Accepted to *European Research Journal of Computer Virology* (2005, to appear)

Malware Phylogeny Generation using Permutations of Code

Md. Enamul Karim, Andrew Walenstein, Arun Lakhotia
Center for Advanced Computer Studies
University of Louisiana at Lafayette
{mek,arun}@cacs.louisiana.edu,walenste@ieee.org

Laxmi Parida
IBM T. J. Watson Research Center
parida@us.ibm.com

Abstract

Malicious programs, such as viruses and worms, are frequently related to previous programs through evolutionary relationships. Discovering those relationships and constructing a phylogeny model is expected to be helpful for analyzing new malware and for establishing a principled naming scheme. Matching permutations of code may help build better models in cases where malware evolution does not keep things in the same order. We describe method for constructing phylogeny models that uses features called n-perms to match possibly permuted code. An experiment was performed to compare the relative effectiveness of vector similarity measures using n-perms and n-grams when comparing permuted variants of programs. The similarity measures using n-perms maintained a greater separation between the similarity scores of permuted families of specimens versus unrelated specimens. A subsequent study using a tree generated through n-perms suggests that phylogeny models based on n-perms may help forensic analysts investigate new specimens, and assist in reconciling malware naming inconsistencies.

1 Introduction

Systematically reusing code has been an elusive target for software development practice since the term "software engineering" was first coined, yet in certain respects reuse may be an everyday practice for malware authors. The term "malware" is in common use as the generic name for malicious code of all sorts, including viruses, trojans, worms, and spyware. Malware authors use generators, incorporate libraries, and borrow code from others. There exists a robust network for exchange, and some malware authors take time to read and understand prior approaches [1]. Malware also frequently evolves due to rapid modify-and-release cycles, creating numerous strains of a common form. The result of this reuse is a tangled network of derivation relationships between malicious programs.

In biology such a network is called a "phylogeny"; an important problem in bioinformatics is automatically generating meaningful phylogeny models based on information in nucleotide, protein, or gene sequences [2]. Generating malware phylogeny models using techniques similar to those used in bioinformatics may assist forensic malware analysts. The models could provide clues for the analyst, particularly in terms of understanding how new specimens relate to those previously seen. Phylogeny models could also serve as a principled basis for naming malware. Despite a 1991 agreement on an overall naming scheme and several papers proposing new schemes, malware naming continues to be a problem in practice [3, 4].

The question remains, though, as to how useful phylogeny models can be built from studying the bodies of malicious programs. The method used to generate the models should be able to account for the types of changes that actually occur in malware evolution. The focus of this paper is building phylogeny models taking into account the fact that programs may be evolved through code rearrangements, including instruction or block reordering. Such reorderings might be a result of malware author changing the behavior and organization of the code, or they might be a result of metamorphic programs modifying their own code [5]. We therefore aim to examine the suitability of methods that can reconstruct derivation relationships in the presence of such perturbations.

Specifically, two feature extraction techniques are examined: n-grams and fixed-length permutations we call "n-perms". These are being investigated because they can find matches of similar segments in programs, and with such matches it is possible to generate models of likely derivation relationships through the analysis of similarity scores. An experiment was performed to gauge their relative abilities in this regard. The experiment involved comparing the relative similarity scores for artificially constructed permuted worms and unrelated worms. n-perms maintained a greater separation between those scores for a range of values of n. A study was then performed to determine how well the names generated by various anti-virus scanners aligned with the trees we generated. The results suggest ways of using such phylogeny models in specimen identification and in reconciling naming inconsistencies.

Malware classification and phylogeny models are described in Section 2, and past approaches are reviewed. The approaches we consider for permutation-based cases are described in Section 3, including an analysis of their expected merits. The experiment and study are described in Sections 4 and 5, respectively, and include a discussion of the results.

2 Malware classification and phylogeny models

Phylogenetic systematics is the study of how organisms relate and can be ordered; a phylogeny is the evolutionary history or relationships between organisms. Molecular phylogenetics takes the approach of studying organism relationships by inferring derivation relationships from the information contained in the organisms themselves. The goal has been described as "to infer process from pattern" [6].

Software, too, has its own analogous field to molecular phylogenetics in which software artifacts are examined and compared in an effort to reconstruct their evolution history (see, e.g., Godfrey & Tu [7]). Creating phylogeny models of malware is a specialized area within this field. Generally speaking, the approach taken is to analyze programs or program components for commonalities and differences, and then from these to infer derivation relationships or other aspects of evolution. It is important to note that, in forensic malware analysis, a phylogeny model need not correspond to the "true" phylogeny in order to be useful. For instance, incidental similarities not related through code derivation may still be helpful in analysis. For this reason we shall take care to avoid conflating the terms "phylogeny" and "phylogeny model".

For context we shall review both classification and phylogeny model generation for malware since the problem of generating phylogeny models shares common ground with classification in terms of needing ways to compare programs. Whether by design or by accident, the published malware comparison methods we are aware of have taken approaches that reduce reliance on sequencing information. We review sequence comparisons for malware analysis and then separate applications of alternative methods into binary classification and phylogeny generation.

2.1 Sequence comparison and alignment

Methods to compare or align sequences and strings are important tools for molecular phylogenetics. Techniques such as suffix trees, edit distance models, and multiple alignment algorithms are staples for comparing genetic information [2]. These sorts of techniques have been applied to benign computer programs as well, including program texts at the source level [8], machine level [9], and in-between [10]. Commercial anti-virus (AV) scanners are also known to use some types of sequence matching in order to classify programs into fine-grained categories [11] (`Win32.Evol.A`, `Win32.Netsky.B`, etc.). The exact techniques used are not known to us, but we do not believe they are substantially similar to suffix trees, edit distances, and the like. Although those methods are popular in bioinformatics they appear not to be widely used for the purpose of classification or phylogeny model generation for malware.

On the one hand, sequence-based methods may work well for phylogeny model generation when sufficient numbers of sequences are preserved during evolution. Consider, for instance, the two worms named `I-Worm.Lohack.`{`a`,`b`} (the notation `X.`{`y`,`z`} is a shorthand for the sequence `X.y`,`X.z`) which we obtained from VX Heavens [12], the widely available malware collection. Both worms are 40,960 bytes long and differ on only some 700 bytes (less than 2%). While these two particular programs share large blocks of common bytes, it cannot be assumed that all related malware will. Nonetheless, if, in practice, related malware families maintain sufficient numbers of common sequences then phylogeny models generated based on the sequence commonalities may be satisfactory.

