J Comput Virol Hack Tech
DOI 10.1007/s11416-013-0185-4

ORIGINAL PAPER

# Structural entropy and metamorphic malware

**Donabelle Baysa · Richard M. Low · Mark Stamp**

Received: 18 December 2012 / Accepted: 25 March 2013

**Abstract** Metamorphic malware is capable of changing its internal structure without altering its functionality. A common signature is nonexistent in highly metamorphic malware and, consequently, such malware can remain undetected under standard signature scanning. In this paper, we apply previous work on structural entropy to the metamorphic detection problem. This technique relies on an analysis of variations in the complexity of data within a file. The process consists of two stages, namely, file segmentation and sequence comparison. In the segmentation stage, we use entropy measurements and wavelet analysis to segment files. The second stage measures the similarity of file pairs by computing an edit distance between the sequences of segments obtained in the first stage. We apply this similarity measure to the metamorphic detection problem and show that we obtain strong results in certain challenging cases.

## 1 Introduction

Metamorphism is a technique that changes the internal structure of software while maintaining its functionality. Malware writers use metamorphism to evade signature-based detection.

Previous research has shown that some opcode-based software similarity measures provide promising results when applied to hacker-produced metamorphic malware [16,20, 35]. However, it has also been shown that it is possible to construct metamorphic code that is not accurately detected using these methods [26].

This research applies the file similarity method developed in [25] to the metamorphic detection problem. This technique uses the concept of structural entropy to analyze the complexity of data within a file. The method consists of two stages, namely, file segmentation and sequence comparison. In the file segmentation stage, an entropy measure and wavelet analysis is used to segment files. The second stage estimates the similarity of a pair of files based on an edit distance between file sequences obtained in the first stage. We apply this approach to classify whether a given file belongs to a specific metamorphic malware family.

This report is organized as follows. Section 2 contains relevant background information. Section 3 describes the design of our scoring method, which closely follows the work in [25]. In Sect. 4, we present our experimental results. Finally, Sect. 5 gives our conclusions and suggests directions for future work.

## 2 Background

To evade signature detection, malware writers often employ advanced obfuscation techniques, including encryption, polymorphism, and metamorphism [4]. It is easily proved that well-designed metamorphic malware is immune to standard signature detection [7].

In this paper, we consider a similarity technique for metamorphic detection. This technique relies on static analysis of executable files. The approach considered here applies directly to binary executables, while most similarity techniques rely on extracted opcode sequences [20,35]. The use of opcode sequences adds significant to scanning overhead [18,22].

D. Baysa · M. Stamp (✉)
Department of Computer Science, San Jose State University,
San Jose, USA
e-mail: stamp@cs.sjsu.edu

R. M. Low
Department of Mathematics, San Jose State University, San Jose, USA

Published online: 14 April 2013

Next, we provide brief background information on malware, with the emphasis on methods used to evade signature detection. Then, we review representative examples of file similarity measures that have been applied to the metamorphic malware detection problem. We also discuss the concept of structural entropy and explain how we apply it in the similarity method considered here.

## 2.1 Malware

As the name implies, malware is software that is designed to cause harm. Examples of such harm include interruption of normal computer operations, destruction of files by infection or deletion, theft of user information, and damage to system resources, such as the hard disk [29]. Malware comes in various forms including viruses, worms, spyware, and trojan horses. In this paper, we restrict our attention to viruses and worms.

### *2.1.1 Viruses and worms*

Different authors employ differing definitions to distinguish between viruses and worms. But by any definition, both viruses and worms are types of malware that replicate and spread within or between hosts [4]. For our purposes, the distinction is that computer viruses are parasitic, in the sense that they embed themselves in other executable program [29], while worms are standalone alone programs [29].

Viruses and worms employ a number of obfuscation techniques to avoid signature-based detection. The goal is to make each copy of the malware will look sufficiently different so that it is impossible to determine any commonality.[1]

In this paper, we focus on metamorphic malware. Examples of metamorphic malware include the Mental Driller's MetaPHOR worm [15], SnakeByte's Next Generation Virus Creation Kit (NGVCK) [24], and the recently-developed research worm analyzed in [26].

Next, we briefly consider various obfuscation techniques. Then we turn our attention to executable file similarity measures. Note that throughout the remainder of this paper, we often use "virus" generically as a synonym for malware.

### *2.1.2 Obfuscation techniques*

Encryption is a relatively simple form of code obfuscation. If each virus body is encrypted using a different key, then the virus bodies will exhibit no common pattern, thereby evading signature detection. However, an encrypted virus must include decryption code, which is subject to signature scanning [36].

Polymorphic malware takes encryption one step further—the decryptor code is morphed between generations [21,36]. As with encrypted code, polymorphic malware relies on encryption to obfuscate the virus body, while code morphing serves to obfuscate the decryptor. Therefore, well-designed polymorphic malware will exhibit no common signature. However, code emulation is a useful AV technique for polymorphic detection. A polymorphic virus decrypts itself at which point it exposes the virus body, which is then susceptible to signature detection. Also, the code decryption process may be susceptible to detection [4].

Metamorphic viruses take code morphing to the limit, that is, the entire code is morphed between generations. Such viruses change their internal structure at each replication while maintaining their original functionality [21,36]. A highly metamorphic virus does not require encryption since each generation is structurally different, and hence retrieval of a common signature is highly unlikely [26].

Next, we consider examples of elementary code morphing techniques. These methods are generally applied at the assembly code level and, consequently, our examples are given in terms of assembly code.

Perhaps the simplest code morphing method is to swap registers. For example, `ADD EDX,0088h` can be converted to `ADD EAX,0088h`, i.e., the register `EDX` can be swapped for `EAX`, assuming `EAX` is not in use. Note that register swapping has no effect on the opcode mnemonic. Furthermore, register swapping can often be detected by signature scanning using wildcards [4].

Equivalent instruction substitution is a more general code morphing technique. An instruction, or set of instructions, can be exchanged for another that yields the same result. For example, `SUB EAX,EAX` is equivalent to `XOR EAX, EAX`.

Provided that instructions have no dependencies, the instructions can be reorder without changing the code functionality. For example, the instructions

```
1: SUB R1, R2
2: SUB R3, R4
```

can be replaced by

```
1: SUB R3, R4
2: SUB R1, R2
```

Code transposition is a potent anti-signature technique.

Subroutine permutation is a simple example of code transposition. A program with $n$ independent subroutines can be easily transformed into $n!$ variants using only subroutine permutation. However, this method alone can expose the virus to detection based on multiple short signatures [4].

Garbage instruction insertion is an effective metamorphic technique. Garbage instructions are those that are either executed without impacting virus behavior or skipped altogether.

[1] To be successful, it is also necessary that the malware, as a group, must be sufficiently similar to non-viral code [26,35].

The former type of garbage instructions are sometimes referred to as "do-nothing" code, while the latter is "dead code." Elementary examples of do-nothing code include `NOP` and `PUSH EAX` followed by `POP EAX`. Both dead code and do-nothing code can be made arbitrarily difficult to detect [11].

## 2.2 File similarity methods

In this section, we review previous research into metamorphic detection. Our purpose here is to present representative examples of techniques, not an exhaustive list.

### *2.2.1 HMM-based detection*

In hidden Markov model (HMM) analysis, we assume where there is an underlying Markov process, but the states are not observable. Some output from the system can be observed, and these observations depend (probabilistically) on the hidden states [27]. Figure 1 illustrates a generic HMM, where the portion above the dotted line is not directly observable. The Markov process consists of the hidden states $X_t$ which are "driven" by the state transition probability matrix $A$. Each hidden state is related to the correspondingobservation $\mathcal{O}_t$ by probability distributions contained in the matrix $B$.

Given a sufficient number of observations, we can train an HMM to represent the data. We can then score a given sequence of observations to determine how well the model fits the observations—a high score indicates that the observed sequence is closely related to the training data [27].

The paper [35] presents a metamorphic detector based on HMMs. An HMM is trained on opcode sequences extracted from members of a given metamorphic family. A threshold is determined and unknown files are then scored against the model. The work in [35] shows that the HMM yields detection results that are superior to popular commercial virus scanners when applied to a hacker-produced metamorphic virus families.

