Hidden Markov Models for Software Piracy Detection

Shabana Kazi  
Department of Computer Science  
San Jose State University  
San Jose, California

Mark Stamp*  
Department of Computer Science  
San Jose State University  
San Jose, California

Abstract

In this paper, we analyze a method for detecting software piracy. A metamorphic generator is used to create morphed copies of a base piece of software. A hidden Markov model is trained on the opcode sequences extracted from these morphed copies and the resulting trained model is used to score suspect software to determine its similarity to the base software. A high score indicates that the suspect software may be a modified version of the base software, suggesting that further investigation is warranted. In contrast, a low score indicates that the suspect software differs significantly from the base software. We show that our approach is robust, in the sense that the base software must be extensively modified before it is not detected.

Keywords: Hidden Markov models, piracy, software, metamorphic, malware

1 Introduction

Unauthorized use of software is known as software piracy. The illegal use of copyrighted software is one common example of piracy [36, 37].

According to a recent study, the commercial value of software piracy grew 14% globally in 2010 to a record high of $58.8 billion, and this amount almost doubled since 2003 [9]. For every dollar of PC software sold, it is estimated that $3 to $4 is lost due to piracy [29].

Pirated software also has an impact on security. Patches and fixes that are released to counter malware-related issues are generally unavailable to unlicensed, pirated products [29]. Obviously such vulnerabilities are a problem for those using pirated software, but they also expose legitimate users to more threats and attack vectors, since attacks are often launched from compromised machines.

*Email: stamp@cs.sjsu.edu
The goal of this project is to develop and test a technique that can be used as an aid to detect pirated software. Our approach is designed for the case where it is suspected that copyrighted software has been illegally copied and modified. Such software is often the focus of litigation [34, 21, 28]. The technique discussed in this paper could, for example, be applied before resorting to the costly abstraction-filtration-comparison (AFC) analysis [16], which is the accepted legal standard for software copyright infringement cases in the United States [24].

In this paper, we propose and analyze a novel technique based on hidden Markov models, where the original software is scored against a suspected pirated copy. A high score indicates that further investigation is warranted, while a low score indicates that the two pieces of software are substantially different. With our approach, scores can be computed after the original software has been distributed, and no special effort is required during the software development process. Our scoring technique uses executable files only, so that no source code is required. In addition, we rely only on statistical analysis—neither the original nor the suspect code is executed. The scoring technique is also fast, efficient, and applicable in general. Extensive experimental results provided in this paper indicate that our approach is robust, in the sense that the original software must be highly modified before we are unable to detect a significant degree of similarity to the original code.

Note that we are proposing a piracy detection technique, not a piracy prevention technique [8]. That is, our goal here is to detect piracy (or, more precisely, software copyright infringement) after the fact, not prevent it from occurring. Although it serves a similar purpose, our approach would generally not be considered a watermarking scheme [5], since no “mark” is embedded into the code. It is also worth noting that in spite of some superficial similarities, our approach is not closely related to—nor likely to be particularly useful for—software plagiarism detection [7, 18, 27].

Our technique was inspired by previous research focused on the detection of metamorphic malware. As a means of evading signature-based detection, metamorphic malware changes its internal structure, while maintaining its functionality. In [38], hidden Markov models (HMMs) are shown to be an effective means of detecting hacker-produced metamorphic viruses. The work in [38] has been extended and further analyzed in [1, 17, 30]. Consequently, the relative strengths and weaknesses of this technique—as applied to metamorphic malware detection—are currently well understood. Some preliminary work towards adapting these technique to the software piracy problem can be found in [20].

This paper is organized as follows. Section 2 discusses background material on metamorphic software and hidden Markov models. In Section 3, we provide an overview of the design of our piracy detection technique. Experimental results are given Section 4, while Section 5 summarizes our work and provides suggestions for future work.
2 Background

2.1 Metamorphic Software

Software metamorphism refers to producing copies of a piece of code that are functionally equivalent, but differ structurally. This method has been used by malware writers to create viruses that are undetectable using standard signature-based anti-virus techniques [2].

Metamorphism has the potential for positive applications as well. In an analogy to biological systems, metamorphism can be used to increase the “genetic diversity” of software [33]. One benefit of such diversity is that the same attack may not work on different metamorphic copies. For example, metamorphism has been shown to be a highly effective defense against buffer overflow attacks [11].