On the other hand, many sequence-based methods may not work well for malware if it has evolved through significant code shuffling and interleaving. Signature-based AV scanners have been known to identify malware by searching for particular sequences [11]. This fact is likely to motivate malware authors to destroy easily identifiable sequences between releases so that they can avoid detection. The ability of AV scanners to detect these sequences is likely to have prompted the emergence of polymorphic and metamorphic malware [5]. Some polymorphic and metamorphic malware—such as `Win32.ZPerm` [13] and `WM/Shuffle.A` [14]—permute their code during replication. Recognizing the self-constructed derivatives will be difficult if these permutations are not accounted for. We feel it is reasonable to expect that permutation and reordering will continue to be one of the methods in the malware authors' toolbox.

2.2 Binary classification

A common technique in text processing is to use n-grams as features for search, comparison, and machine learning. An n-gram is simply a string of n characters occurring in sequence. In using n-grams for malware analysis, the programs are broken down into sequences of n characters which, depending upon the granularity desired and definitions used, could be raw bytes, assembly statements, source lexemes, lines, and so on. As n decreases towards 1, the significance of sequence information is reduced.

Perhaps the earliest application of n-grams for malware analysis was by the research group at IBM, who investigated method for automatically recognizing boot sector viruses [15–18] using artificial neural networks, as well as extracting signatures for Win32 viruses [19]. As features they used byte n-grams with n in the range of 1 to 8, depending upon the particular method or application they used. Kolter & Maloof [20] used a pilot study to determine a suitable value of n, and settled on using byte 4-grams. Values of n ranging from 1 to 10 were used by Abou-Assaleh *et al.* [21], who used a technique for reducing the number of features if they grew past some bound. It is not clear at this point in what conditions the various possible values of n work best. Kephart and Arnold [17] used a range of n to build recognition terms of different lengths, which suggests that they found a fixed n to be insufficient. However Abou-Assaleh *et al.* recounted that in some applications trigrams are able to capture some larger sequence information implicit in the trigrams.

In addition to n-grams, other features have been used to generate heuristic classifiers. Schultz *et al.* [22] compared the performance of Naive Bayes classifiers applied to several different malware features, including the list of dynamically linked libraries referenced by the executable, embedded text strings, and 8-byte chunks. Note that the chunks used by Schultz *et al.* were not overlapping and thus cannot be counted as an application of n-grams.

This collection of past research has demonstrated promising abilities for automatically generating heuristic classifiers that can perform the binary classification decision of separating malicious programs from benign ones. However the record does not indicate how well these techniques would do at finer-grained classifications needed for specimen identification (i.e., naming). While Kolter *et al.* reported accurate classification, we have concerns as to whether their experiences will generalize if packed or encrypted versions of both malicious and benign programs are used in training or test data. A packer such as the `UPX` packer [23] will compress valid executables into a compressed segment and a short segment containing standard unpacking code. Both benign and malicious executables will have similar unpacking codes, but will differ on the compressed portions. The compressed portions will have high entropy and, in fact, tend towards resembling random data. Any n-gram matches of bytes from such sections are likely to be accidental. Thus any comparisons or classification decisions made on the basis of n-gram matches are likely to be based primarily on matches to the decompressing segment, which will be common to both benign and malicious code, and will fail to properly distinguish the two classes.

2.3 Malware phylogeny model generation

Approaches for generating phylogeny models can be differentiated according to (a) the way that program features are selected, (b) the feature comparison methods or measures employed, (c) the type or structure of the models generated, and (d) the algorithms used to generate the models. At the time of this writing the existing results appear in three publications from Goldberg *et al.* [24], Erdélyi and Carrera [25], and Wehner [26].

Regarding features and comparison methods, Goldberg *et al.* used sequences of 20 bytes and used Boolean occurrence matching in their methods for comparing specimens. They used suffix trees to construct 20-gram occurrence counts. Goldberg *et al.* present several phylogeny model generation methods; they reasoned that 20-byte sequences are large enough that each distinct sequence will have been "invented" exactly once, so in one of their generation methods it is assumed that if any sequence is found in more than one program it is safe to infer that one was derived from the other. Erdélyi *et al.* used statically-extracted call graphs as features, and applied a heuristic graph comparison algorithm to measure distances between specimens. The argument made in favor of this method is that it compares specimens, to a degree, on the basis of behavior rather than, say, data or raw bytes. As a result, this technique may even be suitable for comparing certain metamorphic forms of malware, assuming the calling structures can be extracted well enough. Wehner uses a distance measure called Normalized Compression Distance (NCD) which she approximated using the `bzip2` compressor, a block compressor utilizing Burrows-Wheeler block sorting and Huffman coding.

In terms of model structure and generation algorithm, both Wehner and Erdélyi *et al.* use unspecified clusterers to generate purely hierarchical X-trees. We can expect that such tree outputs will at least occasionally produce results of questionable value in cases where multiple inheritance relationships are present. Goldberg *et al.*, in contrast, extract directed acyclic graphs which they called "phyloDAGs". PhyloDAGs can represent multiple inheritance relationships which, from source code comments we have found in released malware, we know to exist.

3 Permutation-based methods for feature extraction

The focus of this paper is on generating phylogeny models for malware that may have evolved, in part, through permutations of code. These permutations could include instruction reordering, block reordering, or subroutine reordering. In such situations the reordering can make sequence-sensitive techniques produce undesirable results if they report similarity scores that are too low for reordered variants or descendants.

For instance, consider the two programs P_1 and P_2 of Figure 1, in which P_2 is derived from P_1 by swap edits. In the figure, distinct letters signify distinct characters from whatever alphabet is being used. P_2 differs from P_1 by a block swap (1-4 swap 9-12) and by two character swaps (2 swap 3 and 9 swap 10 on P_1). The block swaps are highlighted using underlines and the character swaps using overlines. If each of these characters is a source line, the standard `diff` tool from the GNU TextUtils package finds only the *efgh* substring in common. This is because its differencing algorithm is based on LCS and an edit model that does not consider block moves or swaps. There do exist string edit distance models that account for block moves [27]. While these may indeed be highly suitable for malware analysis, they are beyond the scope of this paper.

$$P_1 = ABCDefghIJKL \qquad \text{and} \qquad P_2 = \underline{\overline{JI}\,KL}efgh\underline{A\,\overline{CB}\,D}$$

Figure 1: Pair of sequences related by swap modifications

In this section we consider two different feature types—n-grams, and n-perms—as bases for comparing programs for the purpose of building phylogeny models. Both of them permit permuted sequence matching based on document comparison techniques employing feature occurrence vector similarity measures. Such techniques match common features regardless of their positions within the original documents. From the similarity scores evolutionary relations can be inferred. The feature extraction methods are outlined, their expected strengths and weaknesses are discussed, and methods are outlined for using them in phylogeny model generation.

