryant. Semantics-aware malware detection. In *Symposium on Security and Privacy*, 2005.

[9] Clang: a C language family frontend for LLVM. http://clang.llvm.org, accessed November 2011.

[10] The Mental Driller. Metamorphic permutating high-obfuscating reassembler source. http://vx.netlux.org/29a/29a-6/29a-6.602, accessed December 2011.

[11] Eric Filiol and Sbastien Josse. A statistical model for undecidable viral detection. *Journal in Computer Virology*, 3:65–74, 2007. 10.1007/s11416-007-0041-5.

[12] Jean-Marc Francois. JAHMM: An implementation of hidden Markov models in Java. http://code.google.com/p/jahmm/, accessed October 2011.

[13] GCC, the GNU compiler collection. http://gcc.gnu.org/, accessed November 2011.

[14] D. Iliopoulos, C. Adami, and Peter Szor. Darwin inside the machines: Malware evolution and the consequences for computer security. *CoRR*, abs/1111.2503, 2011.

[15] David Intersimone. Antique software: Turbo C version 2.01. http://edn.embarcadero.com/article/20841, accessed November 2011.

[16] Sébastien Josse and Eric Filiol. Malware spectral analysis: security evaluation of Bayesian network based detection models. In *EICAR Conference*, 2011.

[17] Christopher Krügel, Engin Kirda, Darren Mutz, William K. Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, 2005.

[18] Felix Leder, Bastian Steinbock, and Peter Martini. Classification and detection of metamorphic malware using value set analysis. In *International Conference on Malicious and Unwanted Software MALWARE*, 2009.

[19] Da Lin and Mark Stamp. Hunting for undetectable metamorphic viruses. *Journal in Computer Virology*, 7(3):201–214, 2011.

[20] MinGW — the minimalist GNU for Windows. http://www.mingw.org/, accessed November 2011.

[21] Moinuddin Mohammed. Zeroing in on metaphoric computer viruses. Master's thesis, University of Louisiana at Lafayette, 2003.

[22] Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, and Salvatore J. Stolfo. On the infeasibility of modeling polymorphic shellcode - re-thinking the role of learning in intrusion detection systems. *Machine Learning*, 81(2):179–205, 2010.

[23] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005.

[24] Wing Wong and Mark Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, 2006.

[25] Qinghua Zhang and Douglas S. Reeves. Metaaware: Identifying metamorphic malware. In *ACSAC*, 2007.