12-20-2009

Malware Recognition by Properties of Executables

Cory Redfern
University of New Orleans

Malware Recognition by Properties of Executables

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

by
Cory Redfern
B.S., Western Illinois University, Illinois, 2004
December, 2009

Table of Contents

Acknowledgments

I wish to thank Dr. Bilar for his insight and for helping turn my crazy ideas into something that might be useful. I wish to thank Dr. Winters-Hilt for politely explaining why my first two ideas were not going to work. I wish to thank Dr. Richard III for taking a kid with no sense of reality and producing a successful work in progress. I wish to thank Dr. Niño for beating me over the head with the Knowledge Stick, twice (metaphorically speaking). And to all the faculty and staff who have ever dealt with me anywhere, I offer my profound and heartfelt apologies.

Thanks to Matteo Cantoni of nothink.org for providing malware samples; also thanks to Netlux.org for same. Thanks to all my colleagues who have been here before for their advice. Also, once more, thanks to Dr. Bilar for pointing me in the right direction every time I got lost.

List of Figures

Abstract

This thesis explores what patterns, if any, exist to differentiate non-malware from malware, given only a sequence of raw bytes composing either a received file or a fixed-length initial segment of a received file. If any such patterns are found, their effectiveness as filtering criteria is investigated.

Keywords: affinity propagation, malware, filter

Introduction

Background:
This work is inspired by Bilar's prior work on opcode frequency comparisons between samples of known non-malware and known malware [2].

Motivation:
Malware detection and defense remains an unsolved problem. Zero-day attacks are difficult to predict and counter. On the zero-day, a NIDS can only use the data in received packets to make a decision whether to run an executable. If a differentiating pattern exists in the raw data of a signal, a NIDS can predict whether it is malware with a specified probability of certainty.

Modern malware hides in slack space to keep the file size unchanged and defeats signature-based detection with polymorphic techniques. Both of these approaches result in statistical differences between a malicious and a non-malicious executable. If the nature of these differences can be found by static composition analysis, we can improve on current anti-malware technology.

Objectives:
The main objective of this work is to answer this question: can one construct a useful malware filter by employing established techniques for pattern recognition?

Approach:
This work consists of a sequence of experiments using the scientific method. By asking a question within a specific framework, performing analysis, and then answering the question based on the results, the researcher seeks first to present the truth and then to offer an interpretation of the truth.

Methodology: Sample Sets

Sample set selection: I used random sampling in sample sets 1 and 3, since I am looking for classifying patterns that exist in the set of all executables. Sample sets 2A and 2B show nonrandom sampling because they were built from sample set 1 to test a hypothesis. In the sample sets, only files whose class was known were used. In sample set 3, I assumed that samples caught by a honeypot are definitely malware. Elimination of samples in the malware subsets occurred due to corruption in many files that made them unreadable, but acceptably large quantities remained.

Sample set 1: Contains 1987 malware samples from a vx.netlux.org archive and 1425 samples of non-malware from the Vista system32 directory, for a total of 3412 files. The malware ranges in size from 4 bytes to 9,499,648 bytes and includes a mix of Win32, MS-DOS, and boot-sector malware, with an average size of approximately 95 KB.

Sample set 2A: Contains 1425 malware samples from a vx.netlux.org archive and 1425 samples of non-ware from the Vista system32 directory, for a total of 2850 files. From sample set 1, 558 samples of malware were removed according to the ratio 9:3:3:1:1:1, referring to the malware classes Virus:VirTool:Trojan:HackTool:Email:Constructor. 4 additional samples were removed from 4 of the smaller partitions.

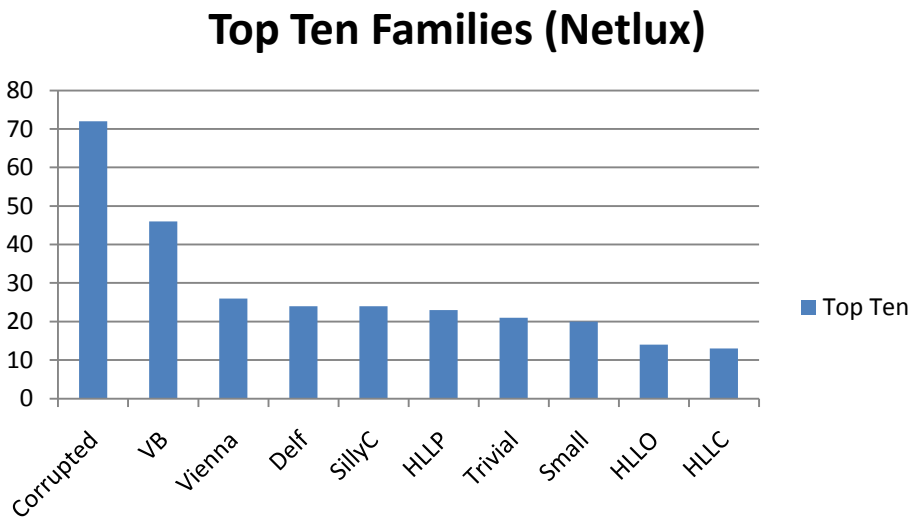
Sample set 2B: Contains 1425 malware samples from a vx.netlux.org archive and 1425 samples of non-malware from the Vista system32 directory, for a total of 2850 files. From sample set 1, 562 Virus class samples were removed.

Sample set 3: Contains 961 samples of malware from a honeypot provided by Matteo Cantoni of nothink.org and 1425 samples of non-malware from the Vista system32 directory, for a total of 2386 files. The malware ranges in size from 11 bytes to 4,383,744 bytes, with an average size of approximately 167 KB.

Sample Set 1: Malware Makeup

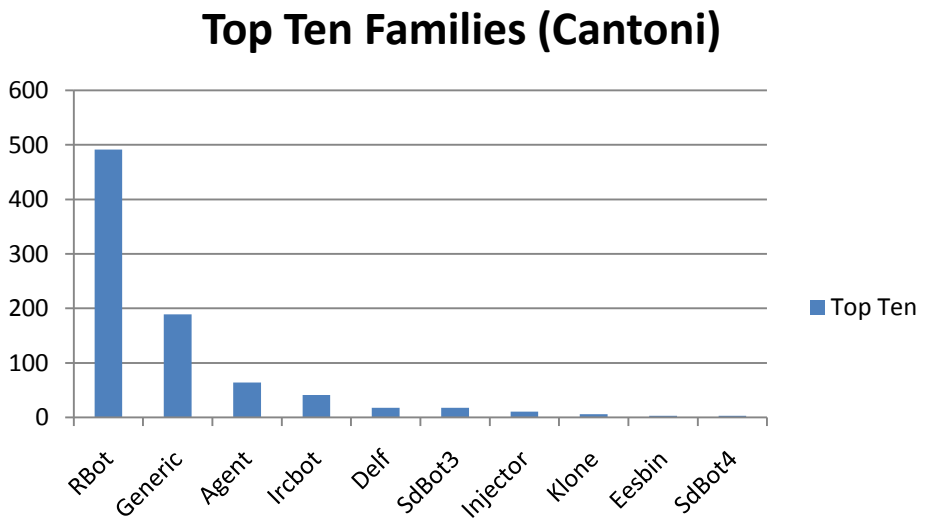
Sample set 1 contains 1987 files, with 1203 distinct family names. Some of the family names are different spellings of the same basic name; however, we are not completely certain which name groups can be merged, so we will not reduce this number by guessing.