3.1 n-grams

n-grams, already been introduced above, are widely used in a variety of applications. With bigrams ($n = 2$), the two programs from Figure 1 have four features in common covering six of 12 characters. Thus several matches occur which may be meaningful for evolution reconstruction.

n-grams might be suboptimal for matching permuted sequences in cases where n does not correspond to the size of the important sequential features. For instance, for $n = 4$ the only feature in common between P_1 and P_2 is the string *efgh*, meaning the two permuted subsequences are missed. Another shortcoming of n-grams could potentially be encountered when n is too small to account for the significance of large sequences. For instance, consider a case when the sequence *aaaaaaabbbbbbbb* is an extraordinarily rare sequence shared by two programs, but the bigrams *aa*, *bb* and *ab* are frequent, and match multiple times. With bigrams, the significance of the rare but shared sequence may be lost. Small n-grams may also find misleading matches between unrelated programs simply because the likelihood of an incidental match is expected to increase as n decreases toward 1. Goldberg *et al.* [24] justified one of their uses of 20-grams, in fact, because of the assumed rarity (and hence significance) of long sequences found to be shared. Nevertheless, there is a computational advantage to selecting small values of n since as n grows the numbers of potential features grows rapidly. For this reason many applications of n-grams choose either bigrams or trigrams ($n = 3$), or apply various feature pruning heuristics.

3.2 n-perms

We define a variation on n-grams called "n-perms". For any sequence of n characters that can be taken to be an n-gram, an n-perm represents every possible permutation of that sequence. Thus n-perms are identical to n-grams except that the order of characters within the n-perm are irrelevant for matching purposes. For example, in $abcab$ there are three 3-grams, abc, bca and cab each with one occurrence. However it has only one 3-perm: abc, with three occurrences. The key idea behind n-perms is that they add another level of sequence invariance to the feature vector approach. If applied to P_1 and P_2 of Figure 1, each of the three permuted blocks are matched with 4-perms, covering the full string; only one 4-gram matches, covering just four of 12 characters. With 2-perms there are six matches covering 10 of 12 characters, whereas 2-grams generate only four matches, covering six of 12 characters.

We expect n-perms to be more tolerant of character-level reorderings (i.e., within a span less than n) than n-grams. In addition, for a given string, the number of possible features is expected to be less for n-perms than for n-grams since some of the sequences distinguished by n-grams will compare as indistinct for n-perms. These differences between n-perms and n-grams may prove advantageous in terms of reducing the number of features that need to be considered, and in terms of increasing match quality for permuted code. However n-perms may also be more "noisy" for a given n than n-grams since unrelated permutations may match. The noise might be controlled by choosing larger values of n, however block moves of smaller sizes may no longer be caught (no 5-perms match for P_1 and P_2 of Figure 1, for example), and the number of features can be expected to rise. As with n-grams, we expect the optimal selection of n may be dependent on the input. In the most general case, no single value of n will catch all permuted commonalities since they may occur at multiple granularities.

3.3 Phylogeny generation using vector similarity

Both n-grams and n-perms can be used as features to match on, and can be utilized to create similarity measures based on vectors of feature occurrences. These vectors, which may be weighted or scaled, are taken to represent the programs, one vector per program. Such feature vector-based methods do not match based on the locations of the features within the programs, and so can detect permutations as being close matches.

In order to compare the relative merits of the extraction methods we implemented families of phylogeny model generators based on these techniques. There are unendingly many ways of constructing distance or similarity measures for these feature vectors [28], and each could be potentially applied to any of the these extraction methods. There are also any number of different heuristics that could be tried for pruning the input space. To keep our investigation tractable, we performed no feature pruning and stuck to using common similarity measures and clustering methods. Each member of the family is implemented as a loosely-coupled collection of programs that execute in a pipeline architecture as follows:

tokenizer $\Longrightarrow$ feature occurrence matrix extractor $\Longrightarrow$ similarity metric calculator $\Longrightarrow$ clusterer

Supplying the appropriate tokenizer allows different alphabets to be used, including bytes, words, lines, etc. Because our tokenizer is perhaps the only unusual component, it is described last.

Feature occurrence matrix extractor. We created two programs—one each for n-grams and n-perms—which take n as a parameter, extract features, and then construct a feature occurrence count matrix such that each entry i, j records the number of times feature i occurs in program j.

Similarity metric calculator. This takes as input a feature occurrence matrix and constructs a symmetric similarity matrix in which entry i, j records the calculated similarity between programs i and j. We tried several similarity metrics in pilot studies before settling on one that appeared to work well. It implements TFxIDF weighting and cosine similarity (see, e.g., Zobel *et al.* [28]), a combination we shall refer to as TFxIDF/cosine. TFxIDF weights the features such that features common to many programs are scaled down in importance, and features common within any given program are scaled up in importance. The weighted matrix is constructed by calculating $tf_{i,j} \log(N/df_i)$, where $tf_{i,j}$ is the count of the number of times feature i occurs in the jth program, and df_i is the count of the number of programs that feature i occurs in.

Clusterer. We used CLUTO [29] to perform clustering. We selected its agglomerative clustering functionality to build dendograms, and used the UPGMA clustering criterion function, which is commonly used in biological phylogeny model generation [29]. Although the resulting phylogeny models cannot capture multiple inheritances, it is a well-known technique and was a suitable baseline from which to start our explorations.

Tokenizer. We built a filter that transforms input programs into sequences of assembly opcodes, which then can be fed as input to the feature extractor. This filter was implemented after observing that many members within a family of worms would vary according to data offsets or data strings, or by inclusion of junk bytes. We wished to select features which were closely related to program behavior, yet were relatively immune to minor changes in offset or data. Similar motivations drove Kolter *et al.* [20] to try classifying programs based, in part, on the dynamic link libraries referenced by the programs. In our experience, transforming the input to abstracted assembly also helpfully reduced the size of the input considerably, making the similarity computations and clustering substantially cheaper.