In this paper, we use metamorphic techniques to simulate modifications an attacker might make to code in an attempt to evade piracy detection. We also use such techniques as an aid to training our detector.

2.1.1 Metamorphic Techniques

This section covers some elementary techniques for generating metamorphic code. We consider only transformations that operate at the assembly code level and we only discuss a few elementary techniques that are most relevant for our purposes. More advanced morphing techniques are discussed in [6].

Dead Code: Perhaps the simplest method of morphing code is to insert “dead code” or “garbage instructions” [25], that is, we insert extraneous opcode sequences that are not executed by the program [15]. Since the inserted code is never executed, there is no semantic effect on the software [4]. An example of dead code insertion appears in Table 1.

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Transformed Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, 1034h</td>
<td>mov eax, 1034h</td>
</tr>
<tr>
<td>sub eax</td>
<td>jmp loc</td>
</tr>
<tr>
<td></td>
<td>push ebp</td>
</tr>
<tr>
<td></td>
<td>pop ebp</td>
</tr>
<tr>
<td></td>
<td>sub esp, 18h</td>
</tr>
<tr>
<td>loc:</td>
<td>sub eax, 1</td>
</tr>
</tbody>
</table>

Code Permutation: Permutation is another obvious technique for morphing code. For example, we can divide the code into frames, then position the frames in a random
order, and maintain the proper control flow by use of branch instructions [10].

It is not difficult to prove that permutation alone is sufficient for metamorphic malware to evade standard signature detection [3]. However, permutation techniques have minimal effect on the HMM-based detection strategies that are the basis for the work considered here [35].

**Instruction Replacement:** We can generally replace a particular instruction or set of instructions with an equivalent instruction or a set of instructions. Table 2 gives a simple example of code substitution.

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Transformed Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add eax, 05H</code></td>
<td><code>add eax, 04H</code></td>
</tr>
<tr>
<td><code>mov al, bl</code></td>
<td><code>add eax, 01H</code></td>
</tr>
<tr>
<td></td>
<td><code>push al</code></td>
</tr>
<tr>
<td></td>
<td><code>pop bl</code></td>
</tr>
</tbody>
</table>

Previous work has shown that instruction substitution is far less effective at defeating HMM-based malware detection than dead code insertion [17], provided the dead is properly selected (as discussed here in Section 4). There are also more limitations on instruction substitution and it is more challenging to implement, as compared to dead code insertion. Consequently, in this paper we focus on dead code insertion, since previous work indicates that this will present the most challenging case for our piracy detection technique.

### 2.2 Hidden Markov Models

A Markov process is a type of statistical model that has states and known transition probabilities [31]. In a Markov process, the states are visible to the observer. In contrast, for a hidden Markov model (HMM) the states are not directly observable [23]. HMMs have been successfully applied to a wide variety of problems, including speech recognition [26] and malware detection [19].

A hidden Markov model is defined by a matrix of state transition probabilities, a probability distribution for each output symbol in each state, and initial state
probabilities. We use following notation to describe an HMM [31]:

\[
\begin{align*}
T &= \text{length of the observation sequence} \\
N &= \text{number of states in the model} \\
M &= \text{number of observation symbols} \\
Q &= \{q_0, q_1, \ldots, q_{N-1}\} = \text{distinct states of the Markov process} \\
V &= \{0, 1, \ldots, M-1\} = \text{set of possible observations} \\
A &= \text{state transition probabilities} \\
B &= \text{observation probability matrix} \\
\pi &= \text{initial state distribution} \\
O &= (O_0, O_1, \ldots, O_{T-1}) = \text{observation sequence}
\end{align*}
\]

A hidden Markov model is defined by the matrices \(A\), \(B\) and \(\pi\) and, therefore, we denote an HMM as \(\lambda = (A, B, \pi)\). Figure 1 illustrates the components of a hidden Markov model, where the region above the dashed line is the “hidden” part.

The strength and utility of HMMs derives largely from the fact that there are efficient algorithms for solving each of the following problems [31].

**Problem 1**: Given a model \(\lambda = (A, B, \pi)\) and an observation sequence \(O\), find \(P(O|\lambda)\). That is, we can score an observation sequence to see how well it fits a given model.