### *2.2.2 n-gram similarity*

The paper [35] also analyzes a similarity measure based on $n$-gram analysis. In this techniques, we compute the similarity of two opcode sequences by finding all matches of three consecutive opcode at any offset, in any order. For example, the opcode sequence

```
(ADD, SUB, MOV)
```

from one file would match the sequence

```
(SUB, MOV, ADD)
```

at any offset in the other file. Each matching subsequence is plotted as a point in the $(x, y)$ plane, where $x$ corresponds to the offset of the subsequence from the first file, and $y$ is the offset of the matching subsequence from the second file. Then noise reduction is applied by removing any segment that does not include points of the form $(x+i, y+i)$, for $i = 0, 1, \ldots, k$, with $k \geq 4$, that is, we discard any segment with diagonal length less than five. Finally, the score is computed by measuring the fraction of the axes that are covered by at least one segment. Figure 2 illustrates this process.

Using this $n$-gram approach, we can obtain a metamorphic detection technique by computing the similarity of an unknown file to a known member of a metamorphic virus family. We use a predetermined threshold to classify the unknown file as either belonging to the metamorphic family or not. In [35], this $n$-gram score is shown to be highly effective at detecting hacker-produced metamorphic viruses.

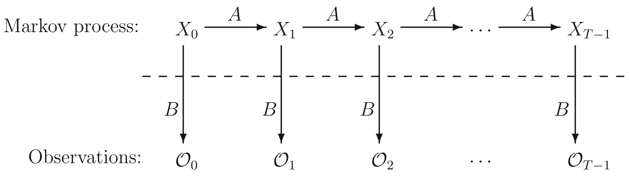


**Fig. 1** Generic hidden Markov model [27]

### *2.2.3 Opcode graph-based similarity*

A similarity score based on opcode graphs is considered in [20]. This graph similarity measure, which outperforms an HMM-based detection in certain cases, relies on a weighted directed graph constructed from extracted opcode sequences. The graph has a node for each distinct opcode and an edge is inserted from each opcode node to all of its successor opcode nodes. An edge weight is assigned based on the probability of the successor opcode node. Figure 3 gives an example of such a weighted opcode graph.

To compare two files, the opcode graph is built for each file. The opcode graphs are then compared directly via a scoring function based on the corresponding edge weights. Once a threshold is set, we can use this technique to score files and classify them as members of a given metamorphic family or not.

In the next section, we discuss a similarity measure from [25], which is based on structural entropy. We also discuss the design of our approach for applying this technique to the metamorphic detection problem.

## 3 Similarity via structural entropy

The similarity method considered here is derived from the paper [25]. This technique is based on static analysis of files using structural entropy as the basis for a similarity measure.

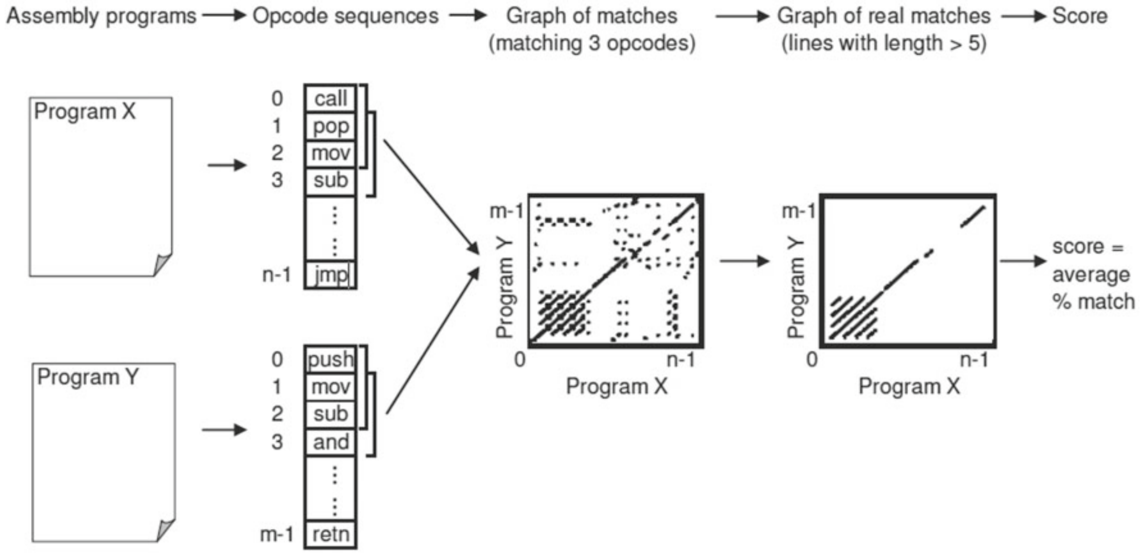
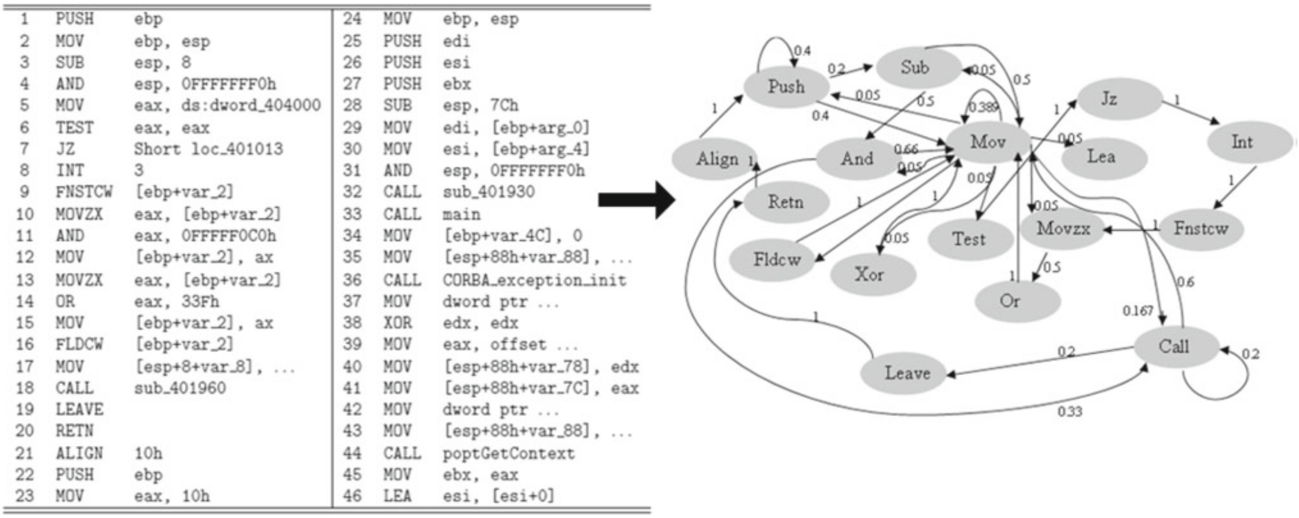


**Fig. 2** *n*-gram similarity [35]



```
PUSH    ebp
MOV     ebp, esp
SUB     esp, 8
AND     esp, 0FFFFFFF0h
MOV     eax, ds:dword_404000
TEST    eax, eax
JZ      Short loc_401013
INT     3
FNSTCW  [ebp+var_2]
MOVZX   eax, [ebp+var_2]
AND     eax, 0FFFFF0C0h
MOV     [ebp+var_2], ax
MOVZX   eax, [ebp+var_2]
OR      eax, 33Fh
MOV     [ebp+var_2], ax
FLDCW   [ebp+var_2]
MOV     [esp+8+var_8], ...
CALL    sub_401960
LEAVE
RETN
ALIGN   10h
PUSH    ebp
MOV     eax, 10h
MOV     ebp, esp
PUSH    edi
PUSH    esi
PUSH    ebx
SUB     esp, 7Ch
MOV     edi, [ebp+arg_0]
MOV     esi, [ebp+arg_4]
AND     esp, 0FFFFFFF0h
CALL    sub_401930
CALL    main
MOV     [ebp+var_4C], 0
MOV     [esp+88h+var_88], ...
CALL    CORBA_exception_init
MOV     dword ptr ...
XOR     edx, edx
MOV     eax, offset ...
MOV     [esp+88h+var_78], edx
MOV     [esp+88h+var_7C], eax
MOV     dword ptr ...
MOV     [esp+88h+var_88], ...
CALL    poptGetContext
MOV     ebx, eax
LEA     esi, [esi+0]
```

**Fig. 3** Assembly instructions to opcode graph [20]

In contrast to the similarity methods discussed in the previous section, this similarity score is computed directly on executable files, i.e., no expensive disassembly step is required.

