The 5th LCN Workshop on Security in Communications Networks (SICK 2009)
Zürich, Switzerland; 20-23 October 2009

Measuring Similarity of Malware Behavior

Martin Apel, Christian Bockermann and Michael Meier
University of Dortmund, D-44221 Dortmund, Germany
{martin.apel, christian.bockermann, michael.meier}@udo.edu

***Abstract*—Malicious software (malware) represents a major threat for computer systems of almost all types. In the past few years the number of prevalent malware samples has increased dramatically due to the fact that malware authors started to deploy morphing (aka obfuscation) techniques in order to hinder detection of such polymorphic malware by anti-malware products. Using these techniques numerous variants of a malware can be generated. All these variants have a different syntactic representation while providing almost the same functionality and showing similar behavior. In order to effectively detect polymorphic malware it is advantageous (if not required) to know which malware samples are variants of a particular malware. Respective approaches for determining this relation between malware samples automatically are currently investigated by a number of researchers. A prerequisite for assessing this relation based on particular features of malware samples is an appropriate similarity or distance measure. In particular a number of approaches for clustering malware samples have been recently published. Thereby different similarity measures are used but without thoroughly discussing their choice. So it is an unanswered question which similarity measures are appropriate for determining respective relations between malware samples. To answer this question we study different distance measures in detail and discuss desirable properties of a distance measure for this particular purpose. We focus on behavioral features of malware and compare and experimentally evaluate different distance measures for malware behavior. Based on our results we identify a most appropriate distance measure for grouping malware samples based on similar behavior.**

I. Introduction

Malicious software (malware) is defined as software performing actions intended by an attacker without consent of the owner when executed. Types of malware include worms, viruses, Trojan horses, bots, spy- and adware. Nowadays technology to detect known malware is mostly based on syntactic signatures. Such signatures specify binary instructions or data sequences that are characteristic of a particular malware sample. In the past few years the number of observed malware samples has increased dramatically, which is owing to the fact that attackers started to morph malware samples in order to avoid detection by syntactic signatures. Using morphing engines like ADMmutate [1] numerous polymorphic malware samples can be generated for a particular malware sample automatically. McAfee, a manufacturer of malware detection products, reported that 1.7 million out of 2.8 million malware samples collected in 2007 are polymorphic variants of other samples [2]. All polymorphic variants of a sample provide almost the same functionality and show similar behavior but have different syntactic representations. Thus detecting these samples using syntactic signatures requires a signature for each particular sample. As a consequence a huge number of signatures need to be created and distributed by anti-malware companies, and have to be processed by detection products at the users system, leading to degradation of the systems performance.

A proposal to solve these problems is the treatment of malware behavior families. That is all samples that are polymorphic variants of each other are treated as members of the same malware behavior family. In order to bring forward techniques that exploit this idea, an automatic method for appropriately grouping samples into respective behavior families is required. Developing such a method is the focus of this contribution.

Having a method that automatically identifies the behavior family of a sample would be the first step towards creating family signatures. A family signature matches all samples of the whole family (and in the ideal case no sample of any other family) and would significantly help to reduce the overall number of signatures required. The method also helps in creating a common naming scheme for malware samples which is a crucial practical problem within the anti-malware community. Since new samples can be mapped to a behavior family of known already analyzed samples, the required effort for analyzing and reverse engineering of the new sample will be reduced.

Mapping malware samples based on some features to families of samples with similar features is a typical application scenario of clustering methods. Regarding the kind of features that are used to analyze malware static and dynamic analysis can be distinguished. Static analysis focuses on features that can be directly extracted from a malware sample. Sequences of data and instructions of a sample are a typical example. Morphing techniques and tools demonstrate that programs of similar functionality do not have to share similar instruction and data sequences. Consequently higher-level structural features like the control-flow graphs (cfg) of programs, which can be extracted by disassemblers, are studied (e.g. [3], [4], [5], [6]). These techniques assume that similar programs have to share a similar structure (cfg). Unfortunately, albeit not yet commonly applied by today's malware authors, techniques exist to change a programs structure (cfg) without changing its functionality. Dynamic analysis avoids these drawbacks by focusing on the behavior of a program, which can be extracted by executing a program and closely observing its activities e.g. at the system call level. A difficulty with dynamic techniques is to trigger the malicious execution path of the program under observation. However, dynamic analysis is a promising alternative for malware family analysis.

A number of proposals to cluster malware have already

978-1-4244-4487-8/09/$25.00 ©2009 IEEE

been made in the literature but only a few focus on behavior of malware. In this paper we present results in developing techniques for clustering malware behavior. Our contribution is threefold: Firstly we identify and discuss desired properties and requirements of an appropriate similarity/distance measure for malware behavior. Secondly we systematically evaluate a number of prominent similarity measures regarding these properties and requirements. Thirdly we present results of an experimental evaluation comparing different similarity measures. The paper is structured as follows. Section II discusses background and related work. Section III introduces the investigated distance metrics as well as the criteria used for comparison. The evaluation methodology and experimental setup is described in section IV. Evaluation results and their discussion are presented in section V and we conclude in section VI.

