SYM-AM-21-082

EXCERPT FROM THE PROCEEDINGS OF THE EIGHTEENTH ANNUAL ACQUISITION RESEARCH SYMPOSIUM

Using Texture Vector Analysis to Measure Computer and Device File Similarity

May 11–13, 2021

Published: May 10, 2021

Prepared for the Naval Postgraduate School, Monterey, CA 93943.

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF DEFENSE MANAGEMENT
NAVAL POSTGRADUATE SCHOOL

The research presented in this report was supported by the Acquisition Research Program of the Graduate School of Defense Management at the Naval Postgraduate School.

To request defense acquisition research, to become a research sponsor, or to print additional copies of reports, please contact any of the staff listed on the Acquisition Research Program website (www.acquisitionresearch.net).

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF DEFENSE MANAGEMENT
NAVAL POSTGRADUATE SCHOOL

Using Texture Vector Analysis to Measure Computer and Device File Similarity

Bruce Allen—is a Faculty Associate at the Naval Postgraduate School where he performs research and supports students in various topics relating to computer science. His specialties include data analytics, robotics, cyber security, digital forensics, and GUI design. [bdallen@nps.edu]

Abstract

Executable programs run on computers and digital devices. These programs are pre-installed by the device vendor or are downloaded or copied from a storage media. It is useful to study file similarity between executable files to verify valid updates, identify potential copyright infringement, identify malware, and detect other abuse of purchased software.

An alternative to relying on simplistic methods of file comparison, such as comparing their hash codes to see if they are identical, is to identify the "texture" of files and then assess its similarity between files. To test this idea, we experimented with a sample of 23 Windows executable file families and 1,386 files. We identify points of similarity between files by comparing sections of data in their standard deviations, means, modes, mode counts, and entropies. When vectors are sufficiently similar, we calculate the offsets (shifts) between the sections to get them to align. Using analysis on these shifts, we can measure file similarity efficiently. By plotting similarity vs. time, we track the progression of similarity between files.

Introduction

Software of unknown pedigree abounds. This is partly due to software being distributed as executable code or a "binary," and evaluating the contents of a binary is technically challenging.

Numerous updates to a binary can occur over the useful life of the executable to address new software requirements, fix software defects, or port the software to a different computing platform. Each of these requires recompilation and results in a new binary.

Executable code can be analyzed using reverse-engineering tools that recover information about the binary's structure, function, and behavior. Some tools recognize data regions inside the code, while more advanced tools analyze the machine instructions to make inferences about the code's function. Because of the differences in instruction set architectures (ISAs), tools use models of ISAs. However, reverse engineering of a binary can be resource-intensive and can be stymied by deliberate anti-reversing techniques used to protect the binary file.

Executable code is vulnerable to malware. By replacing machine instructions with malicious ones, executable code can be transformed into malware. Malware can divert execution of code to perform one or more malicious tasks. Detection of malware contained in adversarial malware binaries is technically challenging, even with the use of artificial-intelligence techniques such as deep learning (Kolosnjaji et al., 2018).

We introduce here an approach based on texture vectors to allow executables to be compared against each other without requiring reverse engineering of the binaries. Our approach can be used as a first step to determine whether reverse engineering is needed.

Background

Contents of an Executable File

A binary contains more than just executable code. It includes fixed data, reserved space,

and links to executable code that is external to the file (Josse et al., 2014). Similarities in fixed data and fixed links are easiest to find because they can be matched directly. Reserved space usually consists of bytes with zero values and is found in many places in a typical executable file. It can complicate similarity measurements since there can be many false matches with zero bytes.

The portion of an executable file that contains the actual executable code consists of machine instructions and their associated operands. When executable code is modified, many machine instructions remain the same but usually their locations shift. Then the memory addresses encoded in their operands may change to compensate for this shift unless the code uses addressing relative to a register. However, register arguments encoded in operands may also shift.

Identifying File Similarity

Numerous approaches exist for identifying similarities between files. They can be used on text files, binary files, images, video, and audio. A few apply to files containing executable code. Some of these executable-analysis tools visualize software evolution in source code using version-control information or source-code file analysis (Arbuckle, 2008). A three-dimensional graph can show where code accesses the operating system or other information about code flow, and graph how these numbers change over the evolution of a software product. The Code Time Machine tool (Aghajani et al., 2017) does this to show the evolution of code metrics for a given file. It shows values along a time-line for the number of lines of code, number of methods, and cyclomatic complexity (i.e., the number of paths the code can take given the possible conditions written into the code). A three-dimensional graph of files and file relations between versions relates files. Circles represent releases, squares represent files, and edges represent associations (Koike & Chu, 1997). Other tools that graph code evolution are CVSScan (Voinea et al., 2005) and EPOSee (Burch et al., 2005).

There are many types of files. Three important ones are:

- **Text:** Text typically consists of words arranged in sentences. It may also be fragmented because of formatting, as in a formatted PDF file, or may be in short phrases in data tables or in the data section of executable code. We can measure text similarity by comparing words.
- **Arbitrary bytes:** What may appear as arbitrary bytes may be numeric data, compressed data, or executable codes. Numeric data often has low entropy because many of the bytes tend to be zero. Compressed data has high entropy because unused byte patterns in the data are removed.
- **Audio and video:** Audio and video data consists of bytes arranged in sequences. Bytes can be compared by aligning the sequences.

There are many algorithms for identifying similarities in data. Some work better than others given the type of data being compared. Methods used in comparing files are:

- **Comparing byte sequences:** Comparing content of text files to identify similarity is a common operation. One approach tries to find the longest common subsequences (Bergroth & Hakonen, 2000), where text that does not match is identified as new or deleted content. Algorithms for efficient string matching include Knuth-Morris-Pratt (Wikipedia, 2020), which allows searching without backtracking when a near match is found, and Boyer-Moore (Cole, 2018), which skips alignments when searching for specific text. Although intended for text, both algorithms may be used with executable code. Another popular algorithm for text files is implemented in the “diff” utility developed

for the Linux operating system (Shotts, 2019).

- **Comparing executable bytes:** Comparing bytes in files is similar to comparing text in files. However, bytes of files containing executable code are unlikely to match on operands and thus only the operators should be compared. For Intel architectures, operands are usually spaced at 4-byte or 8-byte intervals. Because of this, Cabezas and Mooij (n.d.) says that "binary file analysis by both binary diffing and cryptographic hash signatures comparison is a very limited approach to identify source code being re-used" and suggests metadata analysis. Regardless, for Intel architectures, it is useful to compare at every fourth or eighth byte because this will often align runs of comparisons with operators.
- **Comparing histograms:** We can identify common sequences of N bytes (N-grams) between files and contiguous sequences of bytes of a given length and compute a histogram of them. In Jang and Brumley (2009), 5-grams are used, and a Bloom filter is used as an efficient data structure for storing N-gram patterns that are found. Overall file similarity can be measured with the Jaccard index, the count of N-grams in common divided by number of distinct N-grams in both files. To take frequencies of the N-grams into account in measuring similarity, the cosine similarity or the Kullback-Leibler divergence can be used (Rowe, 2018).
- **Transforming values before comparison:** It may work better to measure similarity on transforms of the data values. This is commonly done for audio and video data; perceptual hashing (Hadmi et al., 2012) provides a similar hash output if features are similar. For images, we can transform the image to frequency space, or apply convolutions to it to enhance features. For signals, we can apply the Fourier transform to obtain frequencies.
- **Comparing metadata:** Initial comparisons of files can use their descriptive data to decide if they are sufficiently related to be worth further analysis. For example, if we know two executable files are built to run on a Microsoft Windows system using the same Intel instruction-set architecture, they are worth comparing. We can also compare metadata about sections and data structures within the files (Cabezas & Mooij, n.d.). Metadata includes:
    - File types and subtypes.
    - Data compression parameters. Cloning is indicated if the compressed size is significantly smaller than the combined size of its parts (Cabezas & Mooij, n.d.).
    - Mentions of precompiled libraries.
    - Hashcodes on the files.
- **Comparing decompiled data:** Executable files can be decompiled into text, and we can compare this text. Disassemblers and decompilers can do this, though they are not perfect. Disassemblers turn the bytes of executable code into corresponding machine code mnemonics and symbolic names, addresses, and offsets. Decompilers go further by turning bytes into source code.