**Problem 2**: Given \(\lambda = (A, B, \pi)\) and \(O\), determine an optimal state sequence for the underlying Markov process. That is, we can uncover the (most likely) hidden state sequence.

**Problem 3**: Given \(O\), \(N\), and \(M\), we can find a model \(\lambda = (A, B, \pi)\) that maximizes probability of \(O\). That is, we can train a model to “best” fit an observation sequence.

For the work presented in this paper, we employ the algorithms for Problems 1 and 3. First, we train a model from a given base piece of software, using extracted opcode sequences as the observations (Problem 3). Then the resulting model is used to score a suspect piece of code, again using extracted opcode sequences (Problem 1). A high score indicates a high degree of similarity with the base software.

For the HMMs generated in this research, we experimented with various numbers of hidden states. We found that the number of hidden states was not a critical parameter, and consequently all experiments presented in this paper use HMMs with \(N = 2\).

The next section discusses our experimental design. Both the training and scoring phases are considered in detail.

## 3 Design Overview

Our technique consists of two phases—a training phase and a detection phase. In the training phase, a hidden Markov model is trained using slightly morphed copies of
In the detection phase, we score the suspect software against the model derived in the training phase. Before we discuss training and scoring, we first discuss our metamorphic generator.

### 3.1 Metamorphic Generator

Our metamorphic generator produces morphed opcode sequences derived from any given piece of software. As discussed in Section 2.1.1, there are a number of metamorphic techniques that can be used to generate metamorphic software. For the test cases considered here (see Section 4), we only employ dead code insertion. As mentioned in Section 2.1.1, previous research has shown that inserting carefully chosen dead code is significantly more effective at defeating HMM-based detection than other elementary morphing techniques [17].

To simplify the morphing process, we do not require that the resulting morphed code maintain the functionality of the original code. That is, we insert dead code without inserting the corresponding jump instructions that would be required to avoid executing the dead code. Note that in general, this makes detection more difficult, since the missing jump instructions would make it easier for an HMM-based detector to distinguish the modified code from the original. Furthermore, the increased number of jump instructions could itself provide a useful heuristic, as well as providing a means of identifying and removing dead code.

As described in more detail below, the code used for morphing is selected from code that is similar to that used in setting the scoring threshold. This serves to make detection more challenging—as compared to using random morphing or code selected from random files—since our morphing technique tends to make the morphed code more similar to the code used to determine the “no-match” threshold. The bottom line is that our tests are designed to be a worst-case scenario (from the perspective of detection), with respect to both the morphing technique and morphing code selection.

### 3.2 Training

In this phase, the opcode sequence from the base software is extracted. We then create multiple slightly morphed copies of this base opcode sequence. These morphed sequences are appended and a hidden Markov model is trained on the resulting sequence. An overview of the training phase appears in Figure 2.

Morphing in the training phase is used solely to prevent the HMM from overfitting the data. Previous work indicates that a 10% morphing rate will achieve this goal [20], so we have used 10% morphing of the base file in all of our training cases. We experimented with different distributions of the dead code within the base file, but this had little effect on our results and, consequently, for the remainder of this paper, we consider the case where a single block of dead code is used for training; see [13] for details on the additional cases. In contrast, for the scoring phase, the number
blocks of dead code was found to be a significant parameter. This is discussed in more detail below and again when we present our experimental results in Section 4.

The morphed training copies of the base software are used as the dataset for training the HMM in a manner analogous to that used in [38]. Five-fold cross-validation is used [14], that is, the morphed files are split into five equal subsets, and four of the subsets are selected for training. The opcodes from the training subsets are concatenated to obtain a single observation sequence, which is used to train an HMM. The files in the remaining subset are then each scored using the trained model, along with a set of "normal" files, and a threshold is set based on these scores. The process is repeated five times, with a different subset reserved for scoring each time. The models are then compared—if they do not yield similar results, there is likely a strong bias in some of the data. For all of our experiments, the five models yielded comparable results. Figure 3 illustrates the process used to set the threshold.

Once the process in Figure 3 is complete, we have a model and a threshold. We are now in position to score the suspect software, using the procedure described in the next section.