The method analyzed here compares two given files and produces a similarity measure. We start by splitting each file into segments of varying entropy levels using wavelet analysis applied to raw entropy measurements. Given two files that have thus been segmented, we determine a similarity score by computing the edit distance between the corresponding segments. Next, we discuss this process in detail.

### 3.1 File segmentation

The key to properly segmenting a file is to locate the areas where significant changes in entropy occur. The standard structure of an executable includes information about its various sections. For example, Fig. 4 shows the format of a Windows portable executable (PE) file [17,19]. However, for the method considered in this paper, we directly analyze the bytes within an executable file. In particular, we do not make explicit use of the information in the executable file header. This file header information would be of little use since we are interested in relatively fine-grained variations within a file. As a side benefit, by ignore header-specific section information, our technique applies without modification to any executable file format, be it Windows PE, Unix ELF [9], or other.

#### *3.1.1 Wavelet transform analysis*

Wavelet analysis is a process of transforming a signal (i.e., a data set) into a more revealing and useful form. Wavelets

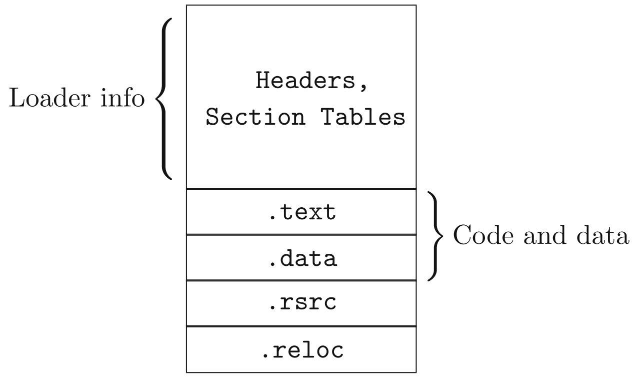


**Fig. 4** Executable file format

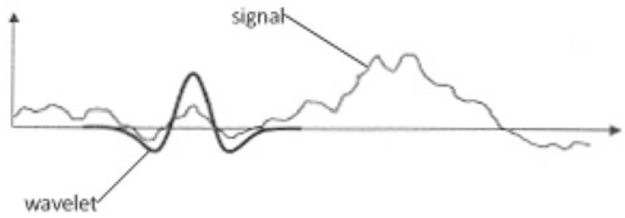


**Fig. 5** Wavelet and signal [1]

are wavelike functions that can be used to analyze raw data in different locations at different scales [1,28,33]. Figure 5 illustrates a signal and a wavelet.

The transformation at a given scale is determined based on the signal approximation and detail at the previous scale. Scaling the wavelet smooths out the high frequency information present in the original data. The transformed data is referred to as a *wavelet transform*, which represents the relationship between the data and the analyzing wavelet. Mathematically, a wavelet transform is a convolution of the signal and the wavelet function over a range of locations and scales [1].

#### *3.1.2 Segmentation using wavelet transform*

We use wavelet analysis to determine those areas in a file where significant changes in entropy occur. The process is as follows. First, we apply a sliding window and compute the corresponding series of entropy levels. Specifically, we determine entropy values which we denote $V = \{v_i : i = 1, \ldots, N\}$, where $N$ is the number of windows and $v_i$ is the entropy of the data within window $i$. Entropy is calculated using Shannon's formula [6,14], that is,

$$v_i = -\sum_{x \in B_i} p(x) \log_2 p(x), \tag{1}$$

where $p(x)$ is the relative frequency of byte $x$ within the $i$th window, or block, $B_i$.

Next, a discrete wavelet transform is applied to the entropy series $V$, that is [30],

$$W(a, b) = \frac{1}{|a|^{1/2}} \sum_{i=1}^{N} v_i \cdot \psi_{HAAR}\left(\frac{t_i - b}{a}\right), \tag{2}$$

where $a$ is a scaling parameter, $b$ is a shifting parameter of the analyzing wavelet, $v_i$ is the entropy of window $i$, window $i$ is at position $t_i$ in the file, $N$ is the number of windows, and $\psi_{HAAR}$ is the Haar wavelet function. The Haar wavelet function is defined by

$$\psi_{HAAR}(t) = \begin{cases} 1, & 0 \le t < 1/2, \\ -1, & 1/2 \le t < 1, \\ 0, & t < 0, t \ge 1. \end{cases}$$

The wavelet transform formula in (2) requires the scaling parameter $a$ to be a power of 2. For example, if we want to transform the raw data to a maximum scale of 16, Eq. (2) will be used successively for $a = 2, 4, 8,$ and 16 and for different locations $b$ in the data. For our purposes, the file segments are determined by the wavelet coefficients at the maximum scale, and boundaries of the segments are determined by local extrema based on a threshold of 0.5. This process is summarized in Fig. 6.

This file segmentation method is illustrated in Fig. 7, which shows the wavelet transform of a sample file, with the corresponding entropy plot given in Fig. 8. On the lower transformation scales, minor changes in entropy levels are evident. In contrast, insignificant changes are ignored at the higher scales. This is why the segments are determined by the coefficients at the highest scale.

### 3.2 Sequence comparison

Once files have been segmented, we need a method to measure the similarity between two files, based on their segments. As in [25], we use Levenshtein distance [13] to compute scores between pairs of files.

In the remainder of this section, we first discuss Levenshtein distance. Then we illustrate how this distance measure is applied to segmented files. Finally, we discuss the entropy similarity score calculation.

#### *3.2.1 Levenshtein distance*

Levenshtein distance, or edit distance, is a measure of the difference between two sequences [13,34]. This distance measure calculates the minimum number of edit operations required to transform one sequence into the other. The set of allowed edit operations are substitution, insertion, and deletion. The substitution operation consists of replacing an element from the first sequence with an element in the second. As the name implies, an insertion consists of inserting a new

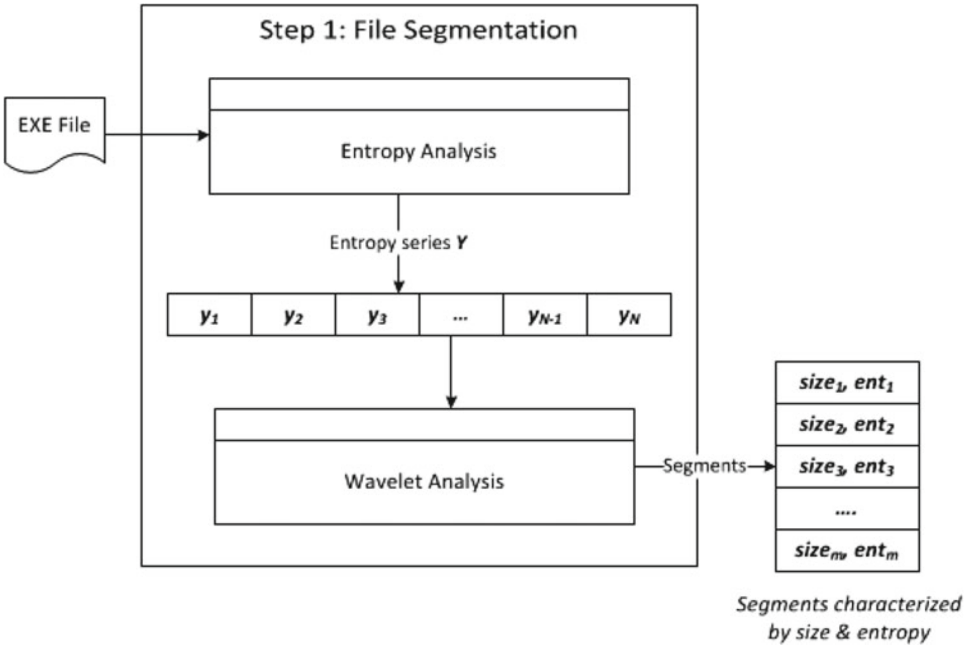


**Fig. 6** File segmentation

element into the second sequence. Finally, a deletion removes an element from the second sequence [2]. Each required edit operation adds to the overall distance.

To illustrate an edit distance computation, consider the two sequences, "eleven" and "elevated". Suppose that each substitution, insertion, or deletion results in a penalty of 1. We want to determine the minimum number of edit operations required to transform the string "eleven" to "elevated". This minimum is at least three, since the following three edits suffice:

1. eleven → elev**a**en (*insert* **a**)
2. elevaen → eleva**t**en (*insert* **t**)
3. elevaten → elevate**d** (*substitute* **d** *for* n)

Next, we show that this transformation cannot be completed in fewer than three edits, that is, we show that the Levenshtein distance between these two strings is 3.