Identifying similarity specifically between versions of source code can be accomplished in several ways:

- **Object-oriented analysis:** Software objects in source-code versions can be visually compared using a difference graph (Seemann & von Gudenberg, 1998). A graph of each version can be created where nodes are classes and node attributes are class methods and variables. Edges connect nodes where attributes of one node reference attributes of another. Then a class relation diagram is constructed that highlights differences in class

relations in two software versions.

- **Software-diagram analysis:** Software diagrams created during design may be compared if available (Rho & Wu, 1988).
- **Version control analysis:** Many products used by the software industry manage source code versioning with a repository (Swierstra & Lh, 2014). Then there is often documentation of the differences between versions.

Calculating File Similarity

In this section we present our texture-vector approach. We perform three layers of calculations to make inferences about similarity and how and where files are similar. Our steps are:

1. Calculate texture-vector datasets from the two files to be compared.
2. Compare texture-vector datasets to identify similarity offsets and produce a similarity offset histogram.
3. Calculate statistics from the heights of the similarity offset histogram to produce a single similarity measure for the comparison of the two files.

This process is illustrated in Figure 1.

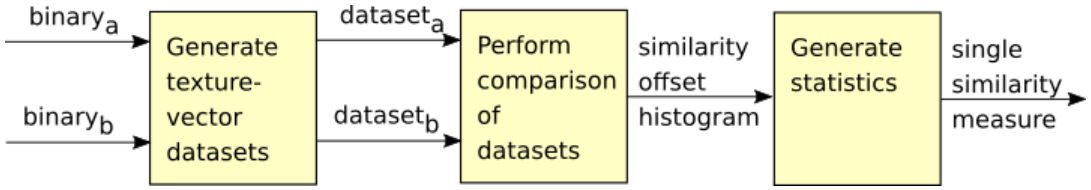


Figure 1: Inference Process

Calculating Texture-Vector Data

Texture vectors are calculated from the byte values of contiguous sections of binary data. Although many transform algorithms are possible, we are specifically interested in transforms that can both represent some unique characteristic of the data and possess a value that can be meaningfully compared to other values to measure similarity.

Sections measured as similar by many transforms have stronger similarities than others. We tested the following transforms for calculating texture vectors on the integer values of the bytes:

- **Standard Deviation:** The standard deviation of the byte values in a section of binary data. Two sections with a similar amount of deviation may be similar.
- **Mean:** The average byte value in the section. When executable code changes, operators may remain the same and help maintain the same mean.
- **Mode:** The most frequent byte value in the section. Often this value was zero in our data. This value is nonmetric and can only be used in computing similarity distances in the sense that it is identical or not.
- **Mode Count:** The count of occurrence of the most frequent byte value in the section.
- **Entropy:** The Shannon entropy of the byte values in the section. Two sections may be similar if the amount of randomness in each section is similar.

Calculating Texture-Vector Distance

Two texture vectors are defined as similar when the first texture-vector is within a

threshold of closeness to the second texture vector by the weighted square of the L2 (Euclidean) distance metric (Defant , 2011). The similarity can be thought of as 1/d2 where d is distance, calculated as: d = w1(dv1)2 + w2(dv2)2 + w3(dv3)2 + w4(dv4)2 + w5(dv5)2 where dv is the difference at a given vector element and w is the weight for a given vector element. For example if texture-vector 1 has values [100, 30, 220, 50, 80], texture-vector 2 has values [101, 32,225, 51, 80], and weights [w1, w2, w3, w4, w5] are [0.25, 0.25, 0.0, 0.25, 0.25], then the L2 distance d2 is 0.25 ∗ 12 + 0.25 ∗ 22 + 0.0 ∗ 52 + 0.25 ∗ 12 + 0.25 ∗ 02 = 0.25 + 1.0 + 0 + 0.25 + 0 = 1.5. A threshold of similarity was used for our graphics; for instance, if the acceptance threshold is 1.0, these vectors are not similar because 1.5 $ 1.0. We set weight values by experiment as explained in the Tuning Rejection Thresholds section. A good threshold identifies numerous correct similarities between the sections of data from which the texture vectors were calculated while excluding non-similarities.

Calculating Similarity Off Between Sections

We calculate similarity offsets by comparing all the texture vectors in one file against all the texture vectors in another file and counting the offsets between the files where the texture vector distance is within the threshold of closeness. When there are many offsets with the same value, this gives high confidence in those byte matches.

We implemented a display to show consistently strong offsets between two files. The display draws lines connecting similar texture vectors. The pattern and quantity of similarity lines indicates the nature and degree of file similarity. Figure 2 shows an example of two very similar versions of executable code, where the texture vector pattern of each file is shown across the top and bottom, and the lines between them indicate points of similarity. The files are both roughly 220 KB in length.

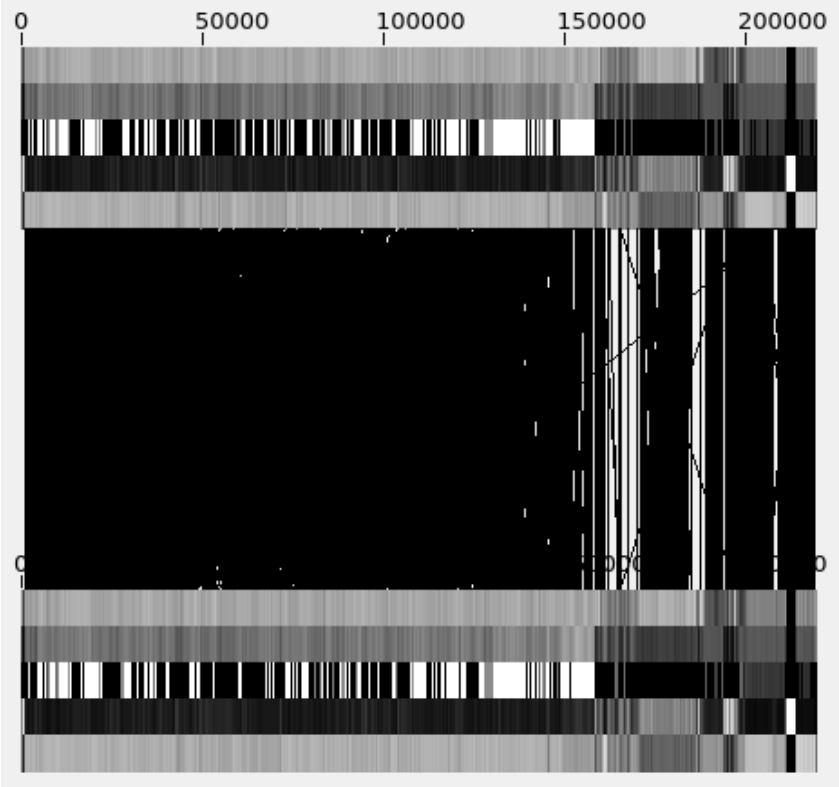


Figure 2. Texture Patterns of Two Very Similar Executable Files and Lines Connecting

Calculating Similar-Section Off Histograms

We calculate a similarity offset histogram from the set of offsets identified when searching for sufficiently similar texture vectors. There can be many thousands of offset values where similar-section matches can occur. To quantify this distribution of offsets, we create a similarity offset histogram and distribute calculated offset values across approximately 400 buckets, which sufficiently categorizes offsets in a viewable form. Consistent offset values are found as peaks on the histogram of offset values and represent likely meaningful similarities.

We calculate the measure of similarity between two files from the heights in the similarity offset histogram to provide a numeric measure of similarity between files. A large spread in heights suggests similarity at specific offsets, indicating similarity, while minimal spread in heights suggests a random distribution of similarity offsets, likely a result of false positives.

Calculating Similarity Measures Between Files

We calculate the measure of similarity between two files from the magnitude of the standard deviation of the heights of the compensated histogram as described in the Calculating Similar-Section Off Histograms section. An example of calculated similarity measure, along with the texture vectors, similarity offsets, and similar-section offset histograms, is shown in Figure 3. The top part describes the files being compared, the weights used in calculating the texture-vector distance, and statistics about the view, including the calculated similarity measure of 334.3535. The middle part shows the two texture-vector patterns, which visually appear identical, along with the center region saturated black with similarity lines. The bottom part shows the similarity histograms, where the similar-section offset histograms have spikes and low points. We will conclude that these two files are nearly identical in the Results section.