One drawback of using opcode sequences is that the programs need to be unpacked or unencrypted, and we must be able to disassemble them. This can be a hardship since many malicious programs are intentionally written to make this difficult. Nevertheless, we considered it to be critical to use unpacked code for the reasons outlined in Section 2.2. Figure 2 illustrates the point of using specimens with root name

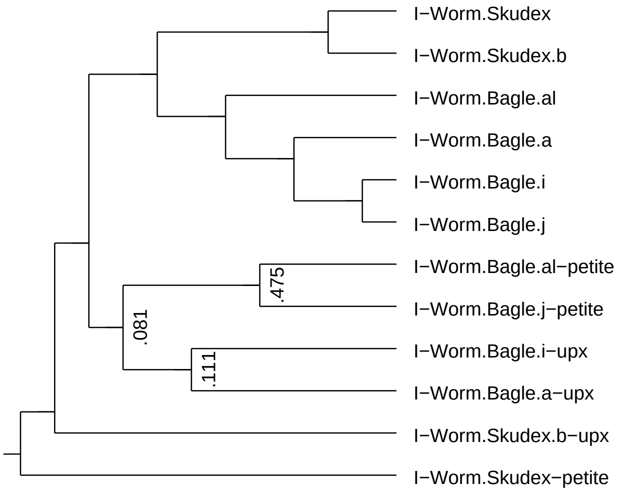


Figure 2: Phylogeny model showing clustering tendency of packed executables

of "Bagle" and "Skudex" from the VX Heavens [12] collection, which shows results using both unpacked versions and versions packed by ourselves. The figure was created using bigrams over bytes. The packed versions were created using one of the `UPX` [23] packer or the `Petite`[1] packer. The packed versions have a "-upx" or "-petite" suffix to distinguish them. The packed versions all either cluster together or are in subtrees completely different from their unpacked versions. Moreover, many of the packed samples are arranged according to the packer type, suggesting that the clustering for the packed versions is based primarily on the unpacker segment. To illustrate this point, the similarity of the two `Petite`-packed `Bagle`s was measured at .475, while the average similarity of the four packed `Bagle`s as a whole was only .081.

While Figure 2 was created using byte bigrams to illustrate a point, in the studies we report below, we use only the opcode sequences as input. In our pilot studies we frequently found that both bytes and opcode sequences produced similar looking trees, but a closer investigation revealed that bytes were less reliable for this purpose.

4 Experiment

An experiment was conducted to explore the relative strengths and weaknesses of the n-gram and n-perm approaches. The hypothesis tested is that n-perms will outperform n-grams in terms of being able to find similarities in programs that differ due to permutation operations. To test this, similarity scores for both permuted and unrelated specimens were collected and compared for several subsets of a test sample.

4.1 Design

A two-phased approach was used, with the first phase being used to select parameters for the second phase.

The first phase tests whether n-perms generate better similarity scores than n-grams for two classes of malware specimen pairs: (1) specimens related by permutation mutations, and (2) unrelated specimens. The first class are called permuted pairs ($\mathcal{P}$) and the second are called unrelated pairs ($\mathcal{U}$). The specific hypothesis was that regardless of the value of n, (1) n-perms would generate higher similarity scores than n-grams for $\mathcal{P}$ pairs, and (2) that n-perms would generate a wider gap in similarity scores between $\mathcal{P}$ pairs and $\mathcal{U}$ pairs. We employed three measures to test this: *Diff*, *Gap*$(max, gram)$, and *Gap*$(max, perm)$, defined as follows. Let $Sim(\mathcal{P}, n, perm)$ and $Sim(\mathcal{U}, n, perm)$ be the averaged n-perm based similarity scores for

[1] `Petite 2.3`, obtained from `http://un4seen.com/petite/` on 4 Mar, 2005.

pairs in $\mathcal{P}$ and $\mathcal{U}$, respectively, for a given n. These are calculated for a sample of m programs by constructing the similarity matrix for the whole sample and then averaging the similarity scores of selected pairs of specimens from the sample: those for unrelated pairs ($\mathcal{U}$) and pairs related by permutation mutations ($\mathcal{P}$). The analogous definitions for n-grams are assumed. Then, for a given n, $\mathit{Diff}(n) = Sim(\mathcal{P}, n, perm) - Sim(\mathcal{P}, n, gram)$ and $\mathit{Gap}(n, perm) = Sim(\mathcal{P}, n, perm) - Sim(\mathcal{U}, n, perm)$. $\mathit{Gap}(max, perm)$ is the maximum of the $\mathit{Gap}(i, perm)$ scores. $\mathit{Gap}(n, gram)$ and $\mathit{Gap}(max, gram)$ are defined analogously. Then the null hypothesis is encoded as $(\exists n.\mathit{Diff}(n) \leq 0) \vee \mathit{Gap}(max, perm) \leq \mathit{Gap}(max, gram)$. We would reject the null hypothesis if $(\forall n.\mathit{Diff}(n) > 0) \wedge \mathit{Gap}(max, perm) > \mathit{Gap}(max, gram)$. The independent variable is n and the dependent variables are *Diff* and *Gap*.

The second phase tests whether the separation holds regardless of the number of unrelated specimens in the sample set. To make the test as fair as possible, the results from the first phase are consulted to determine the values of n and m for which the *Gap* value is greatest for n-grams and m-perms, respectively. This gives us the values for which the similarity functions can reasonably be assumed to operate optimally for the sample set. The independent variable is the sample set size s, and the dependent variable is the $\mathit{Diff}(s)$ The null hypothesis is $\exists s.\mathit{Diff}(s) < 0$. We reject it if no better gaps are found for n-grams, i.e., if $\forall s.\mathit{Gap}(s, perm) \geq \mathit{Gap}(s, gram)$.

4.2 Subject dataset and protocol

Nine benign Windows executables were collected from either the Windows XP System or System32 folders from one of our test machines. 141 Windows-based worms were collected, 6 of which were collected from infected mail arriving at our departmental mail server, and the rest from the VX Heavens archive. We selected only those worms we could successfully unpack and disassemble using IDA Pro.[2] Packed versions of three of the benign programs and 14 of the malicious programs were added by packing them with the `UPX` [23][3]. We discarded all executables and kept only the disassemblies of all 167 specimens.

Three of the disassembled, malicious specimens were selected for hand editing. These were named `I-Worm.Bagle.`$\{$`i,j,s`$\}$. Each of these were permuted many times in an ad hoc manner such that the original semantics were preserved. The permutations involved reordering instructions, rearranging basic blocks, and changing the order of appearance of subroutines. The intent was to mimic certain types of permutations that can occur in malware [14]. Including the three permuted variations, the entire sample had 170 specimens in it.