II. Background and Related Work

Background of the work described here is the AMSEL project, which focuses on the development of a malware early warning system and automatically

- collects malware samples using malware collectors (e.g. Nepenthes [7], Amun [8], HoneyClients [9] and SPAM-Traps),
- analyzes malware samples using dynamic analyzers like CWSandbox [10] and Anubis [11],
- groups collected malware samples into malware behavior families, and
- generates malware behavior family signatures for behavioral detection of malware.

The system aims at automatically providing detection signatures for newly occurred malware as fast as possible. As a prerequisite to generating family signatures we must be able to automatically group the behavior of malware samples into families which is realized using clustering. A few prior proposals on automatically clustering and classifying malware have been made in the literature. Static as well as dynamic features of malware have been used for this purpose.

Karim et al [12] follow the syntactic approach by using frequencies of n-grams of opcodes of instructions present in the malware binary for clustering utilizing standard vector based distance measures. They recognized that instruction reordering would circumvent their approach and propose to use n-perms (variations of n-grams) as feature in order to assign two malware samples with permutated instructions into the same cluster. Unfortunately a number of morphing techniques beyond instruction permutation exist, which would successfully break this approach.

Another (rather) syntactic approach is taken by Wehner [13]. She proposes to use the normalized compression distance (NCD) based on the Kolmogorov Complexity of malware approximated by the compressibility of the malware samples to assess similarity. Again due to its syntactic nature the clustering approach is vulnerable to numerous morphing techniques.

Structural features of malware have also been used to assess the similarity between malware [3]. These proposals typically assume that morphing techniques do not change structural properties of a malware sample. The control flow graph (cfg) of a program is used as feature in these approaches. Extracting this feature requires disassembly of the malware sample. Matching of cfg-signatures in order to detect malware samples has been proposed [14], [15], [5]. Unfortunately techniques exist that make it hard to disassemble a malware sample. Further, as has been discussed by the authors of these approaches, a number of morphing techniques are able to change the cfg of a malware sample while keeping its functionality unchanged.

Due to the limitations of static analysis techniques (see Moser et al [16] for discussion) dynamic techniques analyzing the behavior of malware samples were brought into focus. The basic idea of differentiating programs based on their behavior was first used by Forrest et al [17]. They used sequences of six (later ten) system calls executed by a process to learn a normal model of programs and alert on any deviation from the model in order to detect security violations. Another possibility for extracting structural features from binaries is shown by Daikon [18], which dynamically detects likely invariants in C, C++, Java or Perl programs. As a lot of malware is written in Visual Basic or Delphi it is too limited to be useful in our scenario.

Recently a few proposals have been made to dynamically analyze malware automatically. Bailey at el [19] use more abstract behavior features than simple system or library calls. They chose NCD as distance measure for hierarchical clustering of malware, but do not give reasons for this choice. NCD measures syntactic complexity of behavior reports of malware samples and is known to be universal and does not use any specific information. For example the order of activities in a behavior report is not considered particularly, which raises the questions, if better clustering results can be gained using more appropriate distance measures. This motivates a systematic comparison of distance metrics for clustering behavior of malware.

Lee at al [20] also automatically cluster malware behavior and discuss the choice of a distance measure but did not perform any experimental evaluation. They chose of a variant of the edit distance measure, which has a runtime complexity that is quadratic in the number of system calls in a behavior report and therefore is far too expensive in our scenario.

Rieck et al [21] developed an automatic classification system for malware samples. They learn a classificator based on samples labeled by anti-virus products using support vector machines (SVMs). All occurring samples of new malware families that are unknown to anti virus products would be classified as unknown by this approach. Therefore this classification approach cannot be used for clustering malware samples into families.

Another approach [22] focuses on the alteration of program behavior. It continously monitores a program and compares the current behavior (respectivly system call sequences) to the behavior observed up to this point. Depending on the degree of

analogy between the two behaviors an anomaly signal is send. A central instance correlates the anomaly signals of multiple clients and performs epedemic detection. This does not fit into our scenario, as we want to group software that behaves similarily and are not interested if a software starts to exhibit *new* behavior.

Currently typical malware sample databases contain a number of hundred thousands or even millions of different prevalent malware samples which need to be grouped into families. Each time a new malware sample enters the database the family grouping potentially needs to be updated. In order to support our early warning scenario we need a fast efficient automatic approach to group malware samples on this scale. We investigate which distance measures are appropriate for clustering malware behavior in general and for application in our early warning system in particular. For extracting behavior features of a malware sample we use CWSandbox [10] that monitors the execution of the sample. By executing each sample in the CWSandbox environment for two minutes we get a behavior report representing an execution trace of the sample. Such a trace contains all system calls (one per line) executed by the sample in the order of execution.