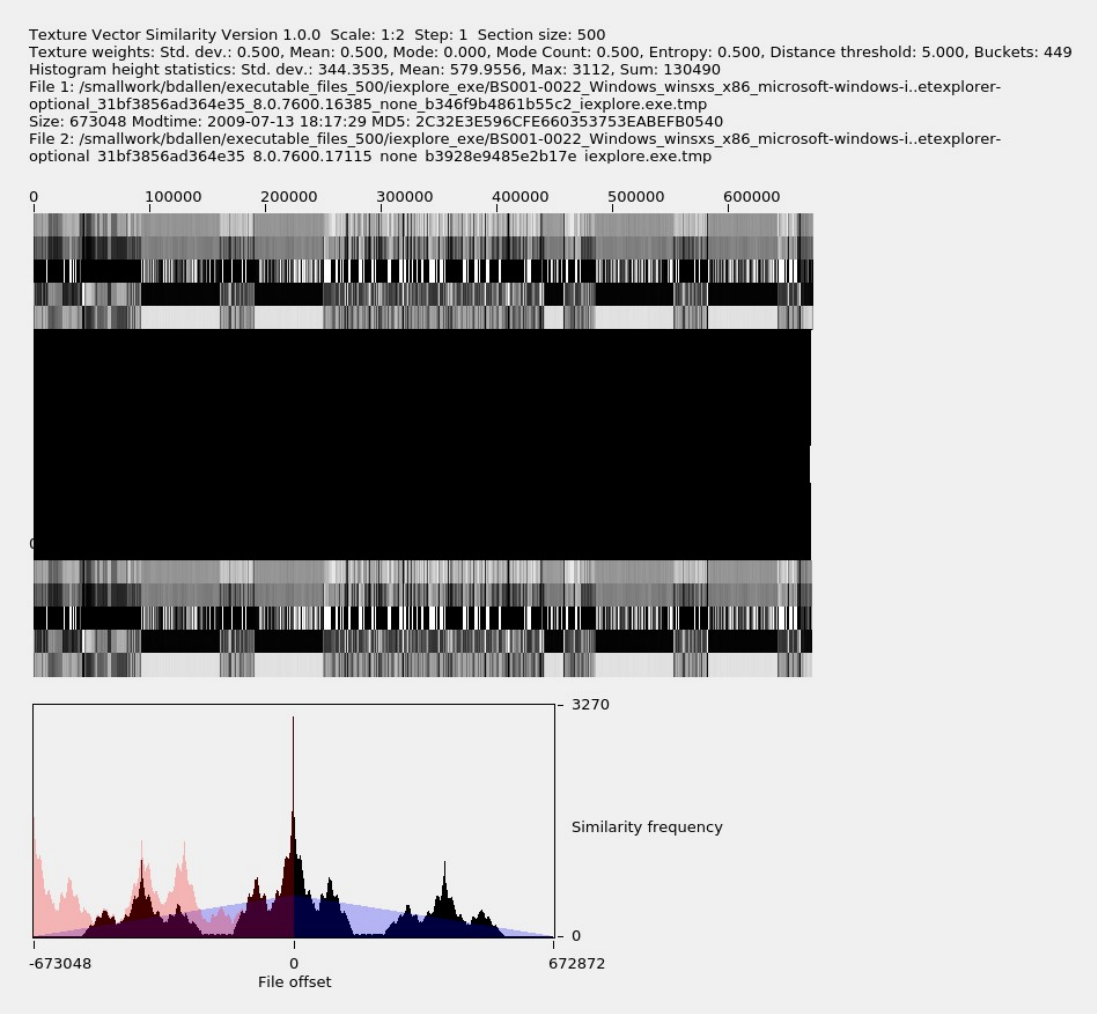


Figure 3. Example of High Value of High Similarity in File Family iexplore exe

Tracking Versions of Executable Code

We can also graph a network of relationships between different versions of the same executable. By using the file modification time for the horizontal axis and the calculated similarity measure described in the Calculating Similarity Measures Between Files section as the vertical axis, we can show the relationships between versions. Files that have a larger similarity measure to the selected file are plotted higher on the vertical axis. Files whose similarity measure is below a user-selectable measure are not plotted. By adjusting the similarity threshold using the SD slider in the Texture-Vector Browser GUI tool, we can remove files with minimal similarity to reveal clusters of files that match with greater similarity. Using this graph, we can make inferences; for example, releases with a similar modification time may be a result of bug fixes or security updates; releases with a smaller similarity measure may have more functional differences or may have added malware. An example of this graph is shown in Figure 9.

Preparing the Dataset of Executable Files

The dataset we studied consisted of executable files, texture-vector files, and similarity-graph files. The initial set of files was a sample of executable .exe and .dll files extracted from the Real Data Corpus (Garfinkel et al., 2009). The Real Data Corpus consists of “images”

(copies) of used disk drives and other devices obtained from non-U.S. countries. The files were extracted using the icat extraction tool from The Sleuth Kit forensics tool, (https://forensicswiki.org/wiki/The_Sleuth_Kit). Prof. Rowe picked 23 representative families of executables defined by a file name for each. Since many of the files were faulty, he used a software wrapper that loaded files for each distinct file contents (as indicated by its hash code) until the wrapper found a non-faulty copy. Names were changed from the original ones to distinguish files with the same names and different contents. The initial set consisted of 1,386 files. Of these, 162 were excluded because their size was greater than 1 MB and 55 were excluded because their size was less than 1 KB. Of the remaining 1,169 files, 35 were excluded because they were identical based on their MD5 cryptographic hash, leaving 1,134 files in our dataset. Figure 4 shows the distribution of file sizes. Note that since all files are from various countries and no files are from the United States, our collection may exclude important versions of software.

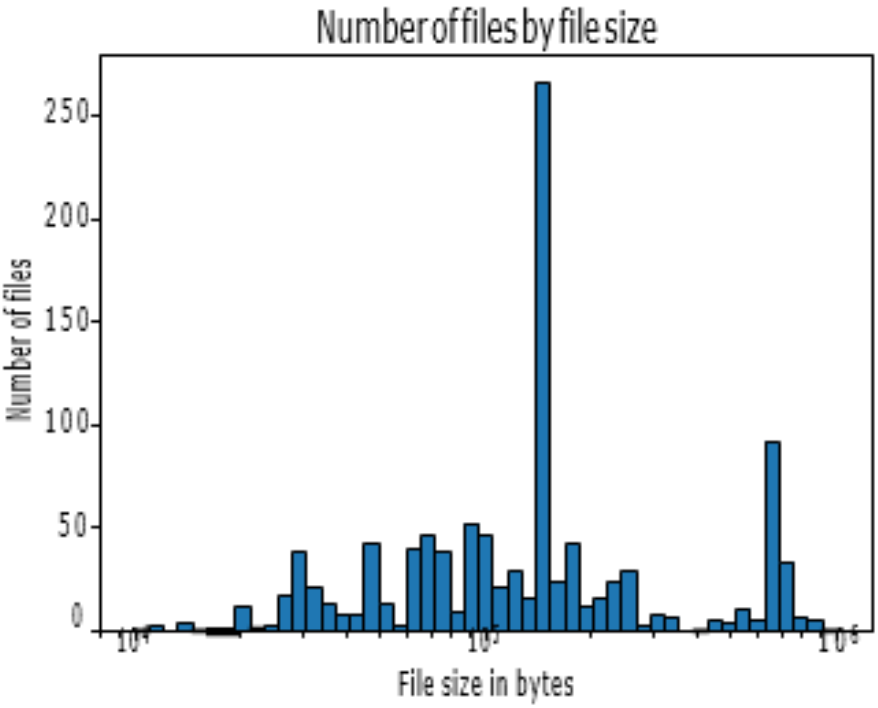


Figure 4. Histogram of File Sizes for our Dataset

The distribution of files by file modification time is shown in Figure 5.