From this collection six subsets were constructed for use as different samples for use in the second part of the experiment . These were constructed by creating a base set and then adding members to it for subsequent sets. The incremental additions are as follows:

	Increment added to subsample	
Subsample	Size	Content
6	6	three hand-modified variations and their original specimens
12	6	three pairs of dissimilar malware specimens
21	9	benign executables
38	17	packed executables, 3 of them benign
76	38	worms
170	94	additional worms

The intent of these selections is to provide increasingly crowded input space so that we could examine the sensitivity of the similarity scores to the presence of both related and unrelated malware specimens. The sample sizes were chosen to reduce the number of samples that would be required if sampling of the subset size was done in a linear fashion. The quantities approximate a logarithmic division (each sample size being roughly double the previous sample). Since only three $\mathcal{P}$ pairs (programs related by permutation mutations) were available, we chose three $\mathcal{U}$ pairs (pairs of unrelated programs) to use in the experiment. These

[2] IDA Pro, version 4.6.0.785, from DataRescue, Inc., `datarescue.com`.

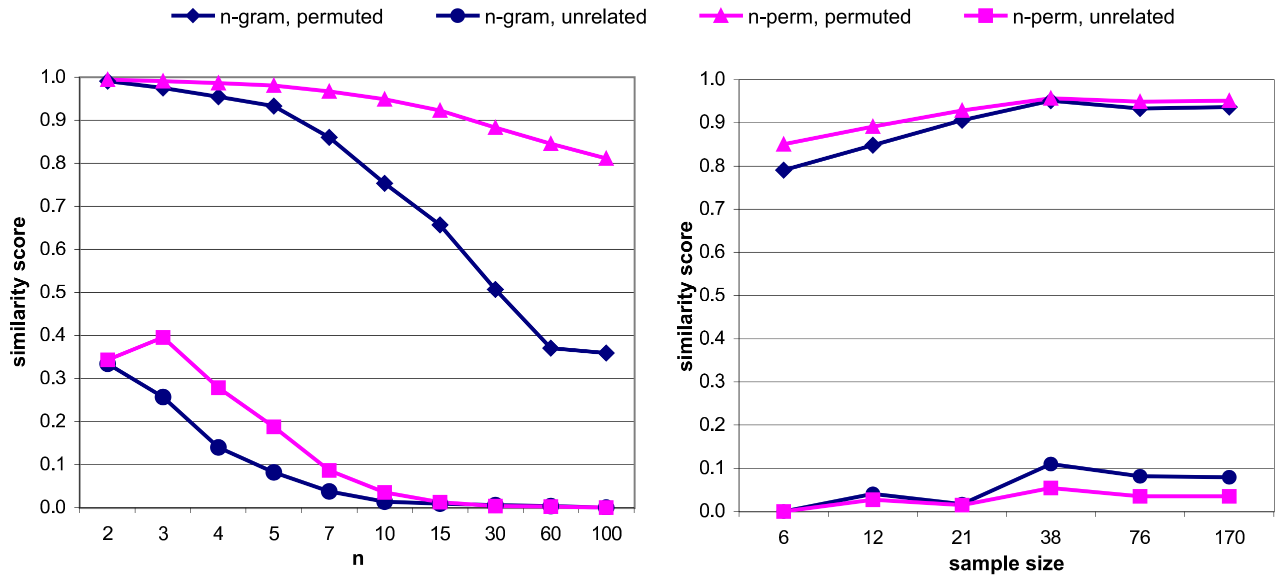
[3] Ultimate Packer for eXecutables, version 1.25, obtained from `upx.sourceforge.net` on 3 Mar, 2005.

were selected from the full set and are included in all but one subsample. They were $<$ `Klez.a`, `Bagle.a` $>$, $<$ `Hermes.a`, `Netsky.x` $>$, and $<$ `Mydoom.g`, `Recory.b` $>$ (all names prefixed with `I-Worm.`).

We employed the apparatus described in Section 3.3 to tokenize the programs into opcode sequences, construct feature matrices, and then calculate similarities. For the first phase we chose the sample subset with 77 specimens in it, and used values of $n = 2, 3, 4, 5, 7, 10, 15, 30, 60, 100$ which, again, approximates a logarithmic scale in sampling n.

4.3 Results

Figure 3(a) and (b), respectively, show the results of the two phases using the sample data. The maximum gaps from the first phase occur at $n = 5$ (.8513) for n-grams and $n = 10$ (.9138) for n-perms. At no value of n is the averaged similarity score for the permuted variants smaller for n-perms than it is for n-grams. According to our criteria we reject both null hypotheses.



(a)

(b)

Figure 3: Averaged similarity scores varying by n and sample size

4.4 Discussion

This limited experiment does not allow firm conclusions to be drawn about how well n-perms will fare in general for phylogeny model generation, as the sample sets are limited, and the experiment did not test the resulting models. Unfortunately, we did not have access to expert-generated phylogenies of malware specimens with verified names. It might be difficult to find one of these with the appropriate permuted specimens. Without such a testbed, this investigation into the similarity score gap was performed as a second alternative. The results suggest that n-perms will be able to catch instances of permuted operations within malware without generating disastrously many false positives. The top two curves of Figure 3(a) suggest that as the value of n rises, n-grams quickly deteriorate in their ability find similarities in permuted malware, whereas n-perm similarity decreases much more slowly on our data. The bottom two curves of Figure 3(a) also suggest that for smaller values of n both n-grams and n-perms may yield higher false positives, n-perms being the worse. We note, however, that these values and trends are likely to be specific to the properties of opcode sequences found in malware—perhaps particularly to Windows/Win32 malware.

Manual inspection of the resulting phylogenies showed the 5-gram and 10-perm phylogenies to be comparable, although the similarity scores for the hand-permuted examples were lower for 5-grams. The rise in similarity values for the distinct pairs at sample size 38 for Figure 3(b) may be noteworthy. This subsample differs from the prior one by containing several packed executables. The probable cause of this deflection is the fact that the added specimens are unlikely to have many features in common with the unpacked specimens so that the IDF values (computed as $\log(N/df_i)$) will rise since the df_i values will drop. This gives further reason to use only unpacked specimens for phylogeny model generation, as the packed specimens are unlikely to match meaningfully and may add false positives.

5 Study of naming and classification

We conducted a study to explore how generated phylogeny models might assist forensic analysts in understanding and naming malware specimens. We were aware that for any given specimen different AV scanners may produce distinct names [3]. Could the phylogenies help sort out and reconcile the different namings? We also wondered if the generated phylogenies could help the analyst when confronted with an apparently new and unrecognized malicious program.