3.3 Scoring

In the detection or scoring phase, we first extract the opcode sequence from the given suspect software. This opcode sequence is scored against the HMM model that was derived in the training phase and the score is compared to the previously derived threshold. A score above the threshold indicates that the suspect software is sufficiently similar to the original software to warrant further investigation. In contrast, a score below the threshold indicates that the suspect software is not similar to the original software. Figure 4 gives a high-level illustration of the detection phase.

To determine the robustness of our approach, the base software was morphed and scored against the model. At this "tampering" phase, the morphing serves to simulate the situation where an attacker modifies the code. We assume that the modifications are designed to evade detection, but in practice the attacker might also change the code for other reasons, such as to modify its behavior in some way.

Figure 5 illustrates the process used to generate the tampered files and score them against the trained model. Detection rates were averaged over a series of experiments, as illustrated in Figure 6.

4 Experimental Results

In this section, we present our experimental results and provide some discussion of the results. Graphs of results are provided and discussed, but first we provide details on the experimental setup.
4.1 Experimental Details

For our experiments, we select a base file from the set of Cygwin utility files. These files have been used as representative examples of “normal” (i.e., non-virus) files in previous studies [17, 20, 38]. The size of each base file ranged from 80 KB to 110 KB.

Training the HMM model on the base code and scoring follows the process discussed in Section 3. Each experiment was repeated 10 times, with a different base file selected each time. Specifically, the following process was used:

- IDA Pro DisAssembler 5.0 [12] was used to disassemble the base file
- The opcode sequence was extracted from the disassembled base file
- We generated 100 morphed copies of the base file, each with 10% morphing and using one block of dead code. Recall that morphed copies of the base software are used to avoid having the HMM overfit the training data.
- Hidden Markov models were trained using the 100 morphed copies with five-fold cross validation
- A threshold was determined based on scores for the morphed files that were not part of the training, along with a set of 15 “normal” files that were otherwise not involved in the morphing or training. The threshold was set at the highest score for any of the “normal” files.

Note that the “normal” files are playing the role of innocent files that are scored against the model. As with the base file, the normal files are selected from Cygwin utility files. The normal files are of roughly similar size to the base files and they would be expected to have more in common with the base file than randomly selected code—as would be expected of suspect files we would score using the trained model.

By setting the threshold as the highest score among the 15 normal files, and assuming the suspect files we actually test are somewhat similar to the normal files, we can expect a false positive rate of less than 1/15, or about 6.67%. If we were to lower the threshold, we would reduce the false negative rate, at the expense of having a higher false positive rate. For the software piracy problem under consideration, a higher false positive rate would likely be tolerable, since we would generally only test a relatively small number of suspect cases, and we certainly want to reduce the false negative rate as much as is practically feasible for these suspect cases.

For testing the robustness of our scheme, various tampering percentages were used, ranging from 10% to 100%, in increments of 10%. In addition various distributions of the tampering code within the files were considered. Specifically, the dead code was inserted as 1, 2, 4, 8, 16, and 32 blocks throughout the tampered file.

In total, 6000 tampered files were generated for each base file tested. Since the process was repeated for 10 distinct base files, 60,000 tampered files were generated and scored for each test case. The detection rate was computed as the average detection rate over these 10 experiments. Note that this averaging process was carried out for each tampering percentage considered (10% to 100%, in increments of 10%) and each block insertion case (1, 2, 4, 8, 16, and 32), for a total of 60 cases.
This entire training and scoring process was repeated 6 times, using different numbers of blocks in the training phase. Consequently, a total of 360,000 tampered files were generated and scored. However, the number of blocks used in the training phase was found to have a minimal effect on the results, so in the remainder of this paper, we only consider one of these six different training cases; see [13] for details on the other five cases.

4.2 Detection Results

Detection rates are given in Figure 7. The detection percentage reflects the number of tampered versions of the base file that were properly classified as being similar to the base file. Consequently, the false negative rate is one minus the given detection rate. Also, as discussed in Section 3, due to the way the thesholds are set, the expected false positive rate is less than 6.67% in each case.

As expected, when the tampering rate increases, detection success decreases. But, perhaps surprisingly, the number of tampering blocks has a significant effect on detection rates, especially at higher tampering rates. The results in Figure 7 show that 1-block tampering has a very strong effect at higher rates—with 70% or more tampering, none of the 1-block tampered files are classified correctly. The 2-block tampering and 32-block tampering cases also differ significantly from the remaining cases. However, at moderate tampering rates, these differences are much less pronounced.