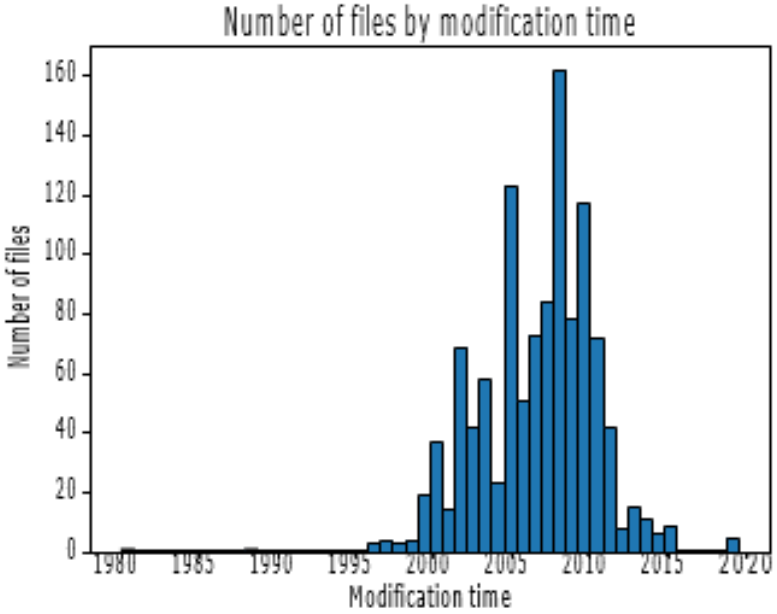


Figure 5. Histogram of File Modification Times for our Dataset

Statistics on the 23 file families that we studied are shown in Table 1. This includes source-code family tabulate_drive_data_py, which allows us to compare some versioned source-code files too.

Preparing the Texture-Vector Files

We created the texture-vector .tv files with the sbatch_calc_tv.bash program, which is part of the Texture-Vector Generator toolset. Due to the computational burden, we calculated texture vectors on the Naval Postgraduate School (NPS) Hamming supercomputer using sbatch parallel processing. We then copied these .tv files to the Texture-Vector Similarity repository, renaming them to their MD5 cryptographic hash value, for access by the Texture-Vector Similarity GUI tool.

Tuning Rejection Thresholds

Similarity is indicated when the square of the L2 distance measure is less than an acceptance threshold, as described in the Calculating Texture-Vector Distance section 3.1.1. We performed our tuning with two arbitrarily selected larger files in the ccalert_dll file family. We began with a default weight of 0.5 for the standard deviation, mean, mode count, and entropy transforms and, after some experimentation, we selected a distance rejection threshold of 5.0 because it resulted in reasonable similarity offsets without an oversaturation of matches. We selected a default weight of 0.0 for the mode because mode values do not quantifiably compare with each other, though an alternative could be to set distances between modes to 0 for identical values and 1 for nonidentical values.

Table 1. Files by File Family

File Family	File count	Min file size	Max file size	Mean file size	Standard deviation of file size
a0003775.dll	14	1591	853504	258271.6	318135.5
bthserv.dll	37	1067	92160	31455.4	19509.8
ccalert.dll	23	189560	267880	225524.2	21199.8
cdfview.dll	244	1178	409600	144513.2	39662.1
dunzip32.dll	34	11091	149040	114370.9	26991.3
hotfix.exe	33	53248	112912	94098.4	13263.9
iexplore.exe	216	3506	903168	461304.5	277712.7
mobsync.exe	80	8192	970752	156818.5	141438.6
msdc.dll	6	159232	194048	174933.3	15696.5
nvrshu.dll	32	151552	262144	240128.0	33724.4
pacman.exe	2	165594	241693	203643.5	53810.1
policytool.exe	104	1224	787508	54764.8	84605.1
powerpnt.exe	19	2310	676112	366290.8	236454.6
rtinstaller32.exe	4	135168	158312	146740.0	9843.3
safrslv.dll	29	1582	65536	41681.3	12648.2
tabulate_drive_data.py	23	18647	47544	34090.3	7213.7
typesheadfind.dll	2	35920	39856	37888.0	2783.2
udlaunch.exe	4	118784	118784	118784.0	0.0
vsplugin.dll	8	65606	118801	88180.2	15049.3
webcInt.dll	80	1261	611328	96930.6	92513.1
winprint.dll	7	12048	44544	29627.4	13120.4
wmplayer.exe	120	2864	520192	142871.3	101072.6
xxxwiadr.dll	13	8192	311296	123327.4	75040.4

We examined our tuning of weight values by setting all weight values to 0.0 and then, one weight at a time, examined the saturation of matched offsets as we adjusted the weight for each texture contribution from 0.0 to 1.0. For each weight adjustment, we observed that the quantity of similarity offsets identified would vary as we changed the weight and also that there was a visually understandable quantity of similarity at weight 0.5. Given this, we accepted our weight and rejection threshold values as our default values. These defaults are shown in Table 2.

Table 2. Default Texture-Vector Threshold Settings

Setting	Type	Value
Standard Deviation	Weight	0.5
Mean	Weight	0.5
Mode	Weight	0.0
Mode Count	Weight	0.5
Entropy	Weight	0.5
Rejection threshold	Threshold	5.0

Preparing the Similarity-Graph Files

We created the similarity-graph files by running the sbatch_ddiff_tv.bash program which is part of the Texture-Vector Similarity toolset. We calculated the similarity metrics on the NPS Hamming supercomputer using sbatch parallel processing with a job queue size of 700, resulting in a graph of 1,134 nodes and 463,486 edges from which we can create a similarity matrix across all file families. We compared files across file families in order to measure

similarity between known dissimilar files. There are 642,411 possible edges, but we dropped 178,925 of them because they had less than two similarity matches. This processing took about 15 hours. Runtime of each file pair varied because file sizes varied.

Table 2. Default Texture-Vector Threshold Settings

Setting	Type	Value
Standard Deviation	Weight	0.5
Mean	Weight	0.5
Mode	Weight	0.0
Mode Count	Weight	0.5
Entropy	Weight	0.5
Rejection threshold	Threshold	5.0

Node data consists of the node index, filename, file family, file size, file-modification time, and file MD5 hashcode. Edge data consists of the edge's source and target file node indexes along with the standard deviation, mean, maximum, and sum similarity metrics described in the Calculating Similarity Measures Between Files section.

Results

To evaluate the ability of our tools to identify similarities between executable files, we examined the 642,411 texture-vector similarity measures calculated for each pair of files for the 1,134 files. Of the 642,411 possible comparisons, 463,486 of them produced nonzero similarity values. Similarity measure values varied from zero to about 300. The distribution of these 463,486 similarity values across all files in our dataset is shown in Figure 6. Due to the uneven distribution of these values, a similarity threshold cannot be calculated using a normal gaussian distribution. Most similarity measure values were less than 10, which is where the curve becomes level. This suggests that actual similarity between two files may be indicated when their similarity measure is greater than 10.

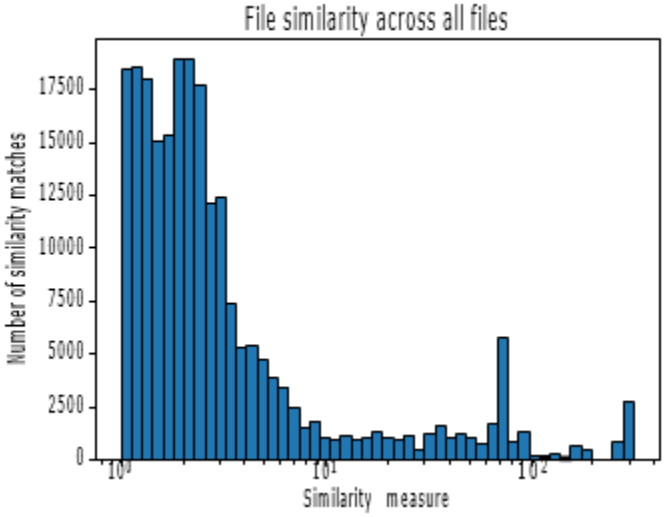


Figure 6. Histogram of Similarity Matches Across All Files in Our Dataset

Evaluating Similarities

To establish a baseline of what the similarity measure values are for similar files, we calculated the mean similarity measures within and across file families. We tested whether the similarity measure between files of the same file family was higher than the similarity measure between files in different file families. The confusion matrix for file similarity across all file families in our dataset is in Table 3. Rows and columns represent file families using the numbers in the second column. The mean similarity measures between files within file families is typically greater than the mean similarity between files in other file families, showing that our approach for identifying file similarity is useful.