Out of 1987 samples, the largest family accounted for 72 samples or about 4% of the total malware. The 2nd largest accounted for about 2% and there were 6 families that each accounted for 1%. The following bar graph shows the statistics:



Sample Set 3: Malware Makeup

Sample set 3 contains 961 files. Since they are named by hash codes, I ran AVG on a copy of the archive to get their true names. AVG found 866 threats, of which 151 were unique; of those 151, there were 24 unique families. 2 of the files contained a threat found in an alternate data stream; 97 files were found without contained threats. RBot was the most prevalent with 491 occurrences. The following bar graph shows the top 10 families:



Sample set 3 shows the nature of malware distribution in the wild; the most prevalent species captured are reflected by these skewed statistics. Around March 2009, when these samples were captured, most of these families were recognized by the Virus Bulletin (www.virusbtn.com). The reported overall prevalence of each type differs from this chart, since this only refers to one honeypot system, but most of the malware captured here was also seen in the Virus Bulletin reports.

Methodology: Techniques and Tools

Data Tabulation Type 1: For each sample, 256 features are extracted, corresponding to the number of times each possible byte value appears in the file, using the TableMaker utility.

Data Tabulation Type 2: For each sample, 256 features are extracted, corresponding to the number of times each possible byte value appears after a 0x0F byte in the file, using the OpcodeTableMaker utility.

Sample Size Normalization: Before performing clustering or other operations, each byte count is divided by the total number of bytes in the associated sample, using the TableNormalizer utility.

Similarity Preprocessing: The similarity metric used is the Euclidean distance. The apCluster utility requires a file containing a similarity measure for each pair of tuples in the set, which is provided by the APSimilarityPreprocessor utility.

Preference Preprocessing: This is set to the default preference -15.561256 for each tuple in a given set, so that all samples initially have the same chance to become a representative sample. The value -15.561256 was chosen by the authors of the apCluster utility.

Clustering: Affinity propagation, a method proposed by Frey and Dueck, is employed using their apCluster utility. The following section explains affinity propagation in detail.

Hex Analysis: HxD by Maël Hörz was used; it played a small but key role.

Methodology: Affinity Propagation

In a nutshell, affinity propagation is a data clustering technique that does not require any initial estimation external to the given data; it treats all data points as potential representative samples and uses iterative message passing to select the right ones. This gives it an advantage over algorithms like k-means that usually have to be run multiple times with different initial guesses.

When the 2007 paper is quoted in this section, the term “exemplar” is defined to mean “representative sample”.

Implementations of Frey & Dueck’s affinity propagation algorithm require a file containing a similarity measure for each sample pair and a file containing an initial preference for each sample. Implementations ensure that a partitioning of the sample set is generated in accordance with the algorithm specification.

At the start of execution, the chosen preferences give the likelihood for each data point to become a representative sample. If all preferences are equal at initialization, then all data points are equally likely to be representative samples. In this work, all preferences were initialized to the default value of -15.561256; this value was used in the sample preferences file that came with the apCluster utility provided by the authors of the 2007 paper.

At the start of execution, the initial similarity measures have the following format: (i, k, s(i, k)). The s(i, k) refers to “how well the data point with index k is suited to be the exemplar for data point i.”

During execution, the values of s(i, k) are updated by message passing. Two types of messages are passed: the “responsibility” r(i, k) and the “availability” a(i, k). The message r(i, k) is sent from point i to point k, telling k the “accumulated evidence for how well-suited point k is to serve as the exemplar for point i, taking into account other potential exemplars for point i.” The message a(i, k) is sent from point k to point i, telling i “the accumulated evidence for how appropriate it would be for point i to choose point k as its exemplar, taking into account the support from other points that point k should be an exemplar.” Initially, all availability values are zero.

The values r(i, k) at each iteration are computed as s(i, k) – max {a(i, k') + s(i, k')} where the max function is calculated over all k' such that k' ≠ k. This means that for each k' other than k, its availability to and similarity with i are added to obtain the total attractiveness of k', and the highest such value is subtracted from the similarity of i with k to get the net attraction of i to k. For the self-responsibility r(k, k), we perform s(k, k) – max {a(k, k') + s(k, k')}; that is, the input preference minus the highest attractiveness from k to one specific k'.

The values a(i, k) at each iteration are computed as min {0, r(k, k) + sum (max {0, r(i', k)}) } where the sum function is calculated over all i' such that i' ≠ i or k. This means we sum all the positive external responsibility values, add that to the self-responsibility, and use the minimum of the result and zero. Thus, a(i, k) <= 0 for all (i, k). The values a(k, k) are computed as sum (max {0, r(i', k)}) where i' ≠ k.

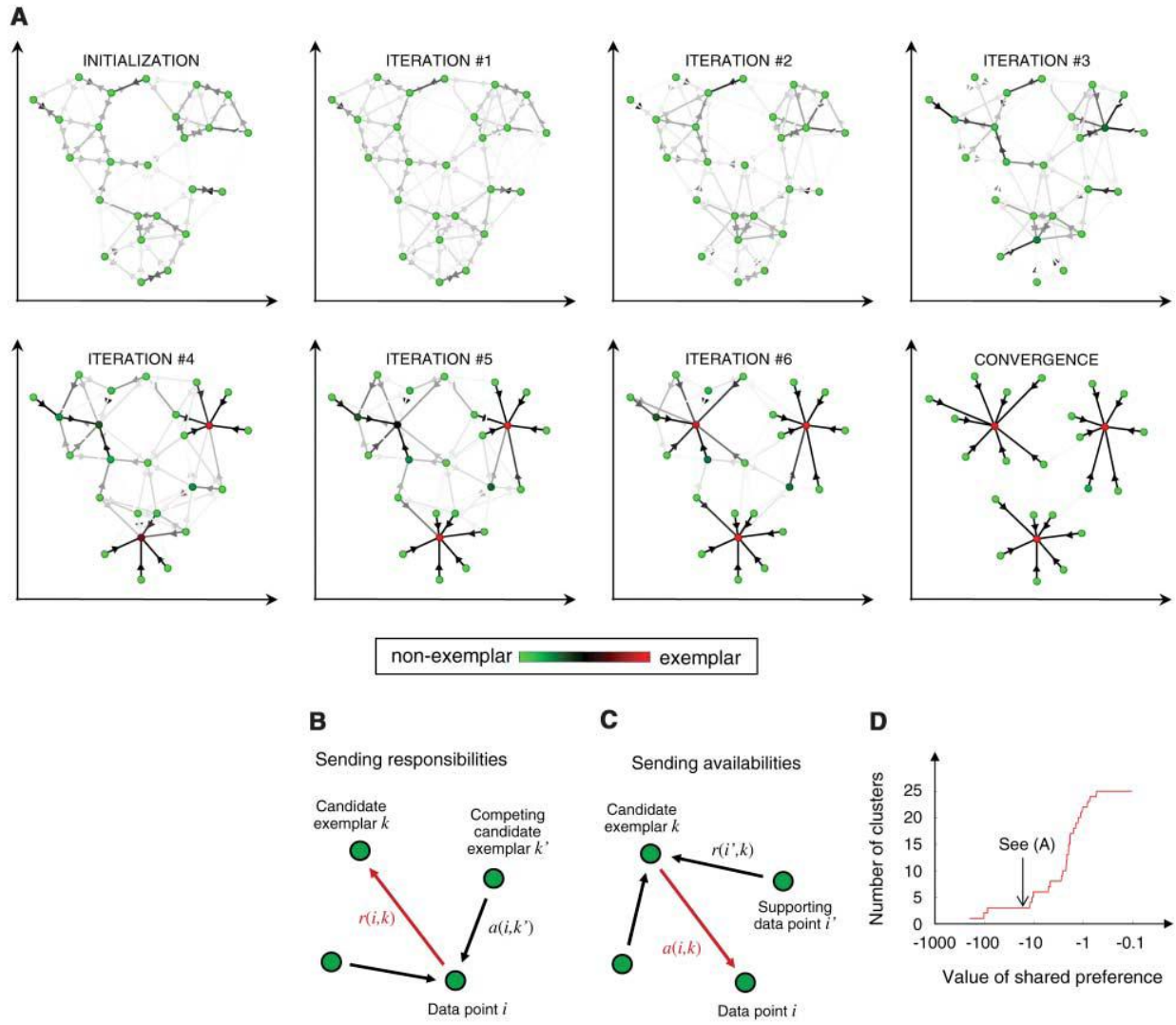
In sum: self-availability is an aggregate of a data point's positive responsibility values regarding all other points; self-responsibility is the difference between the input preference and the maximum attractiveness of all other data points; attractiveness is the sum of availability and similarity; availability is the minimum of zero and the sum of the point's self-responsibility with the aggregate of its positive external responsibility values; and responsibility is the difference between the similarity of one point to another point and the maximum attractiveness of all data points other than the two chosen. The calculations of a point's responsibility affect its availability, and vice versa. Each calculation requires information spanning the data set; thus, at each iteration, each point's values are updated according to the point's relationships with all other points.

