NPS-IT-20-014

ACQUISITION RESEARCH PROGRAM
SPONSORED REPORT SERIES

Analysis of Differences between Versions of Software Executables

December 2, 2019

Dr. Neil C. Rowe, Professor

Bruce D. Allen, Research Associate

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.

Abstract

Software is frequently involved today in acquisitions. It is important to identify fraudulent, malicious, or illegally copied software but that is more difficult than identifying those features in physical objects. This work applied methods of digital forensics to this task. We studied differences between versions of software by comparing their executable files. We used a large database (“corpus”) of around 2600 digital-forensic copies of secondary storage of computers and digital devices purchased around the world. We extracted families of executable files in the EXE and DLL formats having the same name, usually representing different updates of the same software. We measured file similarities between files in the same family by finding matches between 8-bit bytes in the two files, and then looking for sequences of unbroken consecutive matches. We developed several kinds of useful visualizations to show file similarities: Two ways to display the bytes that match between two files, and two ways to show the similarities between members of a file family over time. These methods should make it considerably easier to detect fraudulent, malicious, or illegally copied software because it will stand out in the visualizations.

THIS PAGE LEFT INTENTIONALLY BLANK

NPS-IT-20-014

ACQUISITION RESEARCH PROGRAM
SPONSORED REPORT SERIES

Analysis of Differences between Versions of Software Executables

December 2, 2019

Dr. Neil C. Rowe, Professor

Bruce D. Allen, Research Associate

Graduate School of Defense Management

Naval Postgraduate School

THIS PAGE LEFT INTENTIONALLY BLANK

Table of Contents

THIS PAGE LEFT INTENTIONALLY BLANK

List of Figures

List of Tables

THIS PAGE LEFT INTENTIONALLY BLANK

Introduction

Acquisition of software can be challenging compared to acquisition of other products. Software is difficult to analyze, and lack of product liability means many claims made by its vendors are very difficult to confirm and may be exaggerations or, in rare cases, outright fraud. This work tried to characterize what is actually being stored for some major software products and what happens to the software after updates using methods of digital forensics. This provides a new way to verify some of the claims made about software, and should provide additional clues to detect malicious, fraudulent, or illegally copied software.

Software is implemented in a set of executable files. Executable files are instructions for a computer or digital device to run the software. They are encoded in the machine language of the computer or device, and are almost always created by applying compiler software to source code. Executable files are much less human-readable than source code. However, digital forensics rarely encounters source code when it investigates computers and devices, as only executables are necessary to run programs; source code is also more valuable proprietary intellectual property than executables and is more subject to controlled access than executables. Thus to do digital forensics on programs, an investigator needs tools to analyze executables.

Machine language used by executables consists of instruction codes (operators) and arguments (operands). 32-bit machines have 32-bit (4-byte) instructions and 64-bit machines have 8-byte instructions. Operators specify the type of operation to be performed and operands specify registers (fast storage), memory addresses (slower storage), buffer numbers, or arithmetic constants. The primary duty of compiles is to construct machine instructions that implement the source code; in addition, compilers must assign registers and memory addresses to the data. This means that minor changes to source code can force large number of addresses to shift in memory. Compilers often do specify addresses relative to a register value, but registers may change when parts of a program are expanded or reduced in length. Thus some things in executable code will remain constant through different versions – operators, some register arguments, and some buffer arguments – and others will change. Comparing versions of an executable requires focusing on the things that remain constant that are scattered through the file.

Scope of the Study

Our immediate goal was to study the differences between the same executable (binary file) over time and between versions. This study had several purposes. One purpose was to see how much could be told from an executable alone about the changes over time. When an executable is recompiled for even a minor update, most registers and addresses can change. However, instruction codes ("opcodes") and constant data should not change very much with different versions, and we should be able to match them reliably. Blank areas are frequent in executables, usually appearing as sequences of zero bytes, and we can match those albeit less reliably. This means we should be able to see how much was added or subtracted from the executable with each version to get an idea of the magnitude of an update.