Consider a pair of sequences $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_m)$. Then we can define a recurrence for Levenshtein distance between these two sequences, using a general cost function as follows [2]. We compute the elements of the matrix $D_{(n+1)\times(m+1)} = \{d_{i,j}\}$ using the recursion

$$d_{i,j} = \begin{cases} 0 & \text{if } i = j = 0 \\ d_{0,j-1} + \delta_Y(j) & \text{if } i = 0 \text{ and } j > 0 \\ d_{i-1,0} + \delta_X(i) & \text{if } i > 0 \text{ and } j = 0 \\ d_{i-1,j-1} & \text{if } x_i = y_j \\ \min \begin{Bmatrix} d_{i,j-1} + \delta_Y(j) \\ d_{i-1,j} + \delta_X(i) \\ d_{i-1,j-1} + \delta_{X,Y}(i,j) \end{Bmatrix} & \text{if } x_i \neq y_j \end{cases} \tag{3}$$

where $\delta_Y(j)$ is the cost of inserting $y_j$ into the $Y$ sequence, $\delta_X(i)$ is the cost of deleting $x_i$ from the $X$ sequence, and $\delta_{X,Y}(i, j)$ is the cost of substituting $y_j$ in place of $x_i$. Then $d_{n,m}$ gives the desired distance.

Applying (3) to our previous example, with $\delta_Y = \delta_X = \delta_{X,Y} = 1$, yields the edit matrix

| | | **e** | **l** | **e** | **v** | **a** | **t** | **e** | **d** |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **e** | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **l** | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| **e** | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| **v** | 4 | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 |
| **e** | 5 | 4 | 3 | 2 | 1 | 1 | 2 | 2 | 3 |
| **n** | 6 | 5 | 4 | 3 | 2 | 2 | 2 | 3 | **3** |

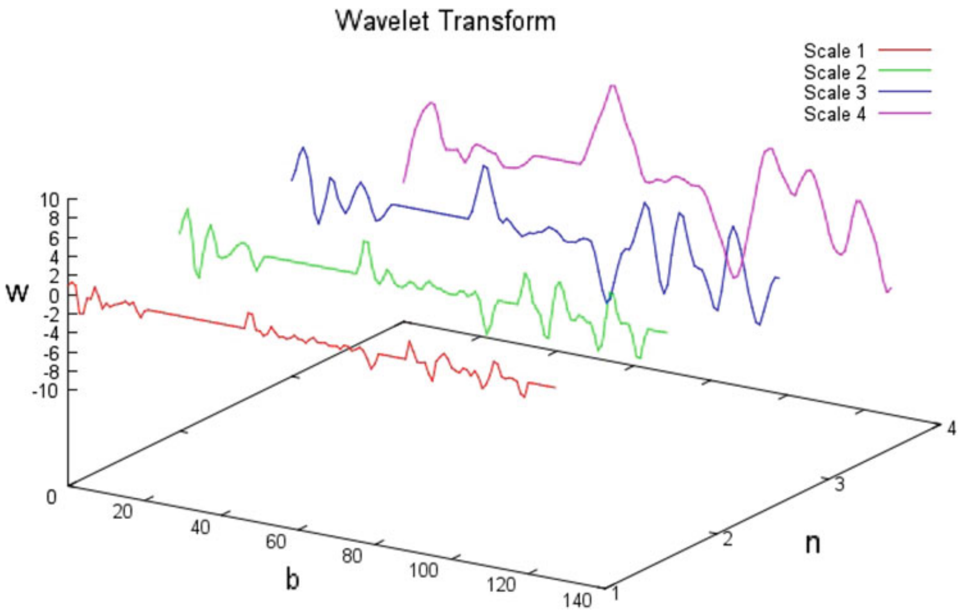
The lower-right element represents the edit distance, which is 3, as claimed above.

### *3.2.2 Sequence alignment using Levenshtein distance*

Let $X$ be a file where $x_i$ for $i = 1, 2, \ldots, n$ are the segments obtained during the segmentation phase described in Sect. 3.1. Similarly, let $Y$ be a file with segments $y_j$ for $j = 1, 2, \ldots, m$. Let $\sigma(x_i)$ be the size of segment $x_i$ and $\varepsilon(x_i)$ its entropy. For the sequence alignment scheme considered here, we employ a cost function that reflects both the differences in segment size and entropy value. As in [25], we penalize size differences according to the function

$$\text{cost}_\sigma(x_i, y_j) = \frac{|\sigma(x_i) - \sigma(y_j)|}{\sigma(x_i) + \sigma(y_j)}. \tag{4}$$

**Fig. 7** Wavelet transform of a sample file



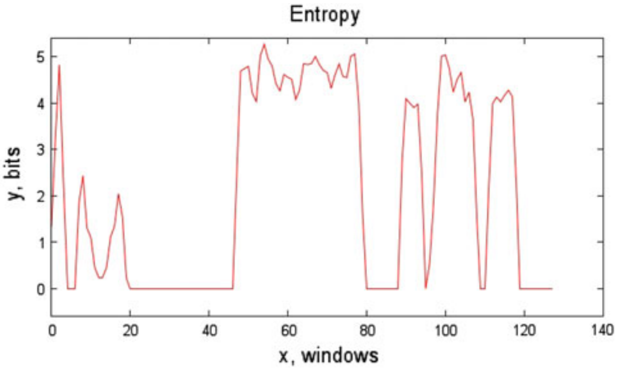


**Fig. 8** Entropy plot of a sample file

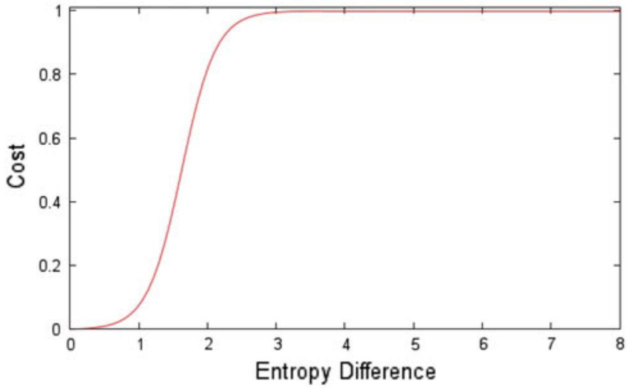


**Fig. 9** Graph of $\text{cost}_\varepsilon$

Note that the minimum value of $\text{cost}_\sigma$ is 0, which is attained when the segments are of equal size, while the maximum of $\text{cost}_\sigma$ is 1.

For difference in entropy, we use the function [25]

$$\text{cost}_\varepsilon(x_i, y_j) = \frac{1}{1 + \exp(-4 \cdot |\varepsilon(x_i) - \varepsilon(y_j)| + 6.5)} - 0.001501. \tag{5}$$

The function (5) represents a sigmoid curve [31], as illustrated in Fig. 9. The use of constants 6.5 and 0.001501 in (5) bounds $\text{cost}_\varepsilon$ between 0 and 1.

We combine the size penalty in (4) and entropy penalty in (5) to obtain

$$\text{cost}(x_i, y_j) = c_\sigma \cdot \text{cost}_\sigma(x_i, y_j) + c_\varepsilon \cdot \text{cost}_\varepsilon(x_i, y_j), \tag{6}$$

where $c_\sigma$ and $c_\varepsilon$ are constants that are used to weight the size and entropy costs, respectively. We apply the cost function (6) when computing the Levenshtein distance between segmented sequence. For the distance calculation, we build the edit distance matrix $D = \{d_{i,j}\}$ using (3) and dynamic programming. The distance between files $X$ and $Y$ is determined by the lower-right element of $D$ array, that is, $d_{n,m}$. To fill in the array $d$, we employ (3) with

$$\begin{aligned}
\delta_Y(j) &= \tau \log \sigma(y_{j-1}) \\
\delta_X(i) &= \tau \log \sigma(x_{i-1}) \\
\delta_{X,Y}(i, j) &= \text{cost}(x_{i-1}, y_{j-1}) \cdot \log\left(\frac{\sigma(x_{i-1}) + \sigma(y_{j-1})}{2}\right)
\end{aligned} \tag{7}$$

and neglecting the case corresponding to $x_i = y_j$ (i.e., outside of the first row and column, we always use the "min" equation corresponding to $x_i \neq y_j$). The values selected for the constants $c_\sigma$, $c_\varepsilon$, and $\tau$ (and other parameters) are given in Sect. 4.2.