III. Distance Metrics

To compare and group such malware traces, a distance measure is needed. Several distance metrics have been investigated in the area of malware research (c.f. section II) but most of them focus on static analysis. We are interested in malware behavior and therefore in distance metrics that are able to differentiate traces of malware execution. The following criteria are important for a distance measure when dealing with malware traces:

1) It has to be *appropriate* in the sense that malware traces representing similar behavior have a small distance, but malware traces of different behavior should have large distances.
2) It has to be efficiently computable due to the huge number of traces that need to be analyzed. In the context of AMSEL [23] or other early warning systems, results are required as fast as possible.
3) The order of system calls in the trace should somehow be respected. The order of system calls, especially their *local* order, can make a big difference. The distance metric should therefore be sensitive to local reordering of system calls. At a larger scale (between blocks of system calls) the order is not that important. It is even desireable that the distane measure is insensitive to these *global* changes, as they can easily vary between multiple executions of the same malware sample. The reasons for this are changed data or changed order of the thread blocks in the trace file. We therefore want the metric to be order sensitive at a local scale but insensitive at a larger or *global* scale.

The following distances have been investigated:

A. Edit Distance

The edit distance (also known as Levenshtein-Distance) is one of the most prominent distance measures for sequential data. It has also been successfully used with malware research by Lee and Mody in [20]. The edit distance measures the minimal costs to make two strings equal. It usually uses the operations insert, delete and modify, of which each may be assigned a different cost. If these costs all equal one, then the costs of making *aaa* into *abb* is two (two modifications) and so is their distance while the distance between *abc* and *abcd* is one (*d* has to be added).

Insert and delete operations should have the same cost, because they can be interchanged (every character added in string S_1 could have been deleted from string S_2 as well). The trivial implementation of the edit distance has a runtime complexity of $O(nm)$ and a space complexity of $O(nm)$. One of the best known algorithm is the Hirschberg algorithm [24], [25], which computes the edit distance with runtime complexity $O(nm)$ and space complexity of $O(n)$.

The standard edit distance works on single characters. This is not appropriate for distances between sequences of system calls as the distance of two system calls is independent of their textual representation - i.e. $d('create_thread','create_file')$ should not be smaller than $d('create_thread','fork')$. The edit distance used here does therefore not work on single characters but on single system calls. Because the traces contain one system call per line the edit distance treats each line as a single symbol. The costs for the *insert* and *delete* operations are one regardless of the specific system calls.

B. Approximated Edit Distance

Because of the quadratic complexity of the edit distance a solution that approximates the edit distance seems appropriate. An often occurring problem in forensics is to determine if a certain file has been altered. Hash functions have been used quite successfully to solve this problem. Their specific application has been altered over the time. The first step towards context triggered piecewise hashing was blockwise hashing instead of hashing the whole file to determine local changes. This solution fails when bytes are removed or inserted into the file, which lead to the development of context triggered piecewise hashing. Context triggered piecewise hashing has been described by Kornblum in [26]. Essentially it uses two hash functions f_1 and f_2. The function f_1 is used to compute the blockwise hashes and can be any *classical* hash function like md5 or sha-1. The function f_2 is used to determine the borders of the blocks and is usually a *rolling hash* which depends only on the last n bytes seen. If this hash (f_2) hits a certain predefined value, f_1 is used to hash the corresponding block. The rolling hash guarantees, that the blocks hashed by the function f_1 are synchronized after an insertion or deletion took place. It has been proven that at most two blocks are needed for the synchronization to happen.

The output of the algorithm described above is a sequence of hash values (h_1 and h_2) for each file (F_1 and F_2). Instead of comparing the files with the edit distance the distance between

F_1 and F_2 is computed by computing the edit distance of the corresponding hash strings h_1 and h_2. The hash strings are a lot smaller than the original files and the edit distance between them is therefore efficiently computable. The factor between a file and its corresponding hash string depends on the average blocksize and therefore on f_2 (it also depends on the actual content of the file but the effect is negligible). It is obvious that the increased run-time performance results in a loss of quality, because information on how many bytes are changed in a changed block is lost.

The approximation is quite good for almost identical files, because they share large parts and therefore a lot of hashed blocks will be equal. It also effectively detects changes that are done to a small number of locations, because this will effect only a small number of blocks. The results are bad if a lot of small changes are fairly well distributed over the whole file because this will result in changes in almost every hashed block and result in a maximal distance which might be wrong regarding regular edit distance.

The descried approximation of the edit distance has been used in other fields than forensics namely in spam detection (SpamSum) and finding source code reuse. Kornblum has published the program ssdeep [27] which can be used to compute the similarity of files using context triggered piecewise hashing.