Quoted text in this section comes from [1]. Everything in this section was written while consulting [1].

In the next section, an example from the original paper is shown.

Methodology: Affinity Propagation Example

This example, taken from [1], shows the iterations of affinity propagation in a simple data set:



Methodology: Affinity Propagation Example

In the previously shown example, part A illustrates the iterative process of electing the representative samples. Parts B and C illustrate the message-passing process: members tell each candidate how likely they are to vote for them, and candidates tell each member why they should be the representative. After much discussion, the representatives are elected.

Part D illustrates the way AP converges. The example data set had 25 points, so initially 25 clusters existed. There is no purpose in obtaining either 1 point per cluster or 1 cluster containing all points, so the algorithm finds the non-singleton set of representative samples with the largest range in its shared preference.

Experiment 1

Question: What is the natural clustering of the data set based on the byte counts of the sample files?

Materials: Sample set 1, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
cliconfg.exe	Good	214	45	21%
deploytk.dll	Good	332	210	63%
dvdupgrd.exe	Good	529	387	73%
netsh.exe	Good	400	251	63%
ntoskrnl.exe	Good	810	519	64%
Trojan.Win32.Bancos.j	Mal	921	909	99%
Virus.MSExcel.Feeder	Mal	206	205	100%

Experiment 1 Analysis

In this experiment, it is observed that the two representative malware samples collected only 13 non-malware samples between them, indicating high specificity. Out of 1987 samples, 1114 were captured by these representative samples, totaling about 56% of the malware; this is a low sensitivity.

The high specificity of the malware classification gives positive evidence for the 1-gram count classification of signals. It is possible that these values result from having more malware than non-malware. We can test this by using a sample set with a 1:1 ratio and a sample set with significantly less malware.

The next experiment will test clustering with two different schemes for reducing sample set 1 to a 1:1 ratio.

Experiment 2A

Question: If the number of malware samples is reduced to equal the number of non-malware samples, will the classifying specificity of malware change?

Tools: Sample set 2A, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
dvdupgrd.exe	Good	639	502	79%
lprmonui.dll	Good	385	299	78%
ntoskrnl.exe	Good	751	521	69%
Email-Worm.VBS.Brit	Mal	188	95	51%
Net-Worm.Win32.Randon	Mal	887	877	99%

Experiment 2B

Question: If the number of malware samples is reduced to equal the number of non-malware samples, will the classifying specificity of malware change?

Tools: Sample set 2B, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
dvdupgrd.exe	Good	678	545	80%
regwiz.exe	Good	265	188	71%
serialui.dll	Good	342	255	75%
Trojan-Dropper.Win32.Delf.dp	Mal	672	274	41%
VirTool.DOS.Apiary	Mal	778	773	99%
VirTool.Win32.NPE	Mal	115	81	70%

Experiment 2 Analysis

The results indicate a reduction in the malware classification specificity and an increase in the non-malware classification specificity after reducing the ratio to 1:1. Despite this, one of the malware representative samples served very well in each reduction scheme. The next experiment will determine whether that event occurs in a different sample set with a malware:non-malware ratio that is less than 1.

Experiment 3

Question: If a different, smaller, set of malware samples is combined with the same set of non-malware samples, will the trend in representative malware samples occur again?

Tools: Sample set 3, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	**Class**	**# files in cluster**	**# files in same class**	**Classifying specificity**
blackbox.dll	Good	626	475	76%
deploytk.dll	Good	306	273	89%
dvdupgrd.exe	Good	453	437	96%
regwiz.exe	Good	301	233	77%
370c2cc8ec1948f1cbbccb9ec58d18ed	Mal	700	693	99%

Experiment 3 Analysis

In experiment 2, the amount of malware was reduced relative to the amount of non-malware, and the results showed a decrease in the specificity of malware and an increase in the specificity of non-malware representative samples. Since that effect happened by two different reduction schemes, it seemed to represent a trend. Experiment 3 shows that trend in the non-malware representative samples. However, experiment 3 was done with even fewer malware samples, yet the lone malware representative sample had 99% specificity. **This resistance to the trend must be the result of the differences between the Cantoni archive and the Netlux archive. This means that modern malware shares a strong boundary with modern non-malware.** It also provides preliminary evidence that a classifying pattern exists in the frequency histogram of 1-gram features.

Since a pattern seems to have been found, the next question is whether the pattern holds for sample segments of a maximum length. It would be almost pointless to design a filter that requires the entire signal to be received, assembled, and fully counted before deciding what it is. If it can be determined from the first packet, though, then the filter can reject the rest of it.

Experiment 4

This experiment determines the classifying specificity of representative samples with a fixed upper bound on the histogram size. For each of the 4 sample sets used in previous experiments, the count is stopped at 256, 512, 1024, 2048, and 4096 bytes, resulting in 20 cluster sets that are then analyzed just like before.

Experiment 4A uses sample set 1. Experiment 4B uses sample set 2A. Experiment 4C uses sample set 2B. Experiment 4D uses sample set 3.

The data for this section is found in Appendix B; a detailed analysis follows next.

Experiment 4 Analysis

The previously defined trend was for malware representative samples to get worse and for non-malware representative samples to get better at classifying as the ratio of malware to non-malware decreased. Experiment 3 showed that something in sample set 3 had changed that trend, causing an improvement in the classifying specificity of both types.

In experiments 1, 2, and 3, the set of malware representative samples tended to have one member with high sensitivity relative to the uniform expectation. Experiment 4 showed that sample sets 1, 2A, and 2B continued this trend for each of the 5 segment bounds, but sample set 3 did not.

For quantification, out of each result set from the 5 experiments under the 4C heading, the size of the malware-exemplified cluster having the greatest combination of specificity and size was accumulated, and the result was averaged; the sizes were 729, 786, 800, 770, and 738, with a truncated average of 764. Thus, while maintaining high specificity, these five Netlux malware representative samples attracted 27% of sample set 2B on average, compared with 27% attracted by Apiary in experiment 2B. However, in experiment 4D the variables in question were filled by the values 327, 333, 390, 495, and 482, averaging to 405 or 17% of sample set 3. Compared with the value 29%, or 700 out of 2386, from experiment 3, this is a significant decrease in size.

The only change between executions of the apCluster utility was the underlying byte counts, and the counts used in experiment 4 are based on the same files counted in the previous corresponding experiments. Experiment 3 and experiment 4D each show a trend that differs from the trend seen in experiments 1 and 2 and in experiments 4A, 4B, and 4C, respectively. The sample sets behind those experiments differed from the sample set behind experiments 3 and 4D only in the source of the malware samples. Taken together, these facts indicate a fundamental difference between samples from Netlux and samples from Cantoni's honeypot as pertains to the objective of this work. This means one of the sets must be chosen, excluding the other, when doing analysis to create a malware filter. I chose to focus on the newer Cantoni archive from here on.