Table 3. Mean File Similarity Between File Families

Family	No.	1	2	3	4	5	6	7	8	9	10	11	12
a0003775.dll	1	4.5	1.2	5.4	2.1	3.8	3.0	3.6	2.8	2.2	3.3	5.1	2.3
bthserv.dll	2	1.2	3.7	1.2	0.7	0.7	0.7	0.5	0.7	0.6	0.3	0.9	0.6
ccalert.dll	3	5.4	1.2	11.4	2.5	3.9	3.2	2.2	2.9	3.6	4.3	4.8	2.2
cdfview.dll	4	2.1	0.7	2.5	10.0	1.3	1.1	1.6	2.2	1.8	0.9	1.7	0.8
dunzip32.dll	5	3.8	0.7	3.9	1.3	5.1	2.3	4.0	2.1	2.0	3.4	3.6	1.6
hotfix.exe	6	3.0	0.7	3.2	1.1	2.3	8.5	1.3	2.0	1.4	3.8	3.1	1.6
iexplore.exe	7	3.6	0.5	2.2	1.6	4.0	1.3	130.2	9.3	1.6	2.4	1.5	7.5
mobsync.exe	8	2.8	0.7	2.9	2.2	2.1	2.0	9.3	6.1	1.5	2.3	2.7	1.6
msrdc.dll	9	2.2	0.6	3.6	1.8	2.0	1.4	1.6	1.5	4.5	1.4	2.0	0.9
nvrshu.dll	10	3.3	0.3	4.3	0.9	3.4	3.8	2.4	2.3	1.4	32.9	6.2	2.1
pacman.exe	11	5.1	0.9	4.8	1.7	3.6	3.1	1.5	2.7	2.0	6.2	1.5	2.2
policytool.exe	12	2.3	0.6	2.2	0.8	1.6	1.6	7.5	1.6	0.9	2.1	2.2	2.6
powerpnt.exe	13	3.5	0.4	2.2	1.1	3.1	1.5	41.2	5.8	1.4	2.6	2.2	4.6
rtinstaller32.exe	14	3.4	0.9	4.1	2.0	3.6	2.0	1.6	2.3	2.2	2.3	2.8	1.2
safrslv.dll	15	1.9	0.9	2.2	1.1	1.2	1.6	1.1	1.1	0.7	2.0	2.0	1.0
tabulate_drive_data.py	16	0.1	0.1	0.1	0.1	0.1	0.1	0.2	0.1	-	0.1	-	0.3
typeaheadfind.dll	17	0.9	0.6	1.3	0.7	0.4	0.3	0.4	0.5	0.7	0.1	0.7	0.5
udlaunch.exe	18	2.9	0.4	3.3	1.1	2.5	-	1.3	1.7	1.8	3.3	3.0	-
vsplugin.dll	19	3.0	0.6	3.4	1.0	2.0	2.5	4.0	1.8	1.2	3.2	3.0	1.6
webclnt.dll	20	3.3	1.0	3.6	1.1	2.3	1.3	1.8	1.8	1.5	2.2	2.8	1.0
winprint.dll	21	0.8	0.5	0.9	0.4	0.6	0.5	0.4	0.5	0.5	0.4	0.6	0.6
wmplayer.exe	22	3.1	0.4	3.1	0.9	2.4	2.0	21.7	3.6	1.3	3.0	2.8	2.4
xrxwiadr.dll	23	11.5	0.8	12.1	2.5	9.2	4.1	3.2	4.6	3.3	12.9	13.0	3.8

Family	No.	13	14	15	16	17	18	19	20	21	22	23
a0003775.dll	1	3.5	3.4	1.9	0.1	0.9	2.9	3.0	3.3	0.8	3.1	11.5
bthserv.dll	2	0.4	0.9	0.9	0.1	0.6	0.4	0.6	1.0	0.5	0.4	0.8
ccalert.dll	3	2.2	4.1	2.2	0.1	1.3	3.3	3.4	3.6	0.9	3.1	12.1
cdfview.dll	4	1.1	2.0	1.1	0.1	0.7	1.1	1.0	1.1	0.4	0.9	2.5
dunzip32.dll	5	3.1	3.6	1.2	0.1	0.4	2.5	2.0	2.3	0.6	2.4	9.2
hotfix.exe	6	1.5	2.0	1.6	0.1	0.3	-	2.5	1.3	0.5	2.0	4.1
iexplore.exe	7	41.2	1.6	1.1	0.2	0.4	1.3	4.0	1.8	0.4	21.7	3.2
mobsync.exe	8	5.8	2.3	1.1	0.1	0.5	1.7	1.8	1.8	0.5	3.6	4.6
msrdc.dll	9	1.4	2.2	0.7	-	0.7	1.8	1.2	1.5	0.5	1.3	3.3
nvrshu.dll	10	2.6	2.3	2.0	0.1	0.1	3.3	3.2	2.2	0.4	3.0	12.9
pacman.exe	11	2.2	2.8	2.0	-	0.7	3.0	3.0	2.8	0.6	2.8	13.0
policytool.exe	12	4.6	1.2	1.0	0.3	0.5	-	1.6	1.0	0.6	2.4	3.8
powerpnt.exe	13	76.0	1.5	1.0	0.2	0.2	1.5	2.8	1.7	0.3	12.6	8.2
rtinstaller32.exe	14	1.5	13.4	1.1	0.1	0.4	3.1	1.9	2.0	0.6	1.7	6.3
safrslv.dll	15	1.0	1.1	3.3	0.1	0.8	-	1.4	1.2	0.6	1.1	2.6
tabulate_drive_data.py	16	0.2	0.1	0.1	2.8	0.1	-	0.2	0.2	-	0.1	0.3
typeaheadfind.dll	17	0.2	0.4	0.8	0.1	2.3	0.2	0.5	0.8	0.4	0.2	0.6
udlaunch.exe	18	1.5	3.1	-	-	0.2	-	2.1	0.9	0.5	2.2	3.6
vsplugin.dll	19	2.8	1.9	1.4	0.2	0.5	2.1	3.2	1.7	0.5	2.7	3.5
webclnt.dll	20	1.7	2.0	1.2	0.2	0.8	0.9	1.7	3.8	0.7	1.5	5.0
winprint.dll	21	0.3	0.6	0.6	-	0.4	0.5	0.5	0.7	1.1	0.4	0.6
wmplayer.exe	22	12.6	1.7	1.1	0.1	0.2	2.2	2.7	1.5	0.4	9.1	5.7
xrxwiadr.dll	23	8.2	6.3	2.6	0.3	0.6	3.6	3.5	5.0	0.6	5.7	15.9

Although the greatest average similarity for a given file family is usually within that file family, there are exceptions as between file families a0003775_dll and xrxwiadr_dll. This inconsistency could be due to the differences in file size or to other attributes within the files in these two file groups. An example similarity analysis plot illustrating the problem is Figure 7. Ranges of homogeneous texture vectors contain similar low mode counts and moderately high entropy values, suggesting that our similarity measure is primarily attributed to regions of compressed data rather than similarity in code. The few similarity matches in other regions suggest that there is actually little similarity between these two files.