Our ultimate goal was to provide a basis for recognizing incompetent, fraudulent, or malicious updates to software. A version without substantial changes from a previous version could have been stolen and superficially modified. It could also be malware that changed just a few bytes to remain hard to detect if the size remains the same. However, malicious actors in cyberspace are increasingly brazen, and many versions of software identified as malware are at the opposite extreme of being completely different from legitimate versions. Our experiments measured how different they were.

THIS PAGE LEFT INTENTIONALLY BLANK

Literature Review

Software development is a complex process, and a variety of projects have attempted to assist to software engineers in understanding differences between versions of software. Usually only a few features of source code change between versions, and these can be found by comparisons of the source code (Palix et al, 2010). Changes can be abstracted by generalizations (Zeller and Snelting, 1997). Most useful approaches to version analysis with source code focus on the changes to components since most software engineering is component-based (Gergic, 2003). Executables are more difficult to analyze.

Especially useful to software engineering is visualization to the relationships between different versions of software, and many of these ideas apply to executables as well. (Merino et al, 2018) provides a good survey of methods for software visualization. These include visualization of clustering of versions by similarity (Beyer and Hassan, 2006), tree diagrams of the relationships between versions and components of versions (Arbuckle, 2008; Seeman and Gudenberg, 2009; Kaur and Singh, 2009; Novais et al, 2011; Elsen, 2013; Novais et al, 2107), diagrams comparing versions that use color and position to encode features (Voinea et al, 2005; Kuhn and Stocker, 2012; Hanjalic, 2013), and graphs showing version evolution over time (Aghajani et al, 2017). Some work has addressed visualization of graphs showing mappings between software components (Kim and Notkin, 2006; Kaur and Singh, 2011), abstractions about the software (Rho and Wu, 1998), and connections between parts within different software versions (Burch et al, 2005; Voinea et al, 2005), the latter of which is more directly related to our project. Some visualizations have involved human interaction to guide visualization (North et al, 2016). Some methods for visualizing comparisons between text documents are also relevant to software such as drawing lines between matching items (Shannon et al, 2010).

THIS PAGE LEFT INTENTIONALLY BLANK

Data Used for Testing

We extracted a sample of executable names from the 2600 images in the full-image portion of the Real Drive Corpus, a collection drive images obtained mostly by purchase of used equipment around the world (Garfinkel et al, 2009). There were 29,821,053 executables identified in our corpus by their file extensions, having 3,312,861 different hash values (different contents). Figure 1 shows the distribution of the number of files having the same filename in our corpus, and Figure 2 shows the distribution of the fraction of distinct hash values to total files for each filename. Most executable filenames occur only once, but a significant number of popular executables occurred many times as shown on the left side of the graph.