Experiment 5

Sample set: I used the size-normalized representative sample composition tuples from the experiments on the Cantoni archive.

Question: Do the measured differences among the natural clusters represent real properties of the underlying samples?

Tools: MS Excel

Procedure:

1. Create a copy of the sizenorm file for each sample table.
2. From each copy, remove all but the previously determined representative samples.
3. Build a record of the 256 components of the Euclidean distance for each pair of representative samples, normalized so that the row sum is 1 for each row.
4. Using MS Excel, put together a spreadsheet and investigate connections between the statistical relationships and real-world properties of the representative sample sets.

Experiment 5 Analysis

When a principal component analysis was done on the Euclidean distance elements, the majority of the representative sample pairs differed primarily in the ratio of 0x00 bytes in the files. Given that a uniform distribution over 256 features would yield less than 0.005 across the board, a value larger than 0.10 is substantial. For most representative sample pairs, the ratio of the difference between the 0x00 counts and the sum of all differences between counts was above 0.90. The principal component of the distance between representative samples was the 0x00 feature in 50 out of 52, or 96%, of the pairings within the 6 representative sample sets found in experiments 3 and 4D.

When considering executable files, the appearance of 0x00 signifies one of the following cases: the ADD Eb,Gb opcode; the 2-byte SLDT opcode; the end of a character string; empty bytes in a numeric value; slack space in a sector; or a byte within a random-equivalent file segment, i.e. a packed section or an embedded JPG. The following facts are known: that some malware makes use of SLDT during its operation, and that using slack space at the ends of sectors is a common infection technique. Aside from SLDT use and slack infection, there seems to be no transparent explanation for the 0x00 phenomenon. We will focus on the possibility of SLDT use as a classifying property.

To investigate the effect of SLDT instructions on malware classification, as well as the classifying properties of extended opcodes in general, we will count 2-byte opcodes in sample set 3, employing the 5 count limits used in experiment 4D in addition to counting entire files.

Experiment 6

Assumptions: According to the Intel x86 opcode reference, all 2-byte opcodes begin with 0x0F as the first byte; thus, when counting the number of 2-byte opcodes I simply retained the 256-feature model and only counted the number of times each byte appeared after 0x0F. This method shows errors in cases where a 0x0F value appears in a non-opcode context. I chose to consider the existence of this error margin in the analysis rather than to try removing its influence.

Materials: Sample set 3, OpcodeTableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP.

Procedure 1:

1. Run OpcodeTableMaker on the sample archive
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters
7. Repeat 1-6 for max byte counts of 256, 512, 1024, 2048, and 4096.

Procedure 2: We perform exp. 5 on the opcode count table.

Results: AP did not converge for the 256, 512, and 1024 byte cases.

Full file case:

Representative sample	**Class**	**# files in cluster**	**# files of same class**	**Classifying specificity**
6821bb6c7c39735854ec71afa727df0e	Mal	1127	907	80%
deploytk.dll	Good	704	651	92%
dmdskmgr.dll	Good	416	416	100%
kbdsp.dll	Good	75	75	100%
MP43DMOD.dll	Good	64	63	98%

Experiment 6 Results (continued)

2048 byte case:

Representative sample	**Class**	**# files in cluster**	**# files of same class**	**Classifying specificity**
5e1247a6bdb42424a806d25ac24626b5	Mal	90	85	94%
8bc77c747cc8fdb738736b339ffd5818	Mal	148	138	92%
5943eea30260fb1e36ee199867205b05	Mal	138	84	61%
8842eabe965c0b94d7ba54ed65c125b5	Mal	105	58	55%
65250e322c63df6e96a8cf448c2b0f9b	Mal	86	73	85%
aa5dbda6ad99ccdceb2e9cc3e750f28e	Mal	128	116	91%
catsrvps.dll	Good	153	151	99%
hlink.dll	Good	139	98	71%
kbdno.dll	Good	58	58	100%
napstat.exe	Good	141	141	100%
netevent.dll	Good	95	89	94%
PortableDeviceWiaCompat.dll	Good	1060	702	66%
smlogcfg.dll	Good	45	45	100%

4096 byte case:

Representative sample	**Class**	**# files in cluster**	**# files of same class**	**Classifying specificity**
1b525361c2435f0b9ca313771b9d82b4	Mal	90	71	79%
1fe325940d5f28536d62bd4046b36bcf	Mal	128	65	51%
4e46e9f237b6187b745c0b2b5a7a972c	Mal	88	70	80%
95af9bdce8cdd936d3f250c896621e0c	Mal	92	82	89 %
531c6309f57f38e09bb3b9d21f371636	Mal	157	139	89%
asferror.dll	Good	1080	613	57%
kbdsw.dll	Good	66	66	100%
napstat.exe	Good	122	122	100%
sens.dll	Good	371	347	94%
shmgrate.exe	Good	92	52	57%
ssdpsrv.dll	Good	100	97	97%

Experiment 6 Analysis

In this experiment, unlike in previous experiments, the clustering algorithm did not converge in all cases, given the underlying data of 2-byte opcode counts. It converged for the whole-file case, the 2048-byte case, and the 4096-byte case.

From the clustering results, I found no additional information that would be useful in malware filtering.

When I isolated the representative samples and performed exp. 5 again, I discovered that 0x00 was the principal component in all cases again, followed by 0x01 as the second component in the limited count cases. These opcodes signify two instruction groups that can be used by malware to detect a virtual machine.

The next step is to measure the raw counts in all samples associated with these 2 principal components.

Experiment 7

Question: How well are malware and non-malware separated by the sum of the 0x0F00 and 0x0F01 counts in modern malware?

Materials: Sample set 3 tables, MS Excel

Procedure:

1. Copy each opcode count table to an Excel spreadsheet.
2. Add the first two counts into a sum column and sort by the sum.
3. Split a copy of the table into these 3 classes: sum is 0, sum is 1, sum is > 1.
4. Calculate the amount of malware present in each class.
5. Repeat 1-4 for all 6 cases; cluster convergence is not a factor here.

Results:

Exemplar set	**Sum 0**	**Sum 1**	**Sum > 1**
3.256	23% specificity 37% sensitivity	67% specificity 46% sensitivity	99% specificity 15% sensitivity
3.512	5% specificity 6% sensitivity	71% specificity 75% sensitivity	92% specificity 17% sensitivity
3.1024	5% specificity 6% sensitivity	71% specificity 73% sensitivity	80% specificity 19% sensitivity
3.2048	5% specificity 6% sensitivity	66% specificity 72% sensitivity	67% specificity 20% sensitivity
3.4096	7% specificity 6% sensitivity	54% specificity 54% sensitivity	58% specificity 38% sensitivity
3.FULL	31% specificity 3% sensitivity	65% specificity 25% sensitivity	55% specificity 70% sensitivity

*Note: Due to truncation, the sum of sensitivities across classes is less than 100%

Experiment 7 Analysis

The results clearly show a statistical property of malware capable of separating it from non-malware: when the first 256 bytes of each sample were counted, 147 samples showed 2 or more occurrences of 0x0F00 or 0x0F01, and 146 of those samples were malware. With 961 malware samples in total, we have 15% sensitivity and 99% specificity.

In terms applicable to a practical filter, this experiment seems to suggest that a NIDS can filter out 15% of malware currently in the wild, with a 1% FP rate, simply by scanning the first packet once.

Unfortunately, there are 2 big problems with the results as they stand now. First, there is no executable code in the first 256 bytes of any file, so the occurrences of 0x0F00 and 0x0F01 cannot be explained by VM checking opcodes. Second, only 1 malware sample set was used to reach these conclusions.