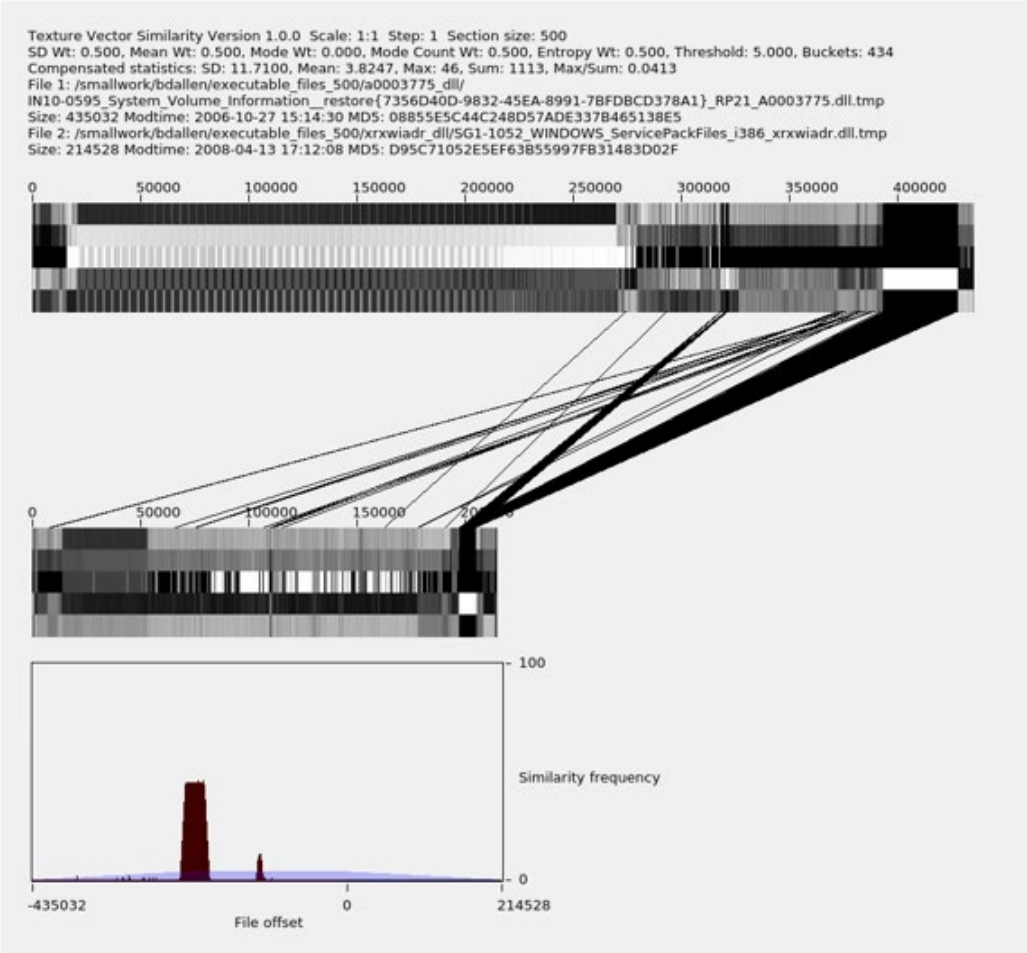


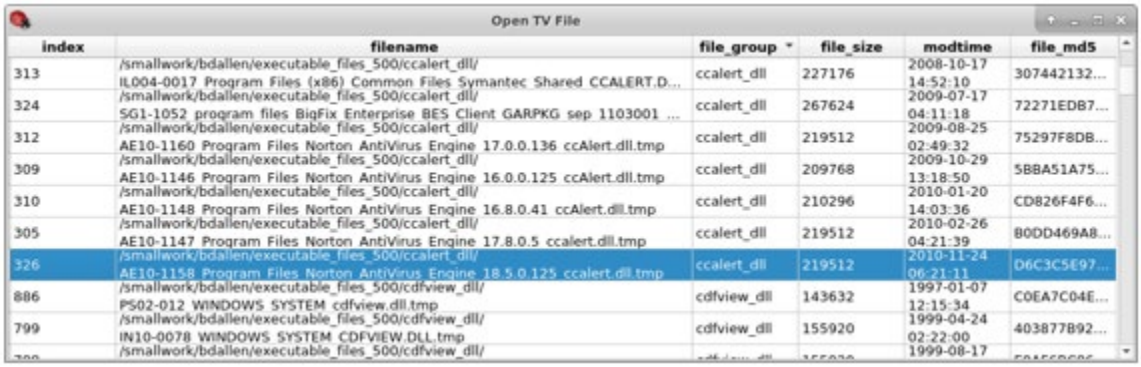
Figure 7. False-Positive Similarity Between Two Files Caused by Homogeneous Compressed Data

Examining Similarity Using the Texture-Vector Browser GUI Tool

Our Texture-Vector Browser GUI tool can examine trends in file similarity based on file creation times and file-similarity measures. Figure 9 shows an example. The horizontal axis is the file modification time. This can be the time the file was created if it was never modified, or the time it was modified by update or by contamination with a virus. The vertical axis is the measure of similarity between the file the user selects and the other files in the view, which if the Stay in group mode is selected, will be files within its family. Files higher up on the vertical axis are more similar to the selected file than files lower down on the vertical axis, where the similarity measure, as described in the Calculating Similarity Measures Between Files section, is the value on the vertical axis. By clicking on a node, the focus of the view changes to show the

similarities between the file associated with the clicked node and other files. By clicking on an edge, the view shows the similarity graph involving the two files associated with the edge.

Using the node listing capability in the Texture-Vector Browser GUI and by sorting the list by file group and modification time, we find and select the file in the ccalert_dll file group with the latest timestamp, as shown in Figure 8.



index	filename	file_group	file_size	modtime	file_md5
313	/smallwork/bdallen/executable_files_500/ccalert_dll/ IL004-0017 Program Files (x86) Common Files Symantec Shared CCALERT.D...	ccalert_dll	227176	2008-10-17 14:52:10	307442132...
324	/smallwork/bdallen/executable_files_500/ccalert_dll/ SG1-1052 program files BigFix Enterprise BES Client GARPKG sep 1103001 ...	ccalert_dll	267624	2009-07-17 04:11:18	72271EDB7...
312	/smallwork/bdallen/executable_files_500/ccalert_dll/ AE10-1160 Program Files Norton AntiVirus Engine 17.0.0.136 ccAlert.dll.tmp	ccalert_dll	219512	2009-08-25 02:49:32	75297F8DB...
309	/smallwork/bdallen/executable_files_500/ccalert_dll/ AE10-1146 Program Files Norton AntiVirus Engine 16.0.0.125 ccAlert.dll.tmp	ccalert_dll	209768	2009-10-29 13:18:50	5BBA51A75...
310	/smallwork/bdallen/executable_files_500/ccalert_dll/ AE10-1148 Program Files Norton AntiVirus Engine 16.8.0.41 ccAlert.dll.tmp	ccalert_dll	210296	2010-01-20 14:03:36	CD826F4F6...
305	/smallwork/bdallen/executable_files_500/ccalert_dll/ AE10-1147 Program Files Norton AntiVirus Engine 17.8.0.5 ccalert.dll.tmp	ccalert_dll	219512	2010-02-26 04:21:39	B0DD469A8...
326	/smallwork/bdallen/executable_files_500/ccalert_dll/ AE10-1158 Program Files Norton AntiVirus Engine 18.5.0.125 ccalert.dll.tmp	ccalert_dll	219512	2010-11-24 06:21:11	D6C3C5E97...
886	/smallwork/bdallen/executable_files_500/cdfview_dll/ PS02-012 WINDOWS SYSTEM cdfview.dll.tmp	cdfview_dll	143632	1997-01-07 12:15:34	C0EA7C04E...
799	/smallwork/bdallen/executable_files_500/cdfview_dll/ IN10-0078 WINDOWS SYSTEM CDFVIEW.DLL.tmp	cdfview_dll	155920	1999-04-24 02:22:00	403877B92...
	/smallwork/bdallen/executable_files_500/cdfview_dll/			1999-08-17	

Figure 8. Sorted Node Listing with Node 326 Selected

In our dataset, this file is named AE10-1158_Program_Files_Norton_AntiVirus_Engine_18.5.0.125_ccalert.dll.tmp, indicating that it is on drive AE10-1158 from United Arab Emirates. It is indexed in our similarity graph dataset as node 326 (in green). The file naming convention is explained in the Peparing the Dataset of Executable Files section. This graph shows node 326 and its similar neighbors and similar edges, where the similarity measure, described in the Calculating Similarity Measures Between Files section, is 1.0 or more. The horizontal axis is the file modification time and the vertical axis is the relative similarity between file (node) 326 and the other files. In this graph, we see two clusters of similarity.