### *3.2.3 Similarity calculation*

Using the edit distance determined from (7), we compute the similarity of files $X$ and $Y$ as

$$\text{similarity} = 100\left(1 - \frac{d_{n,m}}{\text{cost}_{\max}}\right), \tag{8}$$

where $\text{cost}_{\max}$ is the penalty in the worst-case scenario. Intuitively, this worst case occurs when all elements of the first sequence are deleted, and all elements of the second sequence are inserted, which would imply

$$\text{cost}_{\max} = d_{0,m} + d_{n,0}.$$

However, as in [25], we use a slightly different formulation for $\text{cost}_{\max}$. Define

$$\begin{aligned}\delta'_Y(j) &= \delta_Y(j)\\ \delta'_X(i) &= \delta_X(i)\\ \delta'_{X,Y}(i, j) &= 2\tau(\log\sigma(x_{i-1}) + \log\sigma(y_{j-1}))\end{aligned}$$

and compute $d'_{0,m}$ and $d'_{n,0}$ using these functions. Then we let

$$\text{cost}_{\max} = d'_{0,m} + d'_{n,0}. \tag{9}$$

The net effect of the modified version of $\text{cost}_{\max}$ in (9) is to slightly increase the range of scores obtained in (8). Experiments indicate that this provides a marginal improvement in the score [5].

Figure 10 summarizes the process used to score two entropy transform sequences.

## 4 Experiments and results

We applied the method presented in the previous section to the metamorphic virus detection problem. The process we used is analogous to that in [20], except that we compute more scores for each test case.

To classify executable files, we must first determine a scoring threshold. To set a threshold, we perform the following steps:

1. Determine the similarity range for a metamorphic virus family by calculating the similarity score for all pairs of virus files.
2. Determine the similarity range for the metamorphic files used in step 1 versus a set of representative benign files by calculating the similarity score for all pairs.
3. Set a threshold by analyzing the values obtained in Steps 1 and 2.

In the ideal case, the two distributions are disjoint, that is, the lowest score from Step 1 is higher than the highest score from Step 2. In such a case, any threshold between these two extremes will provide perfect detection, i.e., there will be no false positives or false negatives. In general, receiver operator characteristic (ROC) curves provide a useful means of measuring the separation (or lack thereof) between the distributions. For an ROC curve, the area under the curve (AUC) quantifies the separation—an AUC of 1.0 implies ideal separation, while and AUC of 0.5 implies that separation is no better than flipping a coin [8].

### 4.1 Test data

For our experiments, we considered three metamorphic families. We used 50 virus files generated by Second Generation

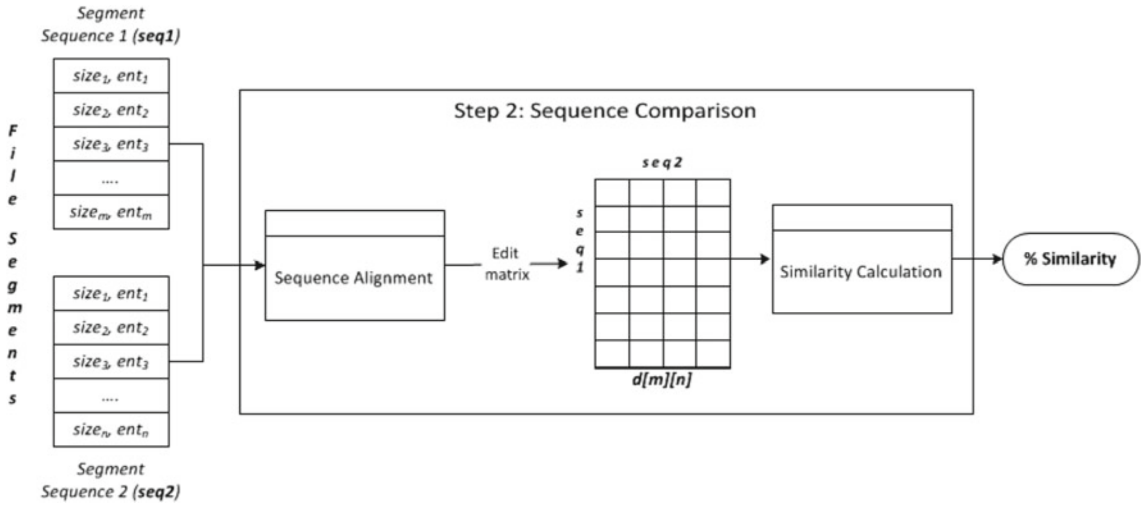


**Fig. 10** Sequence comparison

**Table 1** Cygwin benign files

| | |
|---|---|
| ascii.exe | msgtool.exe |
| banner.exe | putclip.exe |
| conv.exe | readshortcut.exe |
| cygdrop.exe | realpath.exe |
| cygstart.exe | getclip.exe |
| dump.exe | semstat.exe |
| lpr.exe | semtool.exe |
| mkshortcut.exe | shmtool.exe |

Virus Generator (G2) and 50 files from the Next Generation Virus Construction Kit (NGVCK) [32]. For these experiments, the comparison set (i.e., benign files), consists of the 16 randomly selected Cygwin utilities files [12] shown in Table 1. The Cygwin files were chosen since they have served as representative non-virus files in several previous studies, including [20,35].

Another set of experiments was conducted using the metamorphic worms developed in [26]. We denote these worms as MWOR. The worms in [26] were specifically developed to evade statistical opcode-based detection techniques, such as those that rely on HMMs, opcode graphs, or $n$-gram analysis, as discussed in Sect. 2.2.1. To increase the difficulty of statistical detection, the MWOR worms insert dead code selected from benign files. The amount of dead-code is specified as a "padding ratio." For example, a padding ratio of 2 means that there is twice as much dead-code as functional worm code. The instructions used for padding were retrieved from a subset of the benign files that comprise our test set. Consequently, our test set should provide a more difficult challenge than randomly-selected benign files.

In our experiments, we use distinct sets of MWOR files with padding ratios of 0.5, 1.0, 1.5, 2.0, 3.0, and 4.0. For each padding ratio, we conducted a separate experiment using 100 MWOR worms (i.e., a total of 600 MWOR worms). Since the MWOR files are Linux worms, for these experiments our benign set consists of Linux executables. Specifically, we selected the 30 system programs in Table 2 to serve as representative benign files.

## 4.2 Parameters

Recall that the entropy-based distance measure considered here functions in two stages, namely, file segmentation and sequence comparison. In the file segmentation phase, we build an entropy map using a sliding window. We set the window size to 64 bytes for the G2 and NVGCK files, and, due to their larger sizes, 256 bytes for all of the MWOR file sets. The slide size was set to half the window size.

**Table 2** Linux benign files

| | |
|---|---|
| /bin/date | usr/bin/kill |
| /bin/dmesg | usr/bin/killall |
| /bin/grep | usr/bin/last |
| /bin/mknod | usr/bin/ld |
| /bin/mount | usr/bin/namei |
| /bin/rm | usr/bin/nm |
| /bin/sleep | usr/bin/nm-tool |
| /bin/sync | usr/bin/objdump |
| /bin/touch | usr/bin/oclock |
| /usr/bin/as | usr/bin/readelf |
| /usr/bin/at | usr/bin/rpl8 |
| /usr/bin/file | usr/bin/shuf |
| /usr/bin/funzip | usr/bin/size |
| /usr/bin/dig | usr/bin/strip |
| /usr/bin/msgcat | usr/bin/sum |

To analyze the impact of the cost parameters, we calculated the AUC of ROC curves obtained using a range of values for the constants $c_\varepsilon$ and $c_\sigma$ that appear in (6). For $c_\varepsilon$, we used a range from 0 to 1 with 0.05 increments, and for $c_\sigma$, values ranged from 0 to 2 with 0.10 increments. For these experiments, we used the NGVCK file set. Figure 11 gives our results.

Based on these results, we selected

$$c_\varepsilon = 0.4 \quad \text{and} \quad c_\sigma = 1.6.$$

For comparison, in [25], the constants were set to $c_\varepsilon = 0.6$ and $c_\sigma = 1.4$. As can be seen in Fig. 11, such minor differences in the parameters have little impact on the score.