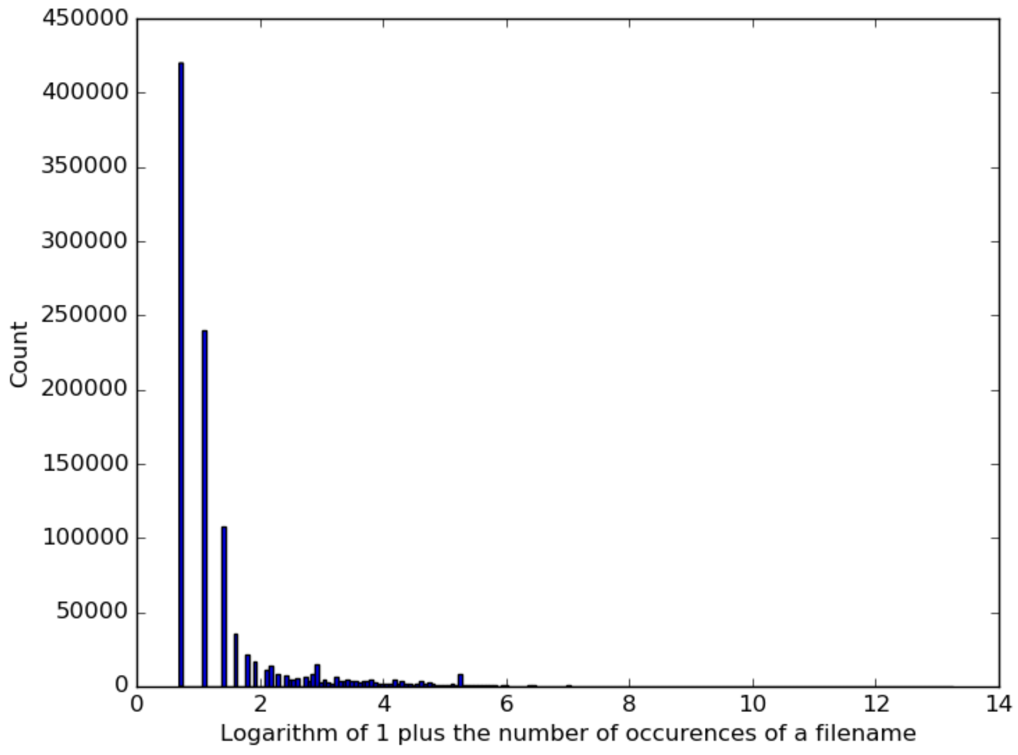


Figure 1: Distribution of the total number of occurrences of a filename in our corpus.

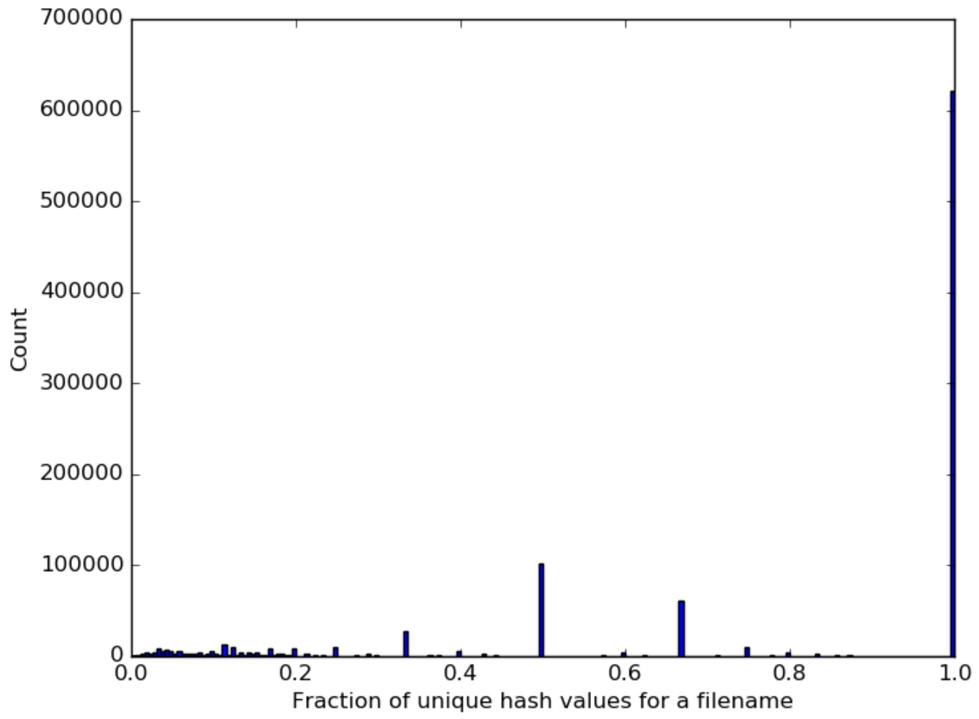


Figure 2: Fraction of distinct hash values (contents) per file name in our corpus.

Table 1 shows the most common types of executables in our corpus based on their extensions. For the experiments reported here we retrieved only files in the DLL and EXE formats, the two most common program-executable formats; EXE is a general-purpose format, and DLL is proprietary to Microsoft and used by most their software. There were 2049 sample files in 56 file families listed in Table 2. Each family had from 2 to 105 distinct file contents as indicated by their MD5 hashcodes. We are currently adding further files to study.

Table 1: The most common file extensions of executables in our corpus.