C. Normalized Compression Distance

The *normalized compression distance* (NCD) relies on the universality of the Kolmogorov complexity, which assigns a complexity value to an object [28]. Although the Kolmogorov complexity itself is not computable, a good compression algorithm can be used to approximate its results as shown in [29], [30]. NCD therefore uses the ability of compression functions to find similarities in files to define a distance between those files. Let $C(x)$ be the compression function then the Normalized Compression Distance (NCD) between X and Y is defined as:

$$NCD(X,Y) = \frac{|C(XY)| - min\{|C(X)|, |C(Y)|\}}{max\{|C(X)|, |C(Y)|\}}$$

The charming bit about the *NCD* is its universality. It can be computed between any two files and still retrieves meaningful results. It can be suspected that this kind of universality and elegance has driven its use in numerous papers [19], [31], [32], [33], [34], [35], [36]. There are nevertheless some drawbacks that are linked to the used compression function $C(x)$. Cebrián et. al. showed in [37] that some blockoriented compression functions (bzip2, gzip, ...) are only usable up to 32kb filesize. There are two other compression functions that are often used with *NCD* namely lzma and ppmd. Our work analysis showed that ppmd suffers from the same drawbacks as bzip2 and gzip, but lzma is immune. Despite of this *NCD* has been used in both static and dynamic malware analysis. Wehner used it in [13] for clustering malware binaries. The results were promising for unpacked (unmorphed) malware but poor for packed (morphed) malware.

Bailey et. al [19] used *NCD* to cluster malware using behavioral reports. These reports did not use system calls but high-level system changes (changed files or registry keys). The paper does not say which compression function was or how big the reports got, but it can be suspected that the reports were sufficiently small to allow the use of bzip2 or gzip.

The universality of NCD is also its weakness. It does not act on any context information but solely on the structure imposed on the file by the compression function itself which can be inappropriate to characterize the content of the file. It also lacks possibilities that normally come with the understanding of the data structure as to ignore certain parts of the data or use weighting techniques. Whether these are major drawbacks for malware behavior analysis is an unanswered question.

D. Manhattan Distance Using Tries

A quite different idea for a distance metric when dealing with sequential data is the use of an *embedding* function which embeds the sequential data into a high-dimensional vector space using a formal language L. Each dimension of a vector corresponds to a specific word of L and holds the number of occurrences in the sequence. The dimension of each vector depends on the number of words in L which is usually very high. On the other side the vectors itself are very sparse. There exist data structures that are tailored for such vectors, namely arrays, tries and suffix trees which are all described in [38]. In our experiments we used tries to represent a vector, which do have very low runtime requirements. The formal language used here are n-grams of system calls. We have varied n between one and ten and evaluated the results. To compute the distance between two vectors the L_1 metric (Manhattan distance) is used.

IV. Evaluation Methodology and Setup

A malware corpus of 1195 samples collected using honeypots has been used for this evaluation. Besides the system calls and their parameters the traces generated by CWSandbox contain additional information regarding the analysis process itself and all information is represented in an XML-Structure. Because these information does not describe the behavior of the malware but the analysis itself, the traces are preprocessed. The preprocessing removes the XML structure in a first step. It further constructs the process-thread-tree, sorts it and writes the sorted process and thread blocks back to the trace file. During sorting the thread and process ids are exchanged for labels that represent their position in the tree. This step extinguishes the *random* ids and normalizes the structure of the process-thread-tree and therefore the structure of the trace. The preprocessing will also remove the parameters from the system calls. The usage of the parameters would result in finer granularity but also in higher runtime and space costs. As we want the clusters to be large and not over-specific as well as the distance function to be efficiently computable, the parameters are omitted.

A. How to Measure Appropriateness

The distance measures shall be used to group malware traces together that describe similar behavior. The grouping is done using clustering algorithms which are using the investigated distance measures. There exist numerous clustering algorithms which can be used, but their evaluation is beyond the scope of this work. Therefore only one clustering algorithm is used here, namely single-link hierarchical clustering [39]. Every distance metric is used to produce a partition of the 1195 traces (set of disjoint subsets containing all traces). To measure the appropriateness of the distance metric we therefore have to measure the appropriateness of the resulting partitions.

Evaluating the partitions is particularly hard due the the difficulty to establish a *ground truth*. It may seem a possibility to use the virus labels assigned to the corresponding malware samples by any anti-virus (AV) scanner. But the labels assigned by AV products heavily depend on static signatures. These signatures evaluate static properties of the binary in contrast to the behavioral properties reported by CWSandbox and the families that are found using static properties might be quite different from the ones established using behavioral features. Our experiments using these labels in conjunction with clustering of behavioral reports where quite discouraging.

In [19] Bailey et. al showed that the virus labels defined by AV products are not consistent between different vendors, not even after annihilating aliases. There are nevertheless succesful attempts to bridge this gap for example [21] but this work does not use clustering algorithms but SVMs to learn a classifier and uses the behavioral properties to approximate the virus labels.

It is not the goal of this paper to approximate the virus labels assigned by any AV product but to find consistent and crisp clusters of *malware samples* that share similar behavior. There are therefore significant differences to our approach:

- No labeled samples are needed.
- The number of groups/clusters is not known in advance.
- Comparison with existing labels derived from static signatures is inappropriate.