5.1 Materials

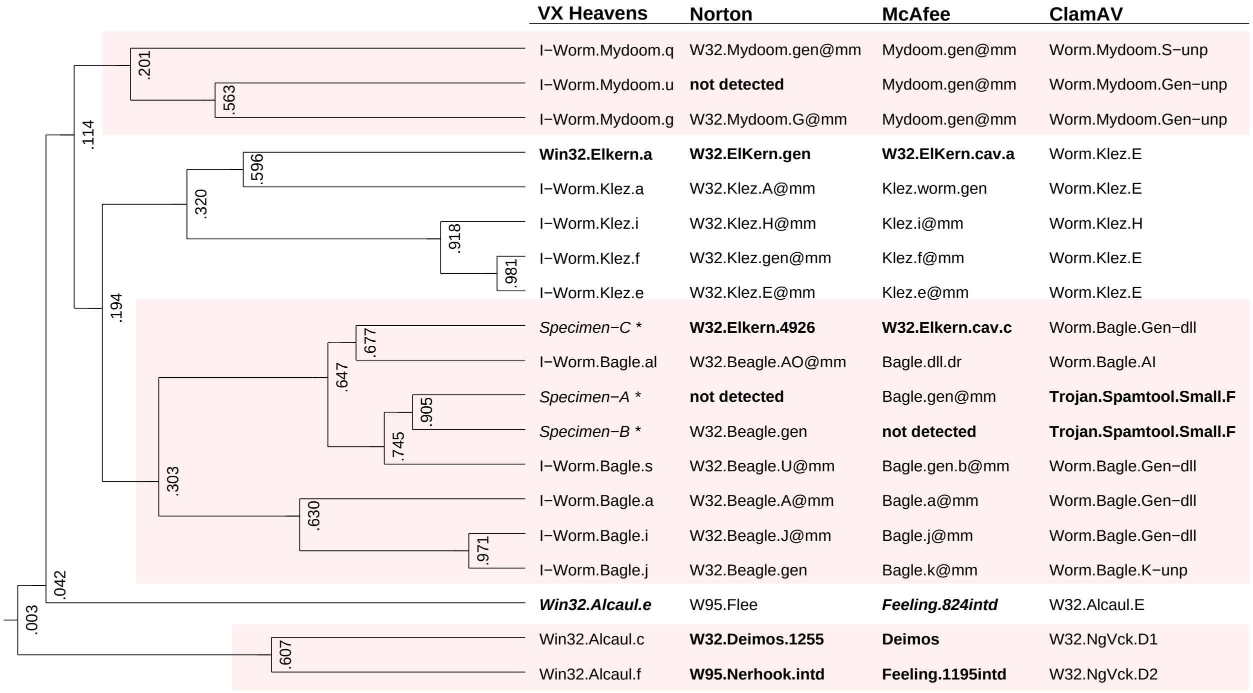
We simulated the condition where new malicious programs are obtained and compared against an existing database. We selected three specimens of worms that were recently captured on our departmental mail server—an indication that these were still circulating in the wild. From the `ClamAV` filter log we knew these were likely to be variants of the "Bagle" worm because the mail filter identified them as being `Worm.Bagle.`{`AG,AU,Gen-zippwd`}. The `Gen-zipped` suffix indicates a generic match, meaning that the specific variant was not identified, or was unknown to the AV scanner. These three specimens were unpacked and renamed to be `Specimen-A,B,C` so as to simulate the case where an unrecognized specimen arrives. `Specimen-C` could not be unpacked using any unpacker we had available to us, so we used an interactive debugger to step through the worm until it unpacked itself into memory and then "dumped" the memory image as an executable. While these transformed versions are not in a form that would be expected to circulate in the wild, we wished to determine whether the phylogeny tree generation method could still help the analyst with new specimens after an initial unpack or decrypt.

We then selected a sample of worms from the VX Heavens [12] collection to use as a base collection. To keep the study focused we selected only the 41 available specimens labeled with the names of the Windows worms `Bagle`, `Klez`, and `Mydoom`. We removed specimens that we could not unpack or which appeared to be redundant, particularly the ones that resulted in identical abstracted assembly. This left us with a sample of size of 15. We could not determine with certainty on what basis the VX Heavens collector named files but, however it was done, we adopted that naming scheme. We generated the abstracted assembly as before by disassembling the specimens and removing all but the opcodes.

We generated a phylogeny model for the sample of 18 specimens (15 reference plus the three "unknown" specimens) using 10-perms. We also scanned the collection with three anti-virus (AV) scanners with updated signatures and made notes of how they were identified. The scanners were Norton AntiVirus, McAfee VirusScan, and ClamAV for Linux. These are hereafter called simply "Norton", "McAfee", and "ClamAV".

The initial scan revealed that `Specimen-C` was identified by the name "Elkern" by both Norton and McAfee. At this point we wondered if the original "generic" match by ClamAV might have been a misclassification. To investigate this mystery we added the specimen called `Win32.Elkern.a` from the VX Heavens collection to the sample and re-generated the phylogeny model on the 19 specimens.

The results appear in Figure 4. The labels of the non-leaf tree nodes record the average similarities between two branches. The cross reference to the names extracted from the AV scanners appears to the



VX Heavens	Norton	McAfee	ClamAV
I−Worm.Mydoom.q	W32.Mydoom.gen@mm	Mydoom.gen@mm	Worm.Mydoom.S−unp
I−Worm.Mydoom.u	**not detected**	Mydoom.gen@mm	Worm.Mydoom.Gen−unp
I−Worm.Mydoom.g	W32.Mydoom.G@mm	Mydoom.gen@mm	Worm.Mydoom.Gen−unp
Win32.Elkern.a	**W32.ElKern.gen**	**W32.ElKern.cav.a**	Worm.Klez.E
I−Worm.Klez.a	W32.Klez.A@mm	Klez.worm.gen	Worm.Klez.E
I−Worm.Klez.i	W32.Klez.H@mm	Klez.i@mm	Worm.Klez.H
I−Worm.Klez.f	W32.Klez.gen@mm	Klez.f@mm	Worm.Klez.E
I−Worm.Klez.e	W32.Klez.E@mm	Klez.e@mm	Worm.Klez.E
*Specimen−C **	**W32.Elkern.4926**	**W32.Elkern.cav.c**	Worm.Bagle.Gen−dll
I−Worm.Bagle.al	W32.Beagle.AO@mm	Bagle.dll.dr	Worm.Bagle.AI
*Specimen−A **	**not detected**	Bagle.gen@mm	**Trojan.Spamtool.Small.F**
*Specimen−B **	W32.Beagle.gen	**not detected**	**Trojan.Spamtool.Small.F**
I−Worm.Bagle.s	W32.Beagle.U@mm	Bagle.gen.b@mm	Worm.Bagle.Gen−dll
I−Worm.Bagle.a	W32.Beagle.A@mm	Bagle.a@mm	Worm.Bagle.Gen−dll
I−Worm.Bagle.i	W32.Beagle.J@mm	Bagle.j@mm	Worm.Bagle.Gen−dll
I−Worm.Bagle.j	W32.Beagle.gen	Bagle.k@mm	Worm.Bagle.K−unp
Win32.Alcaul.e	W95.Flee	***Feeling.824intd***	W32.Alcaul.E
Win32.Alcaul.c	**W32.Deimos.1255**	**Deimos**	W32.NgVck.D1
Win32.Alcaul.f	**W95.Nerhook.intd**	**Feeling.1195intd**	W32.NgVck.D2