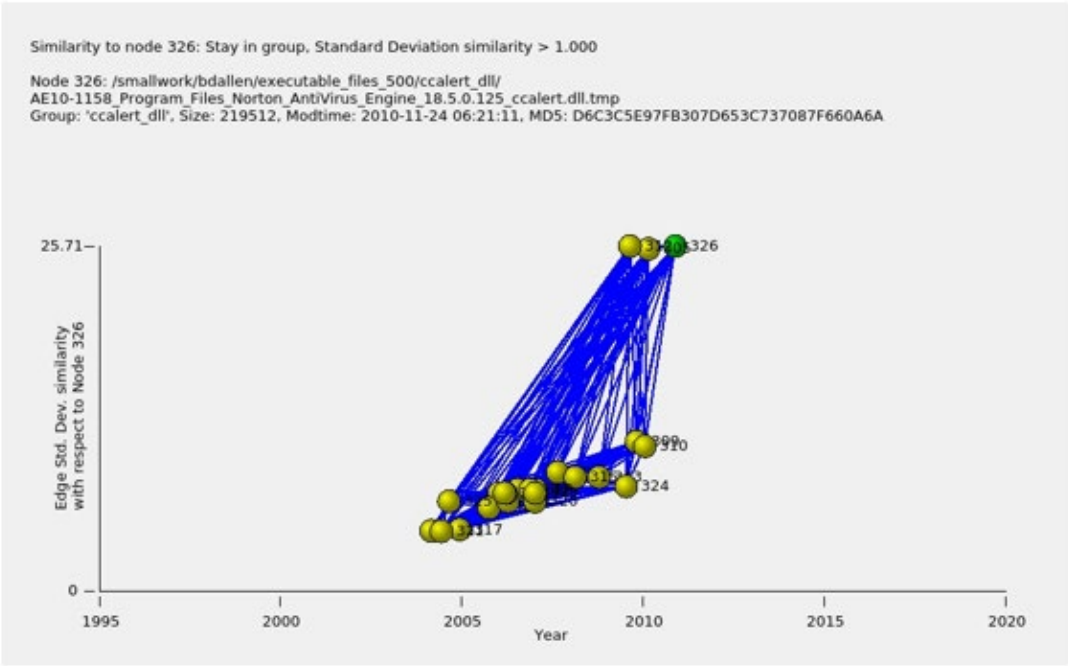


Figure 9. Files (Nodes) and Similarity Measures (Edges) Associated With File Node 326 Showing Modification Times and Similarity to Node 326

Figure 10 shows the analysis of the edge that connects nodes 326 and 312, corresponding to Program Files/Norton AntiVirus Engine 18.5.0.125_ccalert.dll on drive AE10-1158 and Program Files Norton AntiVirus Engine 17.0.136_ccAlert.dll on drive AE10-1160. This display was obtained using the GUI by clicking on the edge shown in Figure 9 that connects these two files. The texture vector patterns appear very similar and the similarity histogram spikes with a similarity count of nearly 370 near file offset 0, a large number, indicating that these two files are similar. We can click on any of the yellow dots in the GUI to select the file corresponding to it to compare other files against it.

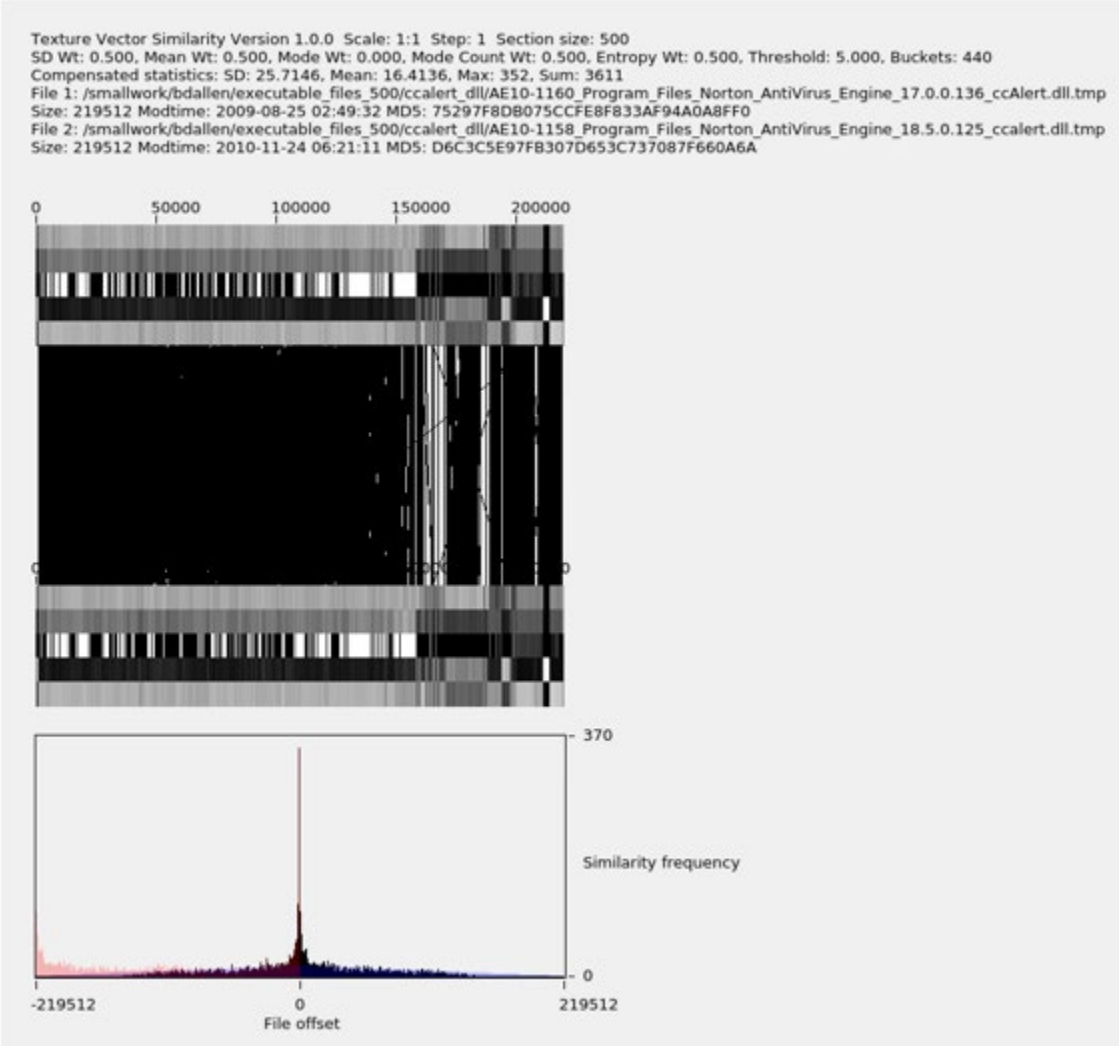


Figure 10. A Detailed Comparison of Files 312 and 326 Showing a High Degree of Similarity

Composition Analysis

By looking at the five bands in the texture-vector diagram, we can make inferences about the regions of executable code files being compared, in particular the locations of header, code, and data sections. For Figure 10, for the first two textures, covering the first 1,000 bytes, the standard deviation, mean, mode, and entropy values are lower than the values in other regions, while the mode count is higher. We infer that this represents a header, and the transition in the texture represents a transition to another type of content. The region from approximately byte 1,000 to byte 160,000, contains relatively medium values of the standard deviation, mean, and entropy, mode values that are either very high or very low, and

consistently low mode counts. We infer that this is the code section. The third region, from approximately byte 160,000 through to the end at byte 219,512, usually has a low mode value while values in the other four statistics vary but consistently with the two files. We infer that this is a region of data mostly unchanged between version. We also infer that the additional 10 KB added in the newer version was new code.

Progressive Time Similarity

Software files tend to be most similar to the previous version. Figure 11 shows an example for the nvrshu_dlll file family. Here, the file with the latest timestamp, WINDOWS system32 nvrshu.dll from the MY01-023 drive from Malaysia, is selected. We see sporadic measures of similarity between 10 and 30 for files before year 2005, but for files after 2005, we see a gradual increase in similarity over time from about 40 to 61.