The constant $\tau$ appears in Eqs. (7) and (9). We let $\tau = 0.3$, which implies that the average penalty for a pair of mismatching segments is 0.3. This is the same value for $\tau$ as used in [25].

## 4.3 Results

For the G2 viruses, we obtained the results in Fig. 12, where the red points correspond to the similarity percentages between virus pairs and the virus-benign scores are denoted by the green points. As can be deduced from the results in Fig. 12, the G2 viruses are, from the perspective structural entropy, almost identical to one another—the computed similarity percentages are all above 95 %. In [35] it was found that the G2 viruses had the highest average $n$-gram similarity among the generators tested, so our result here are not surprising. Based on these test cases, structural entropy similarity can be used to detect G2 viruses with no false positives or false negatives.

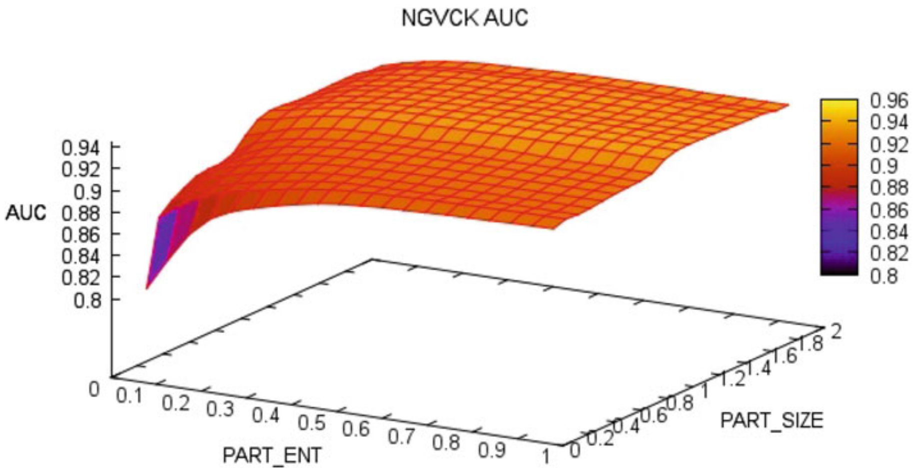


**Fig. 11** AUC for NGVCK with various parameter values

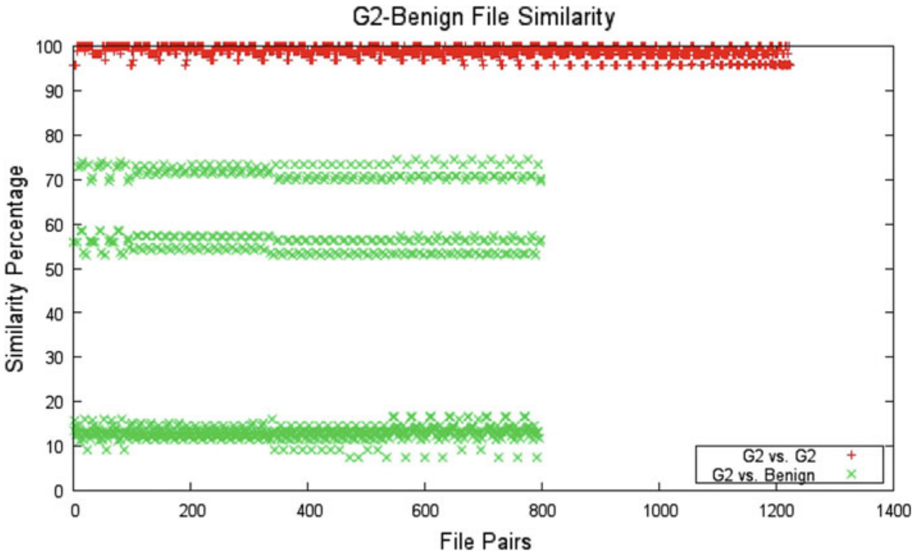


**Fig. 12** G2 similarity

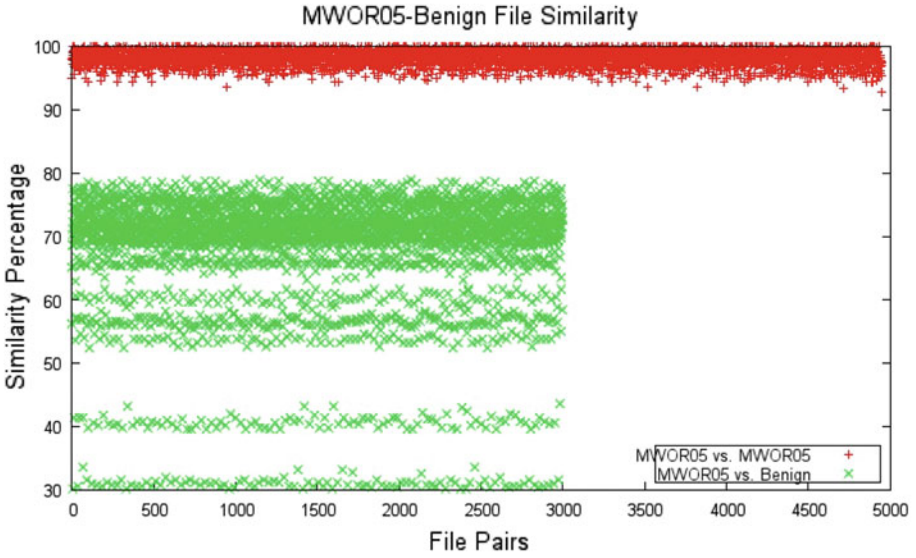


**Fig. 13** MWOR 0.5 similarity

**Table 3** MWOR similarity statistics

| Family | Comparison | Mean | Min | Max |
|---|---|---|---|---|
| MWOR 0.5 | worm vs worm | 98.18 | 92.82 | 100.00 |
| | worm vs benign | 67.47 | 30.19 | 79.02 |
| MWOR 1.0 | Worm vs worm | 97.50 | 93.14 | 100.00 |
| | worm vs benign | 66.63 | 30.50 | 79.61 |
| MWOR 1.5 | worm vs worm | 97.08 | 93.20 | 100.00 |
| | worm vs benign | 68.60 | 24.19 | 78.61 |
| MWOR 2.0 | Worm vs worm | 96.33 | 91.96 | 100.00 |
| | worm vs benign | 68.72 | 26.73 | 78.49 |
| MWOR 2.5 | Worm vs worm | 95.62 | 92.23 | 100.00 |
| | worm vs benign | 68.32 | 24.16 | 77.87 |
| MWOR 3.0 | Worm vs worm | 95.00 | 89.30 | 100.00 |
| | worm vs benign | 67.68 | 26.06 | 77.87 |
| MWOR 4.0 | Worm vs worm | 93.84 | 89.96 | 100.00 |
| | worm vs benign | 65.82 | 27.24 | 78.64 |

Next, we assessed our method against the MWOR families of metamorphic worms. Figure 13 displays the results of our experiments on MWOR worms with a padding ratio of 0.5.

We found that the smallest and largest benign files scored the lowest when compared to the worm files. This is not too surprising, since our score penalizes size differences. Also not surprising is the fact that the subset of the benign files used for padding the worms came closest to the average similarity of the worm files. However, the separation between the distributions shows that we can set a threshold and achieve ideal detection. The statistics for this experiment are summarized in the first entry in Table 3.

We performed the same experiments on MWOR worm families with higher padding ratios; these results are also summarized in Table 3. Although the average similarity between worms decreases as the padding ratio increases, clear separation occurs in every MWOR family, as evidenced by the fact that the minimum "worm vs worm" score exceeds the maximum "worm vs benign" score in every case. Therefore, this structural entropy technique is capable of ideal detection of MWOR worms at all of the padding ratios tested. For the sake of brevity, we have not included score graphs for MWOR padding ratios greater than 0.5. The similarity graphs corresponding to all results in Table 3 can be found in [5].

The results in Table 3 are surprising, since the MWOR worms are not reliably detected using opcode-based HMM analysis, or the opcode-graph technique discussed above [26]. These results indicate that although the opcode statistics differ significantly between MWOR files, the entropy structure remains relatively stable.

Finally, we turn our attention to the NGVCK virus family. These hacker-produced viruses have been shown to be highly metamorphic [35]. We observe that, unlike G2 and MWOR, our NGVCK files differ significantly in size. This is a concern, since the structural entropy score tends to penalize size differences.