In exp. 8, we will apply the exp. 7 procedure to the 1st sample set as an answer to the second problem. In exp. 9, we will answer the first problem by hex analysis.

Experiment 8

Question: How well are malware and non-malware separated by the sum of the 0x0F00 and 0x0F01 counts in older malware?

Materials: Sample set 1 tables, MS Excel

Procedure:

1. Copy each opcode count table to an Excel spreadsheet.
2. Add the first two counts into a sum column and sort by the sum.
3. Split a copy of the table into these 3 classes: sum is 0, sum is 1, sum is > 1.
4. Calculate the amount of malware present in each class.
5. Repeat 1-4 for all 6 cases.

Results:

Exemplar set	**Sum 0**	**Sum 1**	**Sum > 1**
1.256	55% specificity 76% sensitivity	67% specificity 22% sensitivity	94% specificity < 1% sensitivity
1.512	55% specificity 72% sensitivity	64% specificity 25% sensitivity	76% specificity 2% sensitivity
1.1024	53% specificity 62% sensitivity	64% specificity 26% sensitivity	82% specificity 10% sensitivity
1.2048	54% specificity 58% sensitivity	58% specificity 25% sensitivity	77% specificity 16% sensitivity
1.4096	60% specificity 54% sensitivity	50% specificity 22% sensitivity	62% specificity 22% sensitivity
1.FULL	92% specificity 44% sensitivity	69% specificity 12% sensitivity	41% specificity 43% sensitivity

*Note: Due to truncation, the sum of sensitivities across classes is less than 100%

Experiment 8 Analysis

The results based on sample set 1 show the same high specificity of malware for the sum class "> 1" in the 256 byte case, but the sensitivity is low; only 18 malware samples out of 1987 were captured. The trend as the count limit increases is jumpy as well. Considering also that the sensitivity of malware in the sum class "0" is 76%, along with the nature of the sample set, the age range of the malware is probably the cause of these differences.

The differences between the results of these 2 experiments indicate physical evidence of malware evolution. Modern malware instances, such as those that appear in Cantoni's archive, have a higher likelihood of containing VM checks than older instances, as evidenced by this data.

Experiment 9

Question: What is the reason for the occurrences of 0x0F00 and 0x0F01 in the first 256 bytes of an executable file?

Materials: Sample set 1, sample set 3, HxD

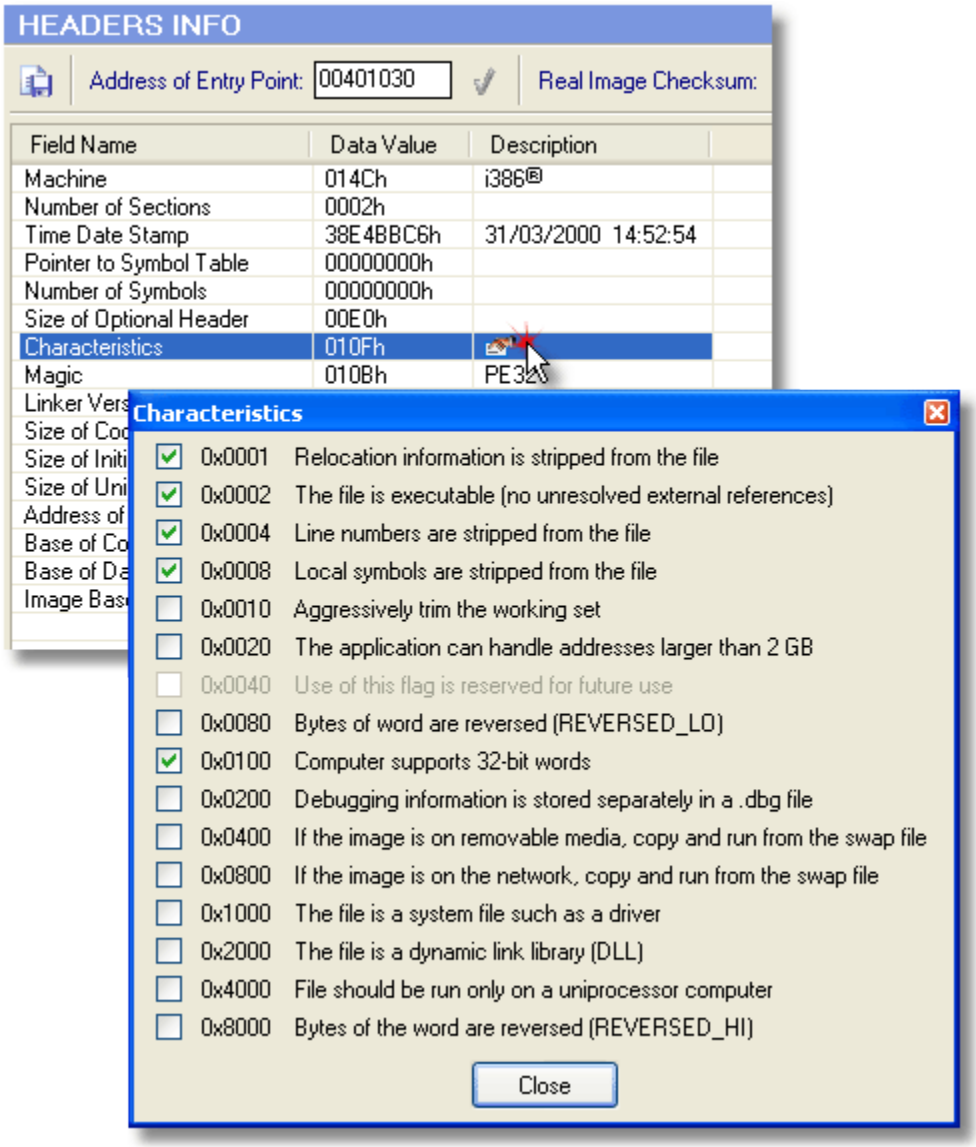
Procedure: Use HxD to look for the named occurrences in the file headers.

Results: The sequence 0x0F01 seems to be part of a semi-constant string in the PE header. In samples that fell into the sum class "> 1", this semi-constant string appeared within the first 256 bytes.

The part of the PE header that affects our statistics is the Characteristics field, a word containing 16 flags. When 0x0F01 is found, this means that only the 5 flags shown in the next illustration are set.

The Characteristics field does not serve to classify malware; the reason behind the statistic seen in exp. 7 is the size of the MZ header that directly precedes the PE header. If the MZ header is small, then the Characteristics field will appear within the first 256 bytes and it is usually 0x0F01; the other occurrence typically seems to happen by accident. If we assume that 1 occurrence has happened by accident, then the active classifier is the length of the MZ header. In fact, combining the MZ and PE headers is a known hex editing technique.

PE Characteristic Flags



This image is copied from [7]. Intel hardware stores words in reverse, so that the value 0x010F is stored as 0x0F followed by 0x01.

Final Conclusions

Based on this work, it is apparent that a pattern capable of filtering malware with a NIDS exists. The pattern is that the sum of 0x0F00 and 0x0F01 occurrences within the first 256 bytes is 2 or greater in 15% of modern executables, with 99% of those executables being malware.

The filter cannot be made complete with the current results, but it can be partially constructed.

Future Work

The future plan for using these results is first to construct a filter using the occurrences of 0x0F00 and 0x0F01. After that, more experiments with modern malware are necessary to increase the amount of malware that can be filtered correctly. Future work thus includes more experiments like exp. 7.

Future work also includes an algorithm for maximizing the sensitivity of malware captured by AP clustering and an investigation of 3-byte opcodes and fuzzy matching around the counted bytes.

Future work also includes the use of different distance metrics in the clustering of samples, such as the Kullback-Leibler divergence. KL divergence fits particularly well within this work since the size-normalized data is essentially a probability distribution.