We want to rate a distance measure d according to the amount of shared behavior that can be found after performing a clustering of the traces using d. Shared behavior of a group C_i (cluster) of traces is modeled in terms of continuous system call sequences that exist in each trace contained in the group (cluster). To find these sequences a well known algorithm from Ukkonen [40] is used. Each system call is modeled as a single character which means that each trace can be regarded as a set of strings (each thread is represented as a string). A generalized suffix tree is used to find all substrings, that are contained in each set of strings, e.g. in each trace contained in C_i. The generalized suffix tree can be constructed in linear time (in the size of the traces), using the algorithm described by [40]. The shared substrings contained in the generalized suffix tree, can also be found in linear time, which means that the shared behavior of a cluster C_i can be determined in linear time.

A distance measure is considered *better* than another distance measure, if shared behavior can be found for more traces. If the number of traces for which shared behavior could be found is equal, the number of clusters should be considered. The distance measure that lead to less clusters provides a shorter description of the shared behavior and is therefore preferred. To guarantee that the shared behavior found for the clusters is not *too* short to be useful a heuristic is used, which is described below.

B. Configuration of the Cluster Algorithm

For cluster algorithms which produce a flat cluster structure, the method described above can be directly used to identify the shared behavior of each cluster. If the cluster algorithm produces a hierarchical cluster structure a method to select the clusters whose shared behavior shall be identified is needed. The additional effort is rewarded with better results as the hierarchical structure can be *unwound* until enough shared behavior can be determined. Hierarchical clustering algorithms produce a hierarchy of clusters, where the height of the links correspond to the dissimilarity of the merged clusters. The resulting hierarchical structure can be visualized using a so called dendrogramm. An example is given in Fig. 1 where six objects named A to F are clustered.

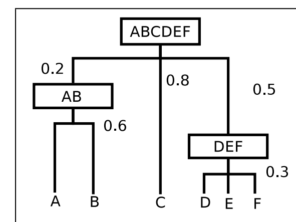

Fig. 1. Example Dendrogramm

To retrieve a flat clustering from a hierarchical clustering one has to decide when to stop the merging of the clusters. Multiple general purpose methods are described in the literature, most prominently cutting at a predefined constant depth, cutting links that are larger than a certain threshold and cutting links that are considerably larger than their siblings (*inconsistency value*) [41]. Cutting the example clustering from Fig. 1 at the constant depth 1 would produce a partition containing the clusters $C = \{\{A, B\}, \{C\}, \{D, E, F\}\}$. If links larger than 0.4 are cut the resulting clusters would be $C = \{\{A\}, \{B\}, \{C\}, \{D, E, F\}\}$. If an inconsistency coefficient of 2.0 is used instead to retrieve a flat clustering, the resulting clusters would be $C = \{\{A, B\}, \{C\}, \{D, E, F\}\}$. Using an inconsistency coefficient of 3.0 would result in $C = \{\{A, B\}, \{C, D, E, F\}\}$.

The use of an *inconsistency value* is a good choice in general but for our particular problem domain another approach suits better and delivered better results. This approach uses the hierarchical structure explicitly and derives information from it. To do so, the dendrogramm is traversed from the largest cluster (the root) down to the leaves. For every node (or

cluster) the amount of shared behavior of the contained traces is determined and rated. A function $sb : C_i \rightarrow S^3$ is defined which computes the shared behavior in C_i using generalized suffix trees as described above. S^3 is the set which contains all substrings of system calls that occur in all traces $t \in C_i$. To rate the amount of shared behavior the first choice would be to sum up the length of the sequences contained in S^3. This rating function would prefer a long sequence which potentially contains only repetitions of the same system call over medium sized sequence containing a mixture of system calls. As the medium sized sequence containing a mixture of system calls does contain a lot more information than the long sequence containing a single system call this is not desirable and another rating function is needed. Another possibility would be to use the entropy of the shared behavior. This would explicitly take into account the amount of information contained in the shared behavior. To compute the entropy can be a CPU time consuming task. We therefore choose an approximation of the entropy and just count the number of different system calls in the sequence. The diversity of a sequence of system calls S is defined as the number of different system calls in S. The *shared behavior rating* $sbr(SS)$ is defined as the sum of the diversities of the sequences in SS and is used to rate the amount of shared behavior in C_i and will be called *diversity* in the sequel.

A node i (respectively a cluster C_i) is selected for a cut, if the traces in C_i contain shared behavior which *diversity* is rated above a certain threshold (e.g. if $sbr(C_i)$ is bigger than 50).

If the described algorithm is used with the example in Fig. 1 the cluster containing all elements would be tested first. If there is not enough shared behavior, the clusters $\{A, B\}, \{C\}$ and $\{D, E, F\}$ are tested. $\{C\}$ can not contain shared behavior as there is just one element in it. Assuming that the other two clusters do contain enough shared behavior, the resulting clusters would be $C = \{\{A, B\}, \{C\}, \{D, E, F\}\}$ with five traces containing shared behavior. Note that the amount of shared behavior is not a distance measure as it does not fulfill the triangle inequality and therefore can not be used directly for the clustering.