Of the 50 NGVCK files in our test set, 34 files are about 4 kB in size, 13 files are about 8 kB, and the remaining three files are 16, 36, and 40 kB. First, we tested the 34 NGVCK files that are 4 kB in size versus the set of Cygwin files in Table 1. The results of this experiment are given in Fig. 14. In this case, the separation is not ideal (in contrast to the G2 and MWOR experiments), since we cannot set a threshold without incurring false positives and/or false negatives.

We repeated the same test on the group of 13 NGVCK files each having size of about 8 kB; Fig. 15 contains the similarity scores. In this case, the separation is not as large as for the G2 or MWOR tests cases, but it is still sufficient for ideal detection.

We then combined the 4 and 8 kB NGVCK files into one experiment and the resulting similarity scores are plotted in

**Fig. 14** NGVCK: 4 kB files

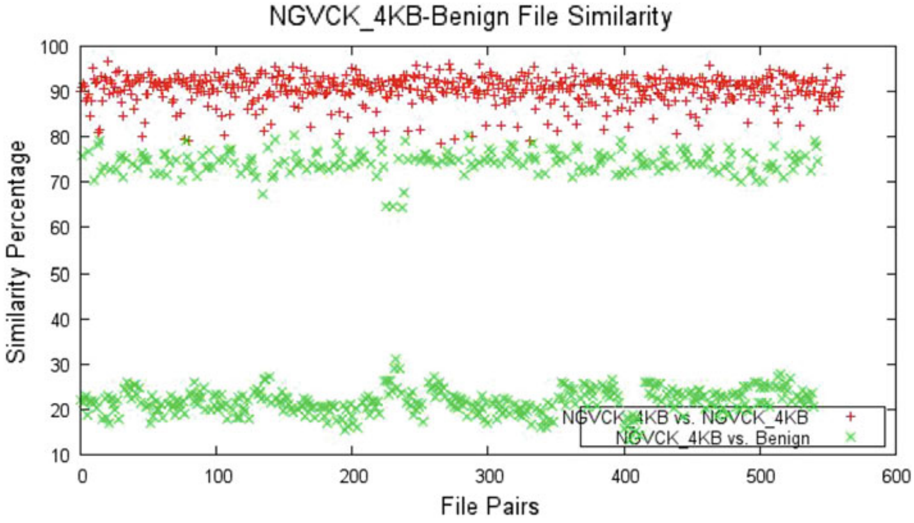


**Fig. 15** NGVCK: 8 kB files

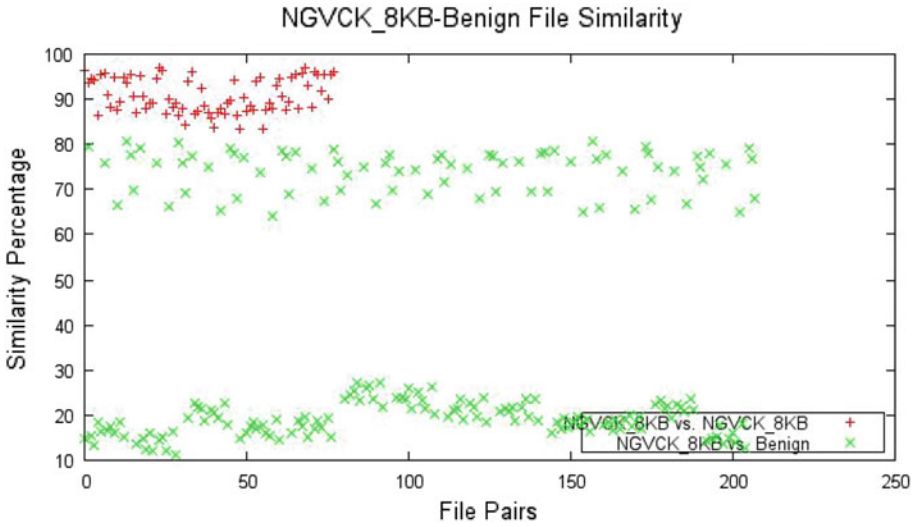
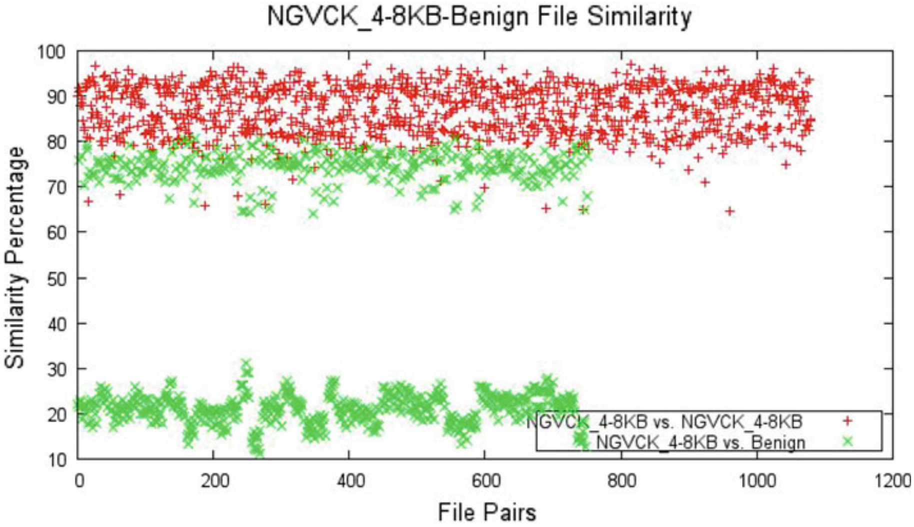


**Fig. 16** NGVCK: 4 kB & 8 kB files



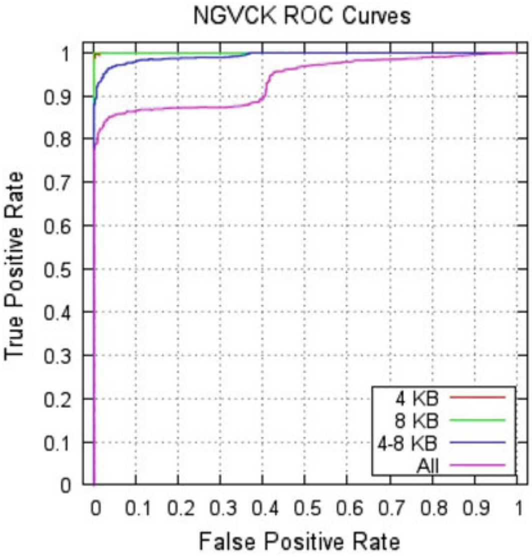


**Fig. 17** NGVCK ROC curves

Fig. 16. In this case, the results are poor. Including the three larger NGVCK files (16, 36, 40 kB) into the set produced even worse results.

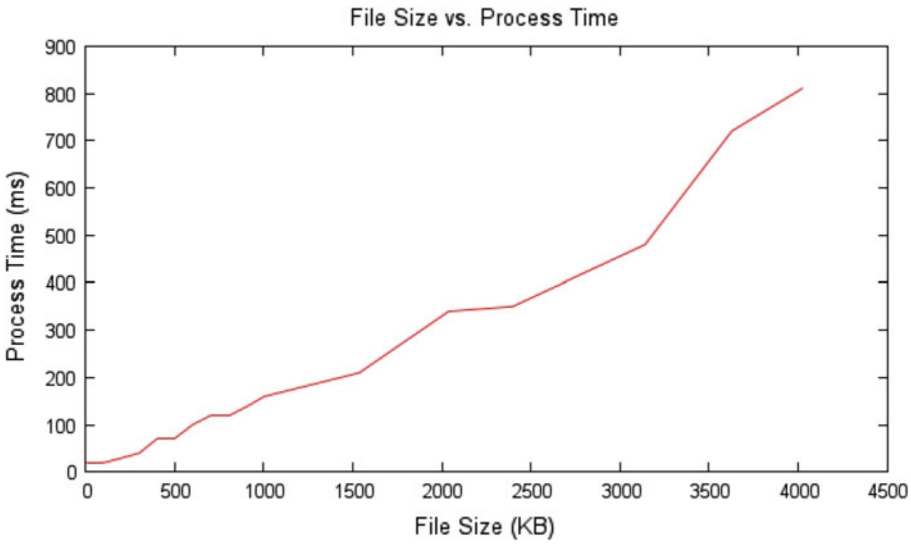
ROC curves for these NGVCK experiments are given in Fig. 17. The corresponding AUC statistics for appear in Table 4.