Figure 4: Phylogeny model and AV scanner naming cross-reference

right of the generated tree. The "main" clusters are highlighted using alternating shading; all of these clusters have within-cluster average similarity scores greater than .200, and the parent clusters all have average similarity scores less than .200. The asterisks beside the names in the first column are a reminder that these are not VX Heavens names, but rather the specimens that are to be treated as if they were new and unidentified. The entry "not detected" indicates the AV scanner did not determine the specimen to be malicious. Note that these scanner results are on the unpacked versions and can be different from the results generated for the original packed versions as they are typically found in the wild. For example, ClamAV reports the packed `I-Worm.Mydoom.u` as `Worm.Mydoom.W` and our decompressed version as `Worm.Mydoom.Gen-unp`.

5.2 Discussion

On classifying unidentified malware

All three of the unidentified specimens fall into the large subtree containing all the VX Heavens-identified `Bagle` worms. `Specimen-A,B` rate as highly similar, and the closest related specimen from the sample is the one named `I-Worm.Bagle.s`. We are confident that `Specimen-B` is a specimen of what Symantec calls `W32.Beagle.AU@mm` because we have verified the specimen embeds the same 145 strings for the web sites listed by Symantec[4] as being the worm's contact sites. These sites are likely hosts that were compromised at the time of release and are likely to change between releases. Through similar investigation we determined `Specimen-A` is likely to be a `W32.Beagle.AZ@mm` and `Specimen-C` is likely to be `W32.Beagle.l@mm`.

Based on this information the small study indicates that gross classification into families may be possible. Given the match of `Specimen-`{`A,B`}, some samples may be positively identified, although we would not be surprised if the AV companies already have reasonable methods for matching incoming

[4] `securityresponse.symantec.com/avcenter/venc/data/w32.beagle.au@mm.html`, last checked 2005.04.05.

samples to the closest known ones in their databases. We are less sure as to how meaningful it is that `Specimen-`{`A,B`} are clustered closely with the specimen named `I-Worm.Bagle.s`, and `Specimen-C` with `I-Worm.Bagle.al`. At the very least, this indicates to the analyst that they are likely to be different, and may suggest that two species from different lineages are circulating.

On name reconciliation

We examined the cross referenced names for cases where the tree would suggest the AV scanner naming is inconsistent. We looked for two classes of inconsistencies. For simplicity we will call these "within-cluster" and "between-cluster" inconsistencies. Within-cluster inconsistencies occur when specimens within a coherent subtree are named with different root names. Between-cluster inconsistencies occur when the same root name is used in multiple, well-separated subtrees. In Figure 4, the within-cluster inconsistencies are highlighted using bold face. The between-cluster inconsistencies are highlighted using italic bold face.

We examined these inconsistencies because we felt they might reveal naming problems. If a scanner generates two different names for entities within the same cluster, then the suspicion arises that the phylogeny model is clustering together unrelated specimens, the naming scheme established by the AV community is confusing, or that the AV scanner is inconsistent or incorrect in its identification and naming. Between-cluster inconsistencies may indicate that aliasing may be occurring because a single name is being used to identify specimens that are not substantially similar.

Investigating the inconsistencies, we saw the clustering of `Win32.Elkern.a` with `I-Worm.Klez.a` was a within-cluster inconsistency. Following this up, it became apparent that this clustering was likely due to the fact that `I-Worm.Klez.a` is known to drop the `Elkern` virus and they may have parallel and intertwined derivation relationships. This is a new and useful knowledge for us for understanding the malware relationships. `Trojan.Spamtool.Small.F` may have a similar reason of multi-pronged malicious attack in its apparently inconsistent naming, since the (terse) description from the ClamAV database update log[5] appears to suggest the specimen is related to `Bagle`'s operations.

Two `Elkern` variants detected by both Norton and McAfee created between-cluster inconsistencies. One of them, corresponding to `Specimen-C`, may actually belong to `Bagle` family.

Another inconsistency appears in the `Alcaul` trees. While ClamAV reports the `I-Worm.Alcaul.`{`c,f`} cluster consistently, both Norton and McAfee report different names. ClamAV's name may indicate why these two specimens may be related: `W32.NgVck` is the name of a virus construction kit. This kit may have been used to create both specimens. Also note that `I-Worm.Alcaul.e` and `I-Worm.Alcaul.f` are recognized by McAfee as `Feeling` variants, which created between-cluster inconsistency. All these suggest that perhaps the naming scheme could be improved and automated through an accepted phylogeny model.

6 Conclusion

Although our results must be considered preliminary, we found that n-perms produce higher similarity scores for permuted programs and produce comparable phylogeny models. n-perms appear to do a better job in differentiating related and unrelated similarities in sample sets with permuted variants, suggesting it is a better choice for constructing phylogeny models in the presence of malware that has evolved through permutations. Our study results suggest that phylogeny models generated using this technique may be able to help reconcile naming inconsistencies and assist in the investigation of new malicious programs.