C. Measuring Order Sensitiveness

To evaluate the order sensitiveness of the different distance metrics an artificial dataset has been created. Three different *real world* malware traces of different size and structure have been chosen (2158, 7901 and 16709 system calls). Each of these traces has been divided into blocks of length 1, 5, 25, 125, 600 and 3000 system calls. The blocks have been permutated randomly and written to new trace files. The distances of these 18 traces to their corresponding original traces are measured to evaluate the order sensitiveness of the distance metric. The distances for each block size of the three traces are then averaged and can be compared. A distance measure is considered *good* if it is sensitive to changes at a *local* level (small block sizes), but insensitive to changes at a *global* level (large block sizes).

D. Measuring Run-Time Performance

As most of the calculations have been performed on a High Performance Computing (HPC) environment that measures the CPU time needed by each job, the measurement of the run-time performance was pretty straightforward. The times were simply taken from the job reports and describes the needed CPU time in seconds. As the time needed by the approximated edit distance is considerably shorter than that of the other distance metrics, it was computed on an standard desktop PC and the consumed time was measured using the *time* command of Linux systems.

V. Results

A. Evaluation of Appropriateness

In the used malware sample set shared behavior that fits our constraints (has a minimum diversity of 50) could be determined for 1195 trace files. 1195 is therefore the maximum number of traces, for whom shared behavior could be identified by the described and used algorithm. Results of the evaluation of appropriateness are shown in Fig. 2.

The evaluation of the appropriateness of NCD_{lzma} failed, as it produced just a single cluster, which contains all traces. The reasons for this is that NCD only measures structural similarity. For every malware trace in our set, there are multiple other traces that have distance 0 to the corresponding trace. As the single-linkage clustering algorithm works in a transitive way (if $d(A, B) = 0 \wedge d(B, C) = 0 \Rightarrow d(A, C) = 0$) all traces are joined into the first cluster in the first step of the clustering. It was not possible to identify shared behavior in this maximal cluster which means that NCD_{lzma} can not sufficiently differentiate between malware traces.

Distance	#Traces	#Clusters	ØDiversity
Edit	893	166	80.14
Approx. Edit	986	87	118.69
NCD_{ppmd}	747	172	103.82
NCD_{lzma}	–	–	–
Manhattan 1	1064	77	83.69
Manhattan 2	1066	74	86.14
Manhattan 3	1066	76	86.09
Manhattan 4	1045	71	85.56
Manhattan 5	1042	71	88.18
Manhattan 6	1041	72	87.54
Manhattan 7	1042	71	87.96
Manhattan 8	1041	72	88.56
Manhattan 9	984	74	88.46
Manhattan 10	982	72	88.83

Fig. 2. Appropriateness of the distance measures

The usage of NCD_{ppmd} delivers slightly better results as shared behavior could be found for 747 trace files in 172 clusters. The average diversity of the found shared behavior is quite high, but this is due to the low average number of traces per cluster (5). A similarly high number of clusters is found if the edit distance is used. The number of traces for

which shared behavior could be found is 893 which means that 74,73% of the shared behavior has been found. Clustering using approximated edit distance surprisingly yields better results than the edit distance itself. Shared behavior is found for 986 traces (87,64%) which are distributed over much less clusters (87). The best results are achieved using the vectorization of sequences along with the Manhattan distance. Depending on the chosen value for n shared behavior is found for 982 to 1066 traces, which are distributed among approximately 75 clusters. The number of traces for which shared behavior could be found is decreasing by trend if n is increasing. The only exception is $n = 1$, which yields slighly worse results than $n = 2$ and $n = 3$. The number of traces for which shared behavior could be found is nevertheless quite close for $1 \leq n \leq 8$. Because of this the appropriateness for the Manhattan distance is very good, as long as $1 \leq n \leq 8$. The results for $n = 9$ and $n = 10$ are slightly worse, but still good. The results show that the average diversity is increasing, if the average number of trace files per cluster decreases. This is due to the higher probability to find more shared behavior, if the clusters are smaller.

B. Evaluation of Order Sensitiveness

Results are shown in Fig. 3 and Fig. 4. The ten Manhattan distance variations are presented in a separate chart to avoid overcrowding the figure. Edit distance, approximated edit

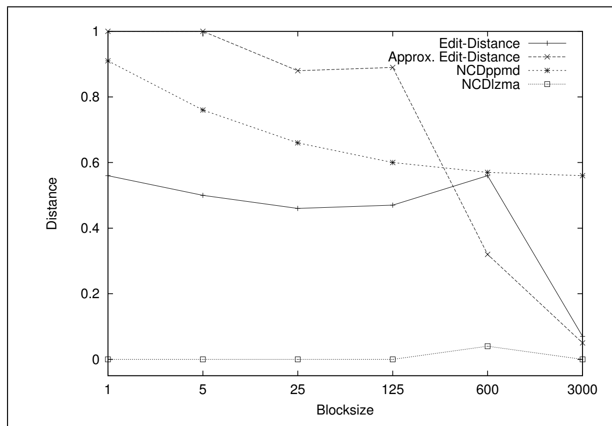

Fig. 3. Order sensitiveness of edit distance, approximated edit distance, NCD_{ppmd} and NCD_{lzma}

distance and NCD_{ppmd} show the expected behavior of being sensitive to reordering at a very low level but none of them seems to fit perfectly. NCD_{ppmd} drops much to slowly and a dissimilarity of 0.55 at a blocksize of 3000 is much to high. The edit distances has a high peak when using 600 lines. The approximated edit distance is doing quite well except for middle sized blocks (125 and 600) where the values (0.9 and 0.35) are too high. NCD_{lzma} is very order insensitive regardless of the size of the blocks that were shuffled.