Finally, to determine the efficiency of this scoring method, we processed a series of files with sizes ranging from 10 to 4 MB. The results are given in Fig. 18, where it can be

**Table 4** AUC of NGVCK

| File sizes | Number of files | AUC |
|---|---|---|
| 4 kB | 34 | 0.99989 |
| 8 kB | 13 | 1.00000 |
| 4 & 8 kB | 47 | 0.99262 |
| All | 50 | 0.93539 |

**Fig. 18** File processing time



observed that the processing time grows linearly with file size. In addition, we see that, in absolute terms, the processing time is about 0.2 s per MB. These timings were obtained on a 64-bit Windows 7 machine with Intel Core i5-2540M processor at 2.60 GHz and 8 GB of RAM. Note that the largest malware file in our experiments is 68 kB, so each of these files required less than 0.02 s to score. The results here indicate that efficiency is a strength of this structural entropy scoring technique.

## 5 Conclusion and future work

The similarity method analyzed in this paper is based on a structural entropy analysis of files. This method applies directly to binary files, and hence no code disassembly or other expensive pre-processing is required.

Our experiments show that structural entropy can be used to classify metamorphic malware of the G2 family. More surprisingly, the experimental evidence shows that we can correctly classify the MWOR metamorphic family, even at high padding ratios. At these high padding ratios, the MWOR family cannot be reliably detectable using the opcode-based analyses in [20,23,26]. Evidently, to evade structural entropy-based detection, a metamorphic generator must morph program structure in a meaningful way, as opposed to simply morphing instructions.

For the NGVCK metamorphic family, our experimental results are more mixed. The structural entropy score depends heavily on segment length and the number of segments. Longer files will tend to produce more segments, so the score is somewhat sensitive to file length. Since the NGVCK files differ significantly in length, this may be the cause of our relative lack of success with these viruses.

Overall, our results indicate that a similarity measure based on structural entropy is a potentially useful—and highly efficient—means for classifying metamorphic malware. At the least, this creates an additional hurdle that virus writers must clear before they can hope to create metamorphic malware that can remain undetected under static analysis.

Future work could include a more detailed analysis of the many parameters that appear in the structural entropy score computation. Research aimed at reducing the score dependence on file length could prove particularly fruitful. In addition, other cost functions could be considered. A study of the relationship between structural entropy and individual morphing strategies could be interesting and insightful.

As in [25], the entropy score analyzed in this paper uses an edit distance to compare entropy transform sequences. Other string comparison measures could be used; examples include $n$-gram scores, HMM-based analysis, profile hidden Markov model (PHMM) techniques [3], and the "simple substitution" distance considered in [23]. Finally, other types of structure-based scores could be tested. For example, the Normalized Compression Distance (NCD) [10] could form the basis for a structural scoring technique analogous to the score analyzed in this paper.

## References

1. Addison, P.: The Illustrated Wavelet Transform Handbook: Introductory Theory and Applications in Science. Engineering, Medicine and Finance. Taylor and Francis Group, New York (2002)
2. Apostolico, A., Galil, Z.: Pattern Matching Algorithms. Oxford University Press, Oxford (1997)
3. Attaluri, S., McGhee, S., Stamp, M.: Profile hidden Markov models and metamorphic virus detection. J. Comput. Virol. **5**(2), 151–169 (2009)

4. Aycock, J.: Computer Viruses and Malware. Springer, New York (2006)
5. Baysa, D.: Structural entropy and metamorphic malware. Master's report, Department of Computer Science, San Jose State University. http://scholarworks.sjsu.edu/etd_projects/283/ (2012)
6. Borda, M.: Fundamentals in Information Theory and Coding. Springer, New York (2011)
7. Borello, J., Me, L.: Code obfuscation techniques for metamorphic viruses. J. Comput. Virol. **4**(3), 30–40 (2008)
8. Bradley, A.P.: The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern Recognit. **30**, 1145–1159 (1997)
9. Burford, S.: Reverse engineering Linux ELF binaries on the x86 platform. http://www.linuxsa.org.au/meetings/reveng-0.2.pdf (2002)
10. Cilibrasi, R., Vitányi, P.M.B.: Clustering by compression. IEEE Trans. Inform. Theory **51**(4), 1523–1545 (2005)
11. Collberg, C., Thomborson, C., Low, C.: A taxonomy of obfuscating transformations. Technical Report #118. The University of Auckland (1997)
12. Cygwin, Cygwin utility files. http://www.cygwin.com/. Accessed Dec 2012
13. Islita, M.: Levenshtein edit distance. http://www.miislita.com/searchito/levenshtein-edit-distance.html (2006)
14. Karmeshu.: Entropy Measures, Maximum Entropy Principle and Emerging Applications. Springer, New York (2003)
15. The Mental Driller, Metamorphism in practice or "How I made MetaPHOR and what I've learnt". http://biblio.l0t3k.net/magazine/en/29a/ (2002)
16. Patel, M.: Similarity tests for metamorphic virus detection, Master's report. Department of Computer Science, San Jose State University. http://scholarworks.sjsu.edu/etd_projects/175/ (2011)
17. Pietrek, M.: Peering inside the PE: a tour of the Win32 portable executable file format. MSDN Magazine. http://msdn.microsoft.com/en-us/magazine/ms809762.aspx (1994)
18. Radhakrishnan, D.: Approximate disassembly, Master's report. Department of Computer Science, San Jose State University. http://scholarworks.sjsu.edu/etd_projects/155/ (2010)
19. Robinson, S.: Expert. NET 1.1 Programming. Apress, New York (2004)
20. Runwal, N., Low, R., Stamp, M.: Opcode graph similarity and metamorphic detection. J. Comput Virol. **8**(1–2), 37–52 (2012)
21. SearchSecurity, Metamorphic and polymorphic malware. http://searchsecurity.techtarget.com/definition/metamorphic-and-polymorphic-malware (2010)
22. Shah, A.: Approximate disassembly using dynamic programming, Master's report. Department of Computer Science, San Jose State University. http://scholarworks.sjsu.edu/etd_projects/8/ (2010)
23. Shanmugam, G., Low, R., Stamp, M.: Simple substitution distance and metamorphic detection. J. Comput. Virol. (to appear)
24. Snakebyte, Next Generation Virus Construction Kit (NGVCK). Open Malware http://www.offensivecomputing.net/ (2000)
25. Sorokin, I.: Comparing files using structural entropy. J. Comput. Virol. **7**(4), 259–265 (2011)
26. Sridhara, S.M., Stamp, M.: Metamorphic worm that carries its own morphing engine. J. Comput. Virol. (2012) (online first$^{\mathrm{TM}}$)
27. Stamp, M.: A revealing introduction to hidden Markov models. http://cs.sjsu.edu/~stamp/RUA/HMM.pdf (2012)
28. Struzik, Z., Siebes, A.: The Haar wavelet transform in the time series similarity paradigm. In: Proceedings of the Third European Conference on Principles of Data Mining and Knowledge Discovery (PKDD '99). Springer, London. http://dl.acm.org/citation.cfm?id=669368 (1999)
29. Symantec, Viruses, worms, and trojans. http://service1.symantec.com/support/nav.nsf/docid/1999041209131106 (2011)
30. Van Fleet, P.: The discrete haar wavelet transformation. Joint Mathematical Meetings, Center for Applied Mathematics, University of St. Thomas. http://cam.mathlab.stthomas.edu/wavelets/pdffiles/NewOrleans07/HaarTransform.pdf (2007)
31. Verschuuren, G.: Excel 2007 for Scientists and Engineers. Holy Macro! Books (2008)
32. Virus files, Department of Computer Science, San Jose State University. http://cs.sjsu.edu/~stamp/viruses/ (2012)
33. Vuorenmaa, T.: The discrete wavelet transform with financial time series applications. Seminar on Learning Systems, University of Helsinki. http://www.rni.helsinki.fi/teaching/sols/TV_RNI.pdf (2003)
34. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM (JACM) **21**(1), 168–173 (1974)
35. Wong, W., Stamp, M.: Hunting for metamorphic engines. J. Comput. Virol. **2**(3), 211–229 (2006)
36. You, I., Yim, K.: Malware obfuscation techniques: a brief survey. In: International Conference on Broadband, Wireless Computing. Communication and Applications (BWCCA), pp. 297–300 (2010)