References and Credits

[1] Frey, Brendan J., and Delbert Dueck. "Clustering by Passing Messages Between Data Points." *Science* 315.5814 (2007): 972-976. Web.
[2] Bilar, Daniel. "Opcodes as predictor for malware." *International Journal of Electronic Security and Digital Forensics* 1.2 (2007): 156-168. Web.
[3] W.J. Li, K. Wang, S.J. Stolfo, B. Herzog, “Fileprints: identifying filetypes by n-gram analysis”, IEEE Information Assurance Workshop, USA, IEEE Press, 2005.
[4] Tabish, S. Momina, M. Zubair Shafiq, and Muddassar Farooq. "Malware detection using statistical analysis of byte-level file content." *KDD Workshop on CyberSecurity and Intelligence Informatics*. Hsinchun Chen and Marc Dacier and Marie-Francine Moens and Gerhard Paass and Christopher C. Yang: Paris, France, ACM. 2009. 23-31. Web.
[5] Intel x86 opcode reference
http://www.sandpile.org/ia32/index.htm
[6] The Virus Bulletin
http://www.virusbtn.com
[7] PE characteristic flags illustration
http://www.heaventools.com/PE-file-header_viewer.htm
[8] Vx.netlux.org: Malware samples pre-2008
[9] Matteo Cantoni, www.nothink.org: Malware samples 2009
[10] Windows Vista: Non-malware samples 2009

Appendix A: Glossary

Zero-day attack: the first appearance of a new malware strain
Slack space: empty space in a disk block immediately following valid file data
NIDS: Network Intrusion Detection System
Opcode: a byte or sequence of bytes referring to a valid instruction

AP: shorthand for affinity propagation
Attractiveness: the sum of a point's similarity and availability to another point
Exemplar: a representative sample out of a data set
Distance measures: metrics for determining how similar two samples are
Euclidean distance: a distance measure based on spatial closeness
Kullback-Leibler divergence: the distance between two probability distributions

Appendix B: Experiment 4 Data

Experiment 4A.256

Question: How is the cluster distribution different when only the first 256 bytes are counted?

Tools: Sample set 1, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 256
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
msvcrt20.dll	Good	400	202	51%
Constructor.DOS.DPOG.02.a	Mal	774	774	100%
Constructor.Win32.VCL	Mal	217	216	> 99%
Virus.BAT.IBBM.generic	Mal	248	248	100%
Virus.DOS.Corrupted.RCE-2772	Mal	25	25	100%
Virus.DOS.GCAE.x	Mal	188	171	91%
Virus.MSWord.Ping.f	Mal	107	107	100%
Virus.Win9x.CIH.dam	Mal	1453	248	17%

Experiment 4A.512

Question: How is the cluster distribution different when only the first 512 bytes are counted?

Tools: Sample set 1, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 512
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
msvbvm60.dll	Good	746	562	75%
HackTool.Win32.Wuftpd	Mal	291	244	84%
Trojan.BAT.FormatC.k	Mal	221	221	100%
Virus.Boot-DOS.ZhengZhou.3584.c	Mal	921	130	14%
Virus.DOS.Corrupted.RCE-2772	Mal	24	24	100%
Virus.DOS.Istanbul.1397	Mal	871	864	99%
Virus.DOS.SillyC.432.c	Mal	231	213	92%
Virus.MSWord.Sun	Mal	107	107	100%

Experiment 4A.1024

Question: How is the cluster distribution different when only the first 1024 bytes are counted?

Tools: Sample set 1, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 1024
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
sigverif.exe	Good	706	511	72%
Spoofer.Win32.VB.c	Mal	612	204	33%
Trojan.BAT.FormatC.t	Mal	218	218	100%
Trojan.Win32.ICQUkr	Mal	830	137	17%
Virus.DOS.Capicua.511	Mal	25	25	100%
Virus.DOS.Nado.Fatill.1336	Mal	914	897	98%
Virus.MSWord.Rimes	Mal	107	107	100%

Experiment 4A.2048

Question: How is the cluster distribution different when only the first 2048 bytes are counted?

Tools: Sample set 1, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 2048
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
hnetmon.dll	Good	735	524	71%
kbduk.dll	Good	312	198	63%
rcbdyctl.dll	Good	731	484	66%
VirTool.Win32.EnterRing0.b	Mal	414	209	50%
Virus.DOS.Corrupted.RCE-2772	Mal	28	28	100%
Virus.DOS.Nado.Fatill.1336	Mal	879	865	98%
Virus.MSExcel.Feeder	Mal	214	214	100%
Virus.MSWord.Rimes	Mal	99	99	100%

Experiment 4A.4096

Question: How is the cluster distribution different when only the first 4096 bytes are counted?

Tools: Sample set 1, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 4096
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
cleanmgr.exe	Good	639	498	78%
kbddv.dll	Good	377	249	66%
Backdoor.Win32.SubSeven.214	Mal	835	375	45%
VirTool.Win32.PSP95	Mal	475	276	58%
Virus.Boot-DOS.Kuarahy.4640	Mal	860	841	98%
Virus.MSWord.Tech.c	Mal	226	226	100%

Experiment 4B.256

Question: How is the cluster distribution different when only the first 256 bytes are counted?

Tools: Sample set 2A, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 256
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
msvcrt20.dll	Good	326	202	62%
Backdoor.Win32.Amitis.143	Mal	103	102	99%
Constructor.DOS.DPOG.02.a	Mal	867	867	100%
Trojan.Win32.Icqpush.b	Mal	1377	172	12%
Virus.DOS.Corrupted.RCE-2772	Mal	20	20	100%
Virus.DOS.GCAE.x	Mal	157	140	89%

Experiment 4B.512

Question: How is the cluster distribution different when only the first 512 bytes are counted?

Tools: Sample set 2A, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 512
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
ir50_32.dll	Good	927	724	78%
Virus.Boot-DOS.ZhengZhou.3584.c	Mal	777	102	13%
Virus.DOS.Corrupted.RCE-2772	Mal	19	19	100%
Virus.DOS.Istanbul.1397	Mal	936	929	99%
Virus.DOS.SillyC.432.c	Mal	191	172	90%

Experiment 4B.1024

Question: How is the cluster distribution different when only the first 1024 bytes are counted?

Tools: Sample set 2A, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 1024
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
kbdsl.dll	Good	451	213	47%
sigverif.exe	Good	662	511	77%
Trojan.Win32.ICQUkr	Mal	751	67	9%
Virus.DOS.Nado.Fatill.1336	Mal	986	969	98%

Experiment 4B.2048

Question: How is the cluster distribution different when only the first 2048 bytes are counted?

Tools: Sample set 2A, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 2048
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
hnetmon.dll	Good	637	522	82%
kbdhela2.dll	Good	274	199	73%
msvcp50.dll	Good	328	206	63%
rcbdyctl.dll	Good	685	485	71%
VirTool.DOS.Apiary	Mal	905	892	99%
Virus.DOS.SillyOC.2000	Mal	21	21	100%

Experiment 4B.4096

Question: How is the cluster distribution different when only the first 4096 bytes are counted?

Tools: Sample set 2A, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 4096
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
cleanmgr.exe	Good	605	520	86%
kbddv.dll	Good	315	249	79%
msvcp50.dll	Good	325	199	61%
HackTool.Win32.NetHacker	Mal	738	293	40%
VirTool.DOS.Apiary	Mal	867	855	99%

Experiment 4C.256

Question: How is the cluster distribution different when only the first 256 bytes are counted?