The Manhattan distance (except for $n = 1$) shows the desired behavior of high distance for small block sizes and small distances for large block sizes. If n grows, the resulting

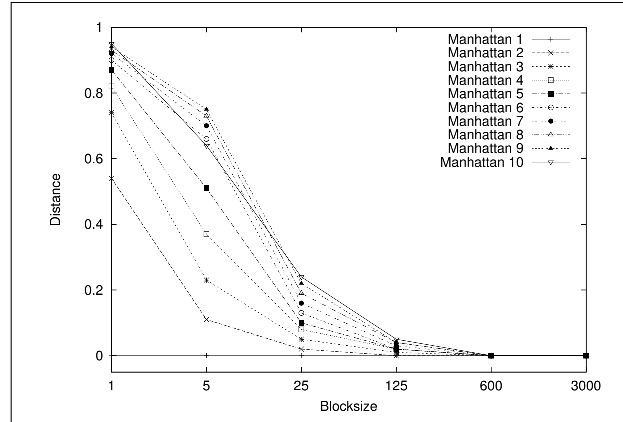

Fig. 4. Order sensitiveness of Manhattan distance

curve gets steeper. The differences between the curves are getting very small for $n > 5$ though. The usage of $n = 1$ results in an order insensitive distance measure.

C. Run-Time Performance Evaluation

The results (Fig. 5) show that *NCD* and the edit distance require a lot more computational resources than the approximated edit distance or the Manhattan distance. The resources needed by the Manhattan distance seem to correlate linearly to the size of the n-grams.

Distance	Time in s	Distance	Time in s
Edit	32611	Manhattan 4	308
Approx. Edit	113	Manhattan 5	365
NCD_{ppmd}	24144	Manhattan 6	406
NCD_{lzma}	56709	Manhattan 7	445
Manhattan 1	172	Manhattan 8	480
Manhattan 2	205	Manhattan 9	525
Manhattan 3	264	Manhattan 10	576

Fig. 5. Run-Time required for the distance measurement

The investigated distance measures can be divided into two groups regarding their run-time requirements. The first group contains the distances with high run-time requirements: the edit distance, NCD_{lzma} and NCD_{ppmd}. Although there are differences between those distances their run-times are considerably longer than those of group two which contains the Manhattan distance (regardless of n) and the approximated edit distance. The algorithms from group one should only be used if there are no bounds on the time that can be used for the computation of the distances.

VI. CONCLUSION

The following table summarizes our results. A minus sign (-) indicates that the respective criterion is not fulfilled while a plus sign (+) signifies fulfillment of a criterion. Signs are doubled, if the distance measure is particularly strong or weak regarding a criterion. Tab. I shows that the NCD variations and

Distance	Appropriate-ness	Ordersen-sitiveness	Perfor-mance
Edit	-	+	–
Approx. Edit	+	+	++
NCD_{ppmd}	–	-	–
NCD_{lzma}	–	–	–
Manhattan 1	++	–	++
Manhattan 2	++	+	++
Manhattan 3	++	++	++
Manhattan 4	++	++	++
Manhattan 5	++	++	++
Manhattan 6	++	++	+
Manhattan 7	++	++	+
Manhattan 8	++	++	+
Manhattan 9	+	++	+
Manhattan 10	+	++	+

TABLE I
SUMMARY OF DISTANCE MEASURES

the edit distance should not be used for the analysis of malware traces. Especially the NCD variation using lzma has various defects, which discourages its use in the analysis of malware traces. Based of our results we suggest to use the Manhattan distance along with $3 - grams$. The resulting vectors should be saved in tries or generalized suffix trees to allow efficient distance computations. A possible alternative would be to use*similarity coefficients* instead of the Manhattan distance. Similarity coefficients are only defined for binary vectors but can be extended to support vectors containing positive integers which is described in [38]. Their evaluation is nevertheless beyond the scope of this paper and planed for future work.

REFERENCES

[1] "Admmutate." [Online]. Available: http://www.securitylab.ru/_tools/ADMmutate-0.8.4.tar.gz

[2] D. Gryaznov and J. Telafici, "What a waste - the anti-virus industrie dos-ing itself," in *Proc. of Virus Bulletin Conf. 2007*, 2007, pp. 130–135.