The results suggest avenues for further investigation. In the present work we used exact matching of n-perms: two n-perms must have exactly same set of elements to match. We focused on exact matching because our motivation was tracing malware evolution in the presence of code reordering. However, apart from code reordering, other evolution steps including instruction substitution, insertion, and deletion could be present. We wonder whether approximate n-perm matching could track evolution in the presence of insertions, deletions, and instruction substitutions. Another possible line of investigation is in combining

[5] From "clamav-virusdb update (daily: 765)" at `lurker.clamav.net/message/20050317.084759.1176d20a.en.html`, Last retrieved 29 March, 2005.

the results of n-perm and n-gram matching. We noted that in Figure 3(a), as n grows the similarity scores for n-perms and n-grams become increasingly mismatched for our hand-crafted permuted variants. This implies that it might be possible to use this similarity score difference to specifically search for or classify permutation variants. For example, if n-perms are used initially to find a family of related specimens, n-gram similarity scores might help identify the ones likely to be related through permutations.

Finally, an intriguing question remains as to whether a threshold can be specified as a useful heuristic for determining whether a new name or a variant name should be assigned to a new specimen. Finding a suitable heuristic threshold would be a useful contribution for both phylogeny model generation, and for assisting in the debates on naming.

Acknowledgements

The authors would like to thank Michael Venable and Rachit Mathur for help in preparing the malicious code samples we used.

References

[1] B. Arief and D. Besnard, "Technical and human issues in computer-based systems security," Tech. Rep. CS-TR-790, School of Computing Science, University of Newcastle-upon-Tyme, 2003.

[2] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge, UK: Cambridge University Press, 1997.

[3] V. Bontchev, "Anti-virus spamming and the virus-naming mess: Part 2," *Virus Bulletin*, pp. 13–15, July 2004.

[4] C. Raiu, "A virus by any other name: Virus naming practices," *Security Focus*, June 2002. `http://www.securityfocus.com/infocus/1587`, Last accessed Mar 5, 2005.

[5] P. Ször and P. Ferrie, "Hunting for metamorphic," in *Proceedings of the 12th Virus Bulletin International Conference*, pp. 123–144, 2001.

[6] National Center for Biotechnology Information, "Just the facts: A basic introduction to the science underlying NCBI resources," Apr. 2004. `http://www.ncbi.nlm.nih.gov/About/primer/phylo.html`, Last retrieved 20 March, 2005.

[7] M. Godfrey and Q. Tu, "Growth, evolution, and structural change in open source software," in *Proceedings of the 4th International Workshop on Principles of Software Evolution*, (Vienna, Austria), pp. 103–106, ACM Press, 2001.

[8] B. S. Baker, "A program for identifying duplicated code," *Computing Science and Statistics*, vol. 24, pp. 49–57, 1992.

[9] Á. Beszédes, R. Ferenc, and T. Gyimóthy, "Survey of code-size reduction methods," *ACM Computing Surveys*, vol. 35, pp. 223–267, Sept. 2003.

[10] B. S. Baker and U. Manber, "Deducing similarities in java sources from bytecodes," in *Proceedings of the USENIX Annual Technical Conference (NO 98)*, June 1998.

[11] R. Marko, "Heuristics: Retrospective and future," in *Proceedings of the Twelfth International Virus Bulletin Conference*, (New Orleans, LA), pp. 107–124, Virus Bulletin, Ltd., 2002.

[12] "VX heavens." Available from `vx.netlux.org` (and mirrors), Last retrieved 5 March, 2005.

[13] M. Jordan, "Dealing with metamorphism," *Virus Bulletin*, pp. 4–6, Oct. 2002.

[14] V. Bontchev and K. Tocheva, "Macro and script virus polymorphism," in *Proceedings of the Twelfth International Virus Bulletin Conference*, (New Orleans, LA), pp. 406–438, Virus Bulletin, Ltd., 2002.

[15] J. O. Kephart, "A biologically inspired immune system for computers," in *Artificial Life IV: Proceedings of the Fourth International Workshop on Synthesis and Simulation of Living Systems* (R. A. Brooks and P. Maes, eds.), pp. 130–139, Cambridge, MA: MIT Press, 1994.

[16] J. O. Kephart, G. B. Sorkin, W. C. Arnold, D. M. Chess, G. J. Tesauro, and S. R. White, "Biologically inspired defenses against computer viruses," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, (Montreal, PQ), pp. 985–996, Morgan Kaufman, 1995.

[17] J. O. Kephart and W. C. Arnold, "Automatic extraction of computer virus signatures," in *Proceedings of the 4th Virus Bulletin International Conference* (R. Ford, ed.), (Abingdon, England), pp. 179–194, Virus Bulletin Ltd., 1994.

[18] G. Tesauro, J. O. Kephart, and G. B. Sorkin, "Neural networks for computer virus recognition," *IEEE Expert*, vol. 11, no. 4, pp. 5–6, 1996.

[19] W. Arnold and G. Tesauro, "Automatically generated Win32 heuristic virus detection," in *Proceedings of the 2000 International Virus Bulletin Conference*, 2000.

[20] J. Z. Kolter and M. A. Maloof, "Learning to detect malicious executables in the wild," in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (W. Kim, R. Kohavi, J. Gehrke, and W. DuMouchel, eds.), (Seattle, WA), pp. 470–478, ACM, 2004.

[21] T. Abou-Assaleh, N. Cercone, V. Kešelj, and R. Sweidan, "Detection of new malicious code using n-grams signatures," in *Second Annual Conference on Privacy, Security and Trust*, (Fredericton, NB, Canada), pp. 193–196, 2004.

[22] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, (Oakland, CA), pp. 38–49, IEEE Computer Society Press, 2001.

[23] M. F. X. J. Oberhumer and L. Molnár, "The Ultimate Packer for eXecutables – homepage." `http://upx.sourceforge.net`, Last retrieved 20 March, 2005.

[24] L. A. Goldberg, P. W. Goldberg, C. A. Phillips, and G. B. Sorkin, "Constructing computer virus phylogenies," *Journal of Algorithms*, vol. 26, pp. 188–208, 1998.

[25] G. Erdélyi and E. Carrera, "Digital genome mapping: Advanced binary malware analysis," in *Proceedings of 15th Virus Bulletin International Conference (VB 2004)*, (Chicago, IL), pp. 187–197, 2004.

[26] S. Wehner, "Analyzing worms using compression." `http://homepages.cwi.nl/~wehner/worms/`, Last accessed Mar 5, 2005.

[27] W. F. Tichy, "The string-to-string correction problem with block moves," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 309–321, 1984.

[28] J. Zobel and A. Moffat, "Exploring the similarity space," *SIGIR Forum*, vol. 32, no. 1, pp. 18–34, 1998.

[29] G. Karypis, "CLUTO: A clustering toolkit, release 2.1.1," Tech. Rep. #02-017, Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, Nov. 2003.