Tools: Sample set 2B, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 256
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
msvcrt20.dll	Good	382	202	53%
Trojan.BAT.KillFiles.at	Mal	202	201	> 99%
Trojan.Win32.Icqpush.b	Mal	1411	206	15%
Trojan-Dropper.Win32.Delf.cf	Mal	729	729	100%
Virus.DOS.GCAE.x	Mal	126	109	87%

Experiment 4C.512

Question: How is the cluster distribution different when only the first 512 bytes are counted?

Tools: Sample set 2B, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 512
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
Spoofer.Win32.VB.c	Mal	325	236	73%
Trojan-Dropper.Win32.Raven	Mal	778	160	21%
Virus.Boot-DOS.ZhengZhou.3584.c	Mal	786	93	12%
Virus.DOS.Corrupted.Eddie.Sign	Mal	175	157	90%
Virus.DOS.Istanbul.1397	Mal	786	779	99%

Experiment 4C.1024

Question: How is the cluster distribution different when only the first 1024 bytes are counted?

Tools: Sample set 2B, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 1024
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
msr2cenu.dll	Good	588	207	35%
sigverif.exe	Good	666	512	77%
Trojan.Win32.ICQUkr	Mal	796	106	13%
VirTool.DOS.WeirdBinder	Mal	800	784	98%

Experiment 4C.2048

Question: How is the cluster distribution different when only the first 2048 bytes are counted?

Tools: Sample set 2B, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 2048
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
hnetmon.dll	Good	713	524	73%
kbduk.dll	Good	297	198	67%
rcbdyctl.dll	Good	687	485	71%
VirTool.DOS.Apiary	Mal	770	757	98%
VirTool.Win32.EnterRing0.b	Mal	383	178	46%

Experiment 4C.4096

Question: How is the cluster distribution different when only the first 4096 bytes are counted?

Tools: Sample set 2B, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 4096
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
cleanmgr.exe	Good	553	469	85%
msvcp50.dll	Good	389	201	52%
regwiz.exe	Good	363	276	76%
Backdoor.Win32.SubSeven.214	Mal	807	340	42%
VirTool.DOS.Apiary	Mal	738	726	98%

Experiment 4D.256

Question: How is the cluster distribution different when only the first 256 bytes are counted?

Tools: Sample set 3, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 256
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	**Class**	**# files in cluster**	**# files of same class**	**Classifying specificity**
msacm.dll	Good	1449	1180	81%
msvcrt20.dll	Good	375	220	59%
88ced2768eba4b2b77372726850bdfad	Mal	327	320	98%
bdd14a3ccfa6c7162e425266bce0c729	Mal	235	217	92%

Experiment 4D.512

Question: How is the cluster distribution different when only the first 512 bytes are counted?

Tools: Sample set 3, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 512
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
23ddbaed383511a9968f53b7bdbbf2e9	Mal	781	17	2%
581a6310045be2711de6105d4cf3f354	Mal	883	309	35%
5b760960b279a54b984d5d9ccc9560b7	Mal	273	217	79 %
e732601321962a9810986d418e839ae2	Mal	333	320	96 %
ff20a4b54baaedf479805d44013bb443	Mal	116	98	84%

Experiment 4D.1024

Question: How is the cluster distribution different when only the first 1024 bytes are counted?

Tools: Sample set 3, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 1024
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
ctl3d32.dll	Good	916	601	66%
onex.dll	Good	560	535	96%
0526a33d93d1bc96d7ea3cfe20fed0bf	Mal	406	194	48%
235abf3acbb7b052e83a481f53a7c46b	Mal	114	99	87%
e92b09accfabaaaf0566444c7224e079	Mal	390	328	84%

Experiment 4D.2048

Question: How is the cluster distribution different when only the first 2048 bytes are counted?

Tools: Sample set 3, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 2048
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
hnetmon.dll	Good	633	572	90%
msvcp50.dll	Good	367	227	62%
2f1bfdfc7045e1ae0b70acd1c646e77c	Mal	495	404	82%
f6b8239db7a1b96fd9afd3df0725c13f	Mal	891	356	40%

Experiment 4D.4096

Question: How is the cluster distribution different when only the first 4096 bytes are counted?

Tools: Sample set 3, TableMaker, TableNormalizer, APSimilarityPreprocessor, APPreferencePreprocessor, apCluster, ClusterSplitterAP

Procedure:

1. Run TableMaker on the sample archive with a max byte count of 4096
2. Run TableNormalizer on the data
3. Run APSimilarityPreprocessor on the size-normalized data
4. Run APPreferencePreprocessor to set the preferences
5. Run apCluster using the calculated similarities and preferences
6. Run ClusterSplitterAP to place all the sample files in their correct clusters

Results:

Representative sample	Class	# files in cluster	# files of same class	Classifying specificity
netsh.exe	Good	446	432	97%
regwiz.exe	Good	248	243	98%
783128a871bfec9ebbd2e2595a5be2b7	Mal	482	447	93%
8dfac3855eb7f4cf36b7826e640c8c0e	Mal	336	135	40%
c1b1a396e2d9ad407fe00c1ec56c101b	Mal	874	360	41%

Appendix C: Source Code

TableMaker.java

```
import java.io.*;

class TableMaker
{
  public static void main(String[] args) throws Exception
  {
    int huge = 2000000000;

    File mainDir = new File(args[0]);
    File[] samples = mainDir.listFiles();
    FileInputStream input;

    int[] byteCounts = new int[256];
    String[] sampleTable = new String[samples.length];
    PrintWriter output;

    int readLength = huge;
    try
    {
      readLength = Integer.parseInt(args[1]);
      output = new PrintWriter(args[0] + "." + readLength + ".data");
    }
    catch(Exception e)
    {
      readLength = huge;
      output = new PrintWriter(args[0] + ".data");
    } // if a limit is provided as an argument, use it; otherwise the limit is ~2GB

    int holder;
    int filePosition;
    for(int i = 0; i < samples.length; i++)
    {
      try
      {
        input = new FileInputStream(samples[i]);
        for(int j = 0; j < 256; j++) byteCounts[j] = 0;

        filePosition = 0;
        holder = input.read();
```

```
      while(holder != -1 && filePosition < readLength)
      {
       holder = (holder + 256) % 256;
       byteCounts[holder] = byteCounts[holder] + 1;
       filePosition++;
       holder = input.read();
      } // read and count bytes until the limit or EOF is reached

      sampleTable[i] = samples[i].toString() + ",";
      for(int j = 0; j < 256; j++)
      {
       sampleTable[i] = sampleTable[i] + byteCounts[j] + ",";
      } // construct the 256-feature data line for the current sample

      sampleTable[i] = sampleTable[i].substring(0, sampleTable[i].length() - 1); // drop trailing ,
      output.println(sampleTable[i]); // print the line to the data file
     }
     catch(FileNotFoundException fnf)
     {
      System.out.println(samples[i]);
     } // if the file is not there, print its name to the console
    }
    output.close();
   }
}
```

TableNormalizer.java

```
import java.io.*;

class TableNormalizer
{
  public static void main(String[] args) throws Exception
  {
    File mainTable = new File(args[0]);
    String dataline = "";
    String[] tokens;
    double numTotalBytes;
    double normalizedByteRatio;
    String outputLine = "";

    BufferedReader console = new BufferedReader(new FileReader(mainTable));
    PrintWriter output = new PrintWriter(args[0] + ".sizenorm");

    dataline = console.readLine();
    while(dataline != null)
    {
      tokens = dataline.split(",");

      numTotalBytes = 0.0;
      for(int i = 1; i < 257; i++) // position 0 is the filename
      {
        numTotalBytes = numTotalBytes + Double.parseDouble(tokens[i]);
      } // get sum of counts for current line

      if(numTotalBytes = 0.0) numTotalBytes = 1.0; // required to avert DivZero errors
      outputLine = tokens[0] + ",";
      for(int i = 1; i < 257; i++)
      {
        normalizedByteRatio = Double.parseDouble(tokens[i]) / numTotalBytes;
        outputLine = outputLine + normalizedByteRatio + ",";
      } // normalize counts as fractions of the sum of counts
      outputLine = outputLine.substring(0, outputLine.length() - 1);
      output.println(outputLine);
      dataline = console.readLine();
    }
    output.close();
  }
}
```