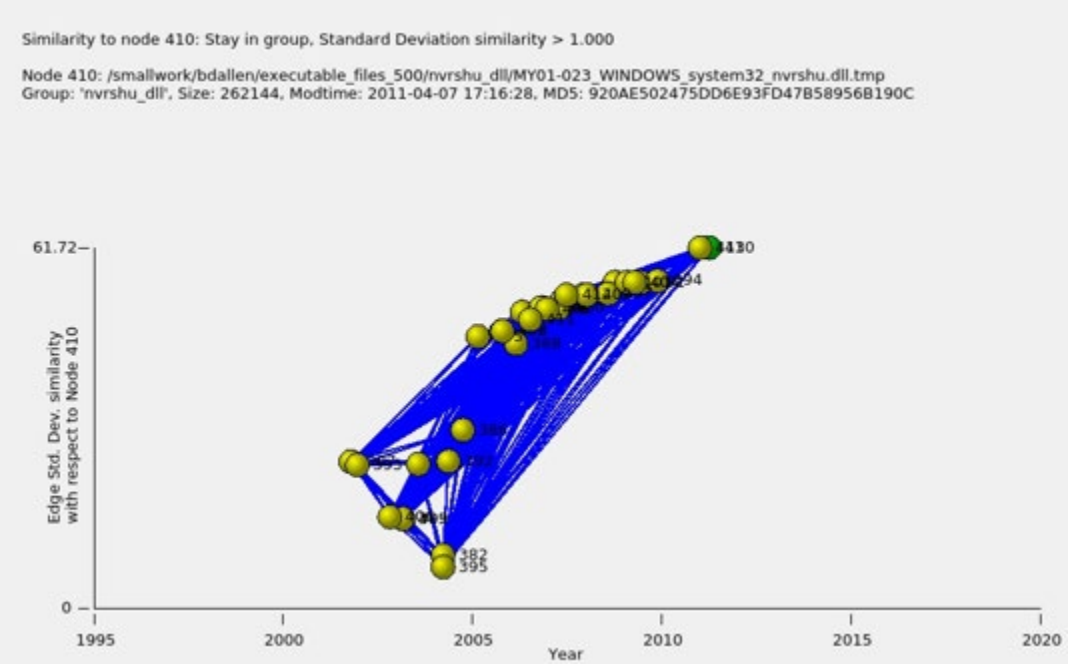


Figure 11: Similarity Increases as Versions Approach the Latest Version

Version Analysis

With these diagrams, we can study on the origin and evolution of versions of files. Although an original file should have the earliest file creation time, file creation times can be modified inadvertently or maliciously. Another clue is that the original file often has the least amount of code. Node 326 in Figure 9, file Program Files/Common Files/Sym antex/Shared ccAlert.dll.tmp from drive PA002-049 from Panama is likely the original file in its group because its file modification time is earliest and its similarity to latest files decreases over time.

A newer version of code that introduces new features is likely to contain more code than the version before it. A newer version that is only a bug fix will be similar in size to the version before it and will have similar texture-vector patterns as in Figure 10.

Files released at approximately the same time may be targeted for different operating system platforms or different feature sets. For example 13 files in the webclnt_dll file family were released over 2 days, 2006-01-03 and 2006-01-04. This is too clustered to be in response to new functionality or bug fixes. These files could be a response to a virus because some of their file sizes are the same and their texture-vector patterns appear identical. However, bear in mind

our sample is incomplete and important versions of software may be missing.

Examining Similarity using Gephi

Although the Texture-Vector Browser GUI tool was specifically designed for examining network graphs created from the dataset of similarity-graph files, graph analytics can also be done with popular open-source tools such as the Gephi graph-visualization tool.

Conclusions and Future Work

Conclusions

This thesis proposed applying a vector of transforms to executable code to create texture-vector data, and then using analytics to identify similarities between executable files. We tested a sample of executable code files with our methods. Our experiments showed files within file families had greater average similarity than files across file families. We found that the visual patterns in the texture vectors were effective in identifying similar regions in two files as well as sections that may be compressed.

Future Work

This work used texture vectors calculated from a section size of 500 bytes. A large section size might reveal similarity across a larger section of data, equivalent to applying a low-pass filter to texture-vector values. A section size that is a power of two or is aligned to the size of fixed-size data might naturally align better with the section boundaries from which texture-vectors are calculated.

Texture vectors may be useful for classifying file types or detecting types of data embedded within a file. Further work in this direction might consist of defining data patterns that map to particular data types.

The open-source tool Gephi offers many capabilities such as filtering and neighbor analytics that can be used to augment the similarity analytics provided by our tool.

Future work might use it to obtain additional insight about file similarity.

References

Aghajani, E., Mocci, A., Bavota, G., & Lanza, M. (2017). The code time machine. In 2017 IEEE 25th International Conference on Program Comprehension (ICPC).

Arbuckle, T. (2008). Visually summarising software change. In 12th International Conference Information Visualization.

Bergroth, T. R. L., & Hakonen, H. (2000). A survey of longest common subsequence algorithms. In Proceedings of the International Symposium on String Processing Information Retrieval (SPIRE ’00), 39–48.

Burch, M., Diehl, S., & Weigerber, P. (2005). Visual data mining in software archives. In SoftVis ’05 Proceedings of the 2005 ACM Symposium on Software Visualization, 37–46.

Cabezas, D., & Mooij, B. (n.d.). Detecting source code re-use through a binary analysis hybrid approach. Forensic Magazine. https://www.forensicmag.com/article/2013/02/detecting-source-code-re-use-through-binary-analysis-hybrid-approach

Cole, R. (2018). Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm (1st ed.). Forgotten Books.

Defant, A. (2011). Classical summation in commutative and noncommutative Lp-spaces (1st ed.). Springer.

Garfinkel, S., Farrell, P., Roussev, V., & Dinolt, G. (2009, August). Bringing science to digital forensics with standardized forensic corpora. Digital Investigation, 6, S2–S11.

Hadmi, A., Puech, W., Said, B. A. E, & Ouahman, A. A. (2012). Perceptual image hashing. In University of Montpellier II, CNRS UMR 5506-LIRMM, France.

Jang, J., & Brumley, D. (2009). Bitshred: Fast, scalable code reuse detection in binary code. CMU-CyLab-10-006.

Josse, S., Bachaalany, E., Gazet, A., & Dang, B. (2014). Practical reverse engineering: x86, x64, ARM, Windows Kernel, reversing tools, and obfuscation (1st Ed.). Wiley.

Koike, H. & Chu, H.-C. (1997). Vrcs: Integrating version control and module management using interactive three-dimensional graphics. In Graduate School of Information Systems University of Electro-Communications Chofu, Tokyo 182, Japan.

Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C., & Roli, F. (2018, December). Adversarial malware binaries: Evading deep learning for malware detection in executables. In Proceedings of the 26th European Signal Processing Conference, Rome, Italy.

Rho, J., & Wu, C. (1988). An efficient version model of software diagrams. In Proceedings 1998 Asia Pacific Software Engineering Conference (Cat. No.98EX240).

Rowe, N. C. (2018). Associating drives based on their artifact and metadata distributions. In Proceedings of the 10th International EAI Conference, ICDF2C 2018, New Orleans, LA, USA.

Seemann, J., & von Gudenberg, J. W. (1998, March). Visualization of differences between versions of object-oriented software. In Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering.

Shotts, W. (2019). The Linux command line (2nd ed.). No Starch Press.

Swierstra, W., & Lh, A. (2014). The semantics of version control. In Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2014), 43–54, New York, NY, USA.

Voinea, L., Telea, A., & van Wijk, J. J. (2005). Cvsscan: Visualization of code evolution. In SoftVis ’05 Proceedings of the 2005 ACM Symposium on Software Visualization, 47–56.

Wikipedia. (2020, February 4). Knuth-Morris-Pratt algorithm. https://en.wikipedia.org/wiki/Knuth-Morris-Pratt algorithm

Acquisition Research Program
Graduate School of Defense Management
Naval Postgraduate School
555 Dyer Road, Ingersoll Hall
Monterey, CA 93943

www.acquisitionresearch.net