[3] H. Flake, "Structural comparison of executable objects," in *Proc. of Dimva 2004*, ser. LNI, vol. 46. GI, 2004, pp. 161–173.

[4] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," in *Proc. of SSTIC2005*, Rennes, France, June 2005.

[5] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Proc. of RAID2005*, ser. LNCS, vol. 3858, 2005, pp. 207–226.

[6] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Proc. of Dimva 2006*, ser. LNCS, vol. 4064. Springer, 2006, pp. 129–143.

[7] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, "The Nepenthes platform: An efficient approach to collect malware," in *RAID*, ser. LNCS, no. 4219. Springer, 2006, pp. 165–184.

[8] "Amun: Python honeypot." [Online]. Available: http://amunhoney.sourceforge.net/

[9] "Capture-hpc client honeypot / honeyclient." [Online]. Available: https://projects.honeynet.org/capture-hpc

[10] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 32–39, 2007.

[11] International Secure Systems Lab, "Anubis: Analyzing unknown binaries," http://anubis.iseclab.org/.

[12] M. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *Journal in Computer Virology*, vol. 1, no. 1-2, pp. 13–23, 2005.

[13] S. Wehner, "Analyzing worms and network traffic using compression," *CoRR*, vol. abs/cs/0504045, 2005.

[14] D. Bruschi, L. Martignoni, and M. Monga, "Code normalization for self-mutating malware," *IEEE Security and Privacy*, vol. 5, no. 2, 2007.

[15] G. Bonfante, M. Kaczmarek, and J.-Y. Marion, "An implementation of morphological malware detection," in *Proc. of EICAR 2008*, France, Laval, pp. 49–62.

[16] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. of ACSAC 2007*. IEEE Computer Society, 2007, pp. 421–430.

[17] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proc. IEEE Symp. on Security and Privacy*, 1996, pp. 120–128.

[18] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, 2007.

[19] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in *Proc. of RAID 2007*, 2007.

[20] T. Lee and J. J. Mody, "Behavioral classification," in *Proc. of EICAR 2006)*, 2006.

[21] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Proc. of DIMVA 2008*, 2008, pp. 108–125.

[22] A. J. Oliner, A. Kulkarni, and A. Aiken, "Community epidemic detection with syzygy."

[23] "Amsel project." [Online]. Available: http://ls6-www.cs.tu-dortmund.de/ meier/AMSEL/

[24] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, vol. 18, no. 6, pp. 341–343, 1975.

[25] ——, "Algorithms for the longest common subsequence problem," *Journal of the ACM*, vol. 24, no. 4, pp. 664–675, 1977.

[26] J. D. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, no. Supplement-1, pp. 91–97, 2006.

[27] "Fuzzy hashing and ssdeep." [Online]. Available: http://ssdeep.sourceforge.net/

[28] M. Li and P. Vitanyi, *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, 1997.

[29] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi, "The similarity metric," in *Proc. of the 14th Annual ACM-SIAM Symp. on Discrete Algorithms*, 2003, pp. 863–872.

[30] R. Cilibrasi and P. Vitanyi, "Clustering by compression," 2004.

[31] E. Keogh, S. Lonardi, and C. A. Ratanamahatana, "Towards parameter-free data mining," in *Proc. 10th ACM SIGKDD*. ACM Press, 2004, pp. 206–215.

[32] A. Bratko, G. V. Cormack, B. Filipič, T. R. Lynam, and B. Zupan, "Spam filtering using statistical data compression models," *Journal of Machine Learning Research*, vol. 7, pp. 2673–2698, 2006.

[33] S. J. Delany and D. Bridge, "Feature-based and feature-free textual cbr: A comparison in spam filtering," in *Proc. of AICS'06*, 2006, pp. 244–253.

[34] M. Li and R. Sleep, "R.: Genre classification via an lz78-based string kernel," in *Proc. of ISMIR 2005*, 2005, pp. 252–259.

[35] M. Nykter, O. Yli-harja, and I. Shmulevich, "Normalized compression distance for gene expression analysis," in *Proc. of GENSIPS*, 2005.

[36] X. C. Brent, X. Chen, B. Francia, M. Li, B. Mckinnon, and A. Seker, "Shared information and program plagiarism detection," *IEEE Trans. on Information Theory*, vol. 50, pp. 1545–1551, 2004.

[37] M. Cebrián, M. Alfonseca, and A. Ortega, "Common pitfalls using the normalized compression distance: what to watch out for in a compressor," *Communications in Information and Systems*, vol. 5, no. 4, pp. 367–384, 2005.

[38] K. Rieck and P. Laskov, "Linear-time computation of similarity measures for sequential data." *Journal of Machine Learning Research (JMLR)*, vol. 9, pp. 23–48, 2008.

[39] Sneath and Sokal, *Numerical Taxonomy*. Freeman, 1973.

[40] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[41] C. T. Zahn, "Graph-theoretical methods for detecting and describing gestalt clusters," *Transactions on Computers*, vol. C-20, no. 1, pp. 68–86, 1971.