APSimilarityPreprocessor.java

```
import java.io.*;

class APSimilarityPreprocessor
{
  public static void main(String[] args) throws Exception
  {
    File mainTable = new File(args[0]);
    String dataline = "";
    String[] tokens;
    double[] pointOne = new double[256];
    double[] pointTwo = new double[256];
    double similarity;

    double[][] dataPoints = new double[30000][256]; // recompile if more than 30000 samples
    int lastRowIndex = 0;

    BufferedReader console = new BufferedReader(new FileReader(mainTable));
    PrintWriter output = new PrintWriter(args[0] + ".similarities");

    dataline = console.readLine();
    while(dataline != null)
    {
      tokens = dataline.split(",");
      for(int i = 0; i < 256; i++)
      {
        dataPoints[lastRowIndex][i] = Double.parseDouble(tokens[i + 1]);
      }
      lastRowIndex = lastRowIndex + 1;
      dataline = console.readLine();
    }
```

```
  for(int i = 1; i <= lastRowIndex; i++)
  {
   pointOne = dataPoints[i - 1];
   for(int j = 1; j <= lastRowIndex; j++)
   {
    if(j != i)
    {
     pointTwo = dataPoints[j - 1];
     similarity = -1.0 * euclideanDistance(pointOne, pointTwo); // AP uses negative similarities
     output.println("" + i + " " + j + " " + similarity);
    }
   }
  } // calculate and store all pairwise distances

  output.close();
 }

 static double euclideanDistance(double[] pointOne, double[] pointTwo)
 {
  if(pointOne.length != pointTwo.length) return -1.0; // one should avoid letting this happen

  double sum = 0.0;
  double partDiff = 0.0;
  for(int i = 0; i < pointOne.length; i++)
  {
   partDiff = pointOne[i] - pointTwo[i];
   sum = sum + (partDiff * partDiff);
  }
  return Math.sqrt(sum);
 }
}
```

APPreferencePreprocessor.java

```
import java.io.*;

class APPreferencePreprocessor
{
  public static void main(String[] args) throws Exception
  {
    File mainTable = new File(args[0]);
    String dataline = "";

    BufferedReader console = new BufferedReader(new FileReader(mainTable));
    PrintWriter output = new PrintWriter(args[0] + ".preferences");

    dataline = console.readLine();
    while(dataline != null)
    {
      output.println("-15.561256"); // default preference; all are set equally
      dataline = console.readLine();
    }
    output.close();
  }
}
```

ClusterSplitterAP.java

```
import java.io.*;

class ClusterSplitterAP
{
  public static void main(String[] args) throws Exception
  {
    int arrayMax = 30000; // recompile if more than 30000 samples

    File clusterDefinitions = new File(args[0] + ".clusters/idx.txt");

    File dataTable = new File(args[0]);
    String[] fileNames = new String[arrayMax];
    BufferedReader console = new BufferedReader(new FileReader(dataTable));

    File workingFile;

    FileInputStream input;
    FileOutputStream output;

    String sourceFile;
    String clusterDirectory;
    String destinationFile;

    String dataline = "";
    String[] tokens;

    int positionCounter = 0;
    int slashIndex = 0;

    dataline = console.readLine();
    while(dataline != null)
    {
      tokens = dataline.split(",");
      fileNames[positionCounter] = tokens[0];

      positionCounter++;
      dataline = console.readLine();
    }
```

```
   console = new BufferedReader(new FileReader(clusterDefinitions));

   positionCounter = 0;
   dataline = console.readLine();
   while(dataline != null)
   {
    sourceFile = fileNames[positionCounter];
    destinationFile = fileNames[Integer.parseInt(dataline) - 1];

    slashIndex = destinationFile.indexOf("\\");
    destinationFile = destinationFile.substring(slashIndex + 1, destinationFile.length());

    clusterDirectory = args[0] + ".clusters/" + destinationFile + ".ap";
    // the directories are named for the representative samples
    try
    {
     workingFile = new File(clusterDirectory);
     workingFile.mkdir();
    }
    catch(Exception e) {} // if it's not there, make it; otherwise do nothing

    slashIndex = sourceFile.indexOf("\\");
    destinationFile = sourceFile.substring(slashIndex + 1, sourceFile.length());
    destinationFile = clusterDirectory + "\\" + destinationFile; // split around the \ and rebuild

    workingFile = new File(sourceFile);
    input = new FileInputStream(workingFile);

    output = new FileOutputStream(destinationFile); // place copies within correct clusters

    int holder = input.read();
    while(holder != -1)
    {
     output.write(holder);
     holder = input.read();
    } // copy file byte by byte
    output.close();
    dataline = console.readLine();
    dataline = console.readLine(); // IDX files are double-spaced
    positionCounter++;
   }
  }
}
```

EuclideanDistanceTabulator.java

```
import java.io.*;
// calculate and store 256 components of pairwise Euclidean distances for use in PCA
class EuclideanDistanceTabulator
{
  public static void main(String[] args) throws Exception
  {
    File mainTable = new File(args[0]);
    String dataline = "";
    String[] tokens;
    double[] pointOne = new double[256];
    double[] pointTwo = new double[256];
    double[] differences = new double[256];
    double diffLineTotal;

    double[][] dataPoints = new double[30000][256]; // recompile if more than 30000 samples
    int lastRowIndex = 0;

    BufferedReader console = new BufferedReader(new FileReader(mainTable));
    PrintWriter output = new PrintWriter(args[0] + ".differences");

    dataline = console.readLine();
    while(dataline != null)
    {
      tokens = dataline.split(",");
      if(tokens.length == 257) // ignore malformed rows
      {
        for(int i = 0; i < 256; i++)
        {
          dataPoints[lastRowIndex][i] = Double.parseDouble(tokens[i + 1]);
        }
        lastRowIndex = lastRowIndex + 1;
      }
      dataline = console.readLine();
    }
```

```
   for(int i = 1; i <= lastRowIndex; i++)
   {
    pointOne = dataPoints[i - 1];
    for(int j = 1; j <= lastRowIndex; j++)
    {
     if(j != i)
     {
      pointTwo = dataPoints[j - 1];
      if(pointOne.length == pointTwo.length)
      {
       dataline = "";
       diffLineTotal = 0.0;
       output.print("" + i + "," + j + ",");
       for(int k = 0; k < pointOne.length; k++)
       {
        differences[k] = pointOne[k] - pointTwo[k];
        differences[k] = differences[k] * differences[k];
        diffLineTotal = diffLineTotal + differences[k];
       } // get the sum of all components on the current line

       for(int k = 0; k < pointOne.length; k++)
       {
        differences[k] = differences[k] / diffLineTotal;
        dataline = dataline + differences[k] + ",";
       } // normalize each component against the line sum

       dataline = dataline.substring(0, dataline.length() - 1);
       output.print(dataline + "\n"); // output 256 normalized distance metric components
      }
     }
    }
   }

   output.close();
  }
}
```

Vita

Cory Redfern was born on August 2nd, 1983 in Marion, IL to Susan and Paul Redfern. He earned his B.S. in Computer Science and his B.A. in French from Western Illinois University in December 2004. After spending some time in the private sector, he returned to school at the University of New Orleans in August 2007 and earned his M.S. in Computer Science in December 2009.

For his research, he worked under Dr. Daniel Bilar.

His interests currently include digital forensics, cryptography, quantum computation, and number theory.