The results in Figure 7 are consistent with previous work on metamorphic malware. For example, in [17], it is shown that a single large block of dead code selected from a non-virus file is significantly more effective at evading HMM-based detection than mixing the dead code more uniformly within the virus. This same phenomenon is observed here—a single large block of dead code has a more detrimental effect on HMM-based detection than dead code that is intermixed throughout the file.

Figure 8 gives a 3-dimensional representation of the detection rate versus the tampering rate and number of blocks used. As in Figure 7, these results again show the strong effect of 1-block tampering as compared to higher numbers of blocks, as well as illustrating the decrease in success at higher tampering rates.

Our results indicate that the best strategy for evading our HMM-based detector is to insert a single, large block of dead code. This is also relatively easy for an attacker to accomplish, as compared to mixing dead code within the file. However, there are some significant drawbacks to using such a simple tampering strategy. For one, we could overcome this attack by computing multiple scores on the file, omitting different sections of code for each score calculation. If any of these tests omit a large percentage of the dead code, we would obtain a high score, and any high score is indicative of similarity to the base software. In addition, an attacker would have to put some effort into disguising the dead code to avoid having it removed in a preprocessing step—one such strategy is analyzed in [22]. Consequently, a 1-block strategy is unlikely to be the attacker’s panacea that it might seem at first glance.
5 Conclusions and Future Work

In this paper, we considered a software piracy detection scheme inspired by previous work on metamorphic malware detection. The scheme is efficient, it can be applied after the fact, and requires no special effort when developing or maintaining software.

Experimental results show that our scheme is robust, in the sense that the base software must be extensively modified before we fail to detect it with a high probability. In addition, real-world results are likely to be better than our experiments indicate. In our experiments, we assume that the attacker chooses the morphing code optimally, in the sense that morphing code is chosen from files similar to those used to set the threshold, and therefore morphing is highly effective at diminishing the HMM scores. In addition, we do not consider the difficulties the attacker would face in maintaining the functioning of the code when morphing. In practice the attacker would be at a significant disadvantage in these respects—the “morphing” code might include some modifications to the code, thereby limiting the morphing options, information on the files used for thresholding would be lacking, and maintaining the desired functionality of the code would be a significant issue.

We set a threshold that under reasonable assumptions limits the false positive rate to about 6.5%. If we are willing to accept a higher false positive rate, we could obtain lower false negative rates. As discussed in Section 4, for this particular problem, a higher false positive rate would likely be acceptable in practice.

Our experiments show that the optimal strategy for an attacker is to insert one large block of dead code. However, in practice the effectiveness of such a strategy could likely be overcome using some of the approaches discussed in Section 4.2.

Future work could include the use of additional morphing techniques and more experiments on a wider variety of file types (e.g., Java bytecode). Also, since 1-block tampering can be the most effective attack strategy, further experiments designed to mitigate the effectiveness of such an attack would be useful.

References


http://www.csi.ucd.ie/content/using-whitespace-patterns-detect-plagiarism-program-code


http://portal.bsa.org/globalpiracy2010/


http://www.cs.sjsu.edu/faculty/stamp/papers/Buffer0verflow.doc

[12] IDA Pro DisAssembler
http://www.hex-rays.com/index.shtml/


http://www.ma.rhul.ac.uk/static/techrep/2008/RHUL-MA-2008-02.pdf/

http://www.ladas.com/Patents/Computer/SoftwareAndCopyright/Softwa06.html


http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf/


Markov process: \[ X_0 \xrightarrow{A} X_1 \xrightarrow{A} X_2 \xrightarrow{A} \ldots \xrightarrow{A} X_{T-1} \]

Observations: \[ O_0 \xrightarrow{} O_1 \xrightarrow{} O_2 \xrightarrow{} \ldots \xrightarrow{} O_{T-1} \]

Figure 1: Hidden Markov Model [31]

Figure 2: Training Phase
Figure 3: Setting the Threshold

Figure 4: Detection Phase
Figure 5: Generating Tampered Files
Figure 6: Detection Rate

Figure 7: Detection Results
Figure 8: Results Summary