Exten-sion	Count	Exten -sion	Count	Exten-sion	Count	Exten-sion	Count
dll	16,188,565	exe	3,938,014	class	1,350,429	pyc	1,002,201
pyo	284,169	qtr	155,815	ocx	113,806	so	84,394
mexw32	77,185	dylib	65,895	rpm	60,250	com	56,636
obj	55,476	pyd	51,661	lxp	38,822	o	37,865
cls	26,856	beam	23,981	8bf	17,435		

In our experiments for each file family, we found all occurrences of its family name (e.g. “acrord32.dll” for the Adobe PDF reader in the 32-bit version) in the metadata of our corpus which included their MD5 hash values; this metadata was calculated during initial forensic analysis with the Fiwalk tool that produces data in DFXML format (Garfinkel, 2009); Fiwalk is now included with the SleuthKit (TSK) open-source tool (www.sleuthkit.org). We then used the SleuthKit “icat” command to extract a file for each hash value from the images in EWF format (Expert Witness Format). Since some corpus files could not be retrieved because of faulty drive images (many were disk drives sold to the used-hardware market when they failed), we tried retrievals for the same hash value until we found one that retrieved successfully with a nonempty result. With so many drives, usually we found an undamaged copy for each hash value for common software families. However, we noted around 5% of files had more than one modification time for the same file contents, something clearly incorrect. Some files matching on hash values did not have the family name because they were updates and caches with temporary names.

We excluded files over 1 megabyte from the file families since many were faulty extractions due to problems by the forensic software in finding the end of the file; executables are rarely over a million bytes, and files that large require considerable processing time to analyze anyway. We only considered files not marked for deletion in our experiments, as we have observed that metadata for deleted files can be unreliable (Rowe, 2016). Nonetheless, some undeleted files were faulty too as several kinds of things can go wrong is storing large amounts of data. We also extracted the modification times for these files from Fiwalk since modification time usually is set at the last change by the vendor and thus is a good indicator of the age of the version. For this we used the earliest modification time of all the instances that we could find of a hashcode.

Table 2: Executable file families used in our study.

Filename	a0003775.dll	acrord32.dll	bitsigd.dll	brmfbidi.dll
Hash count	16	6	6	5
Filename	bthserv.dll	ccalert.dll	cdfview.dll	deviceoperate.dll
Hash count	38	23	72	6
Filename	directdb.dll	dunzip32.dll	libscreen_plugin.dll	mfcm100u.dll
Hash count	88	34	37	37
Filename	mqrt.dll	msadcor.dll	msident.dll	mslbui.dll
Hash count	86	57	100	51
Filename	msnetobj.dll	msrdc.dll	nvrshu.dll	padrs804.dll
Hash count	72	70	32	26
Filename	perfctrs.dll	pmspl.dll	pngfilt.dll	rjcfspln.dll
Hash count	58	8	105	28
Filename	safslv.dll	scanmail.dll	spra0402.dll	tis_outlookx.dll
Hash count	30	2	6	7
Filename	typeaheadfind .dll	vsplugin.dll	w2k_lsa_auth.dll	webclnt.dll
Hash count	2	8	56	34
Filename	winprint.dll	wmpcd.dll	xrxwiadr.dll	
Hash count	8	54	13	
Filename	acrord32info.exe	charmap.exe	dns-sd.exe	dsndup.exe
Hash count	34	50	11	18
Filename	find.exe	hotfix.exe	iexplore.exe	mobsync.exe
Hash count	47	36	81	84
Filename	netscape.exe	nppagent.exe	nvudisp.exe	powerpnt.exe
Hash count	3	52	105	10
Filename	rtinstaller32.exe	snapview.exe	soundman.exe	udlaunch.exe
Hash count	4	15	60	4
Filename	uninstall_plugin .exe	wmplayer.exe	wmpshare.exe	wordicon.exe
Hash count	24	39	44	11
Filename	yserver.exe			
Hash count	16			

Methodology: Preliminary File Analysis

A simple technique of computing the entropy at periodic locations in a file gives a good indicator of its structure. We found plotting the byte entropy on consecutive 512-byte sequences worked well at indicating the parts of a file. Figure 3 shows a random sample of the segment entropies versus relative position in the executable file. There are clear patterns, with entropies around 6 (suggesting machine instructions) predominant in the front of the file, and more varied entropies (suggesting data) in the rest of the file. The plot indicates that most executables are in the form of a block of code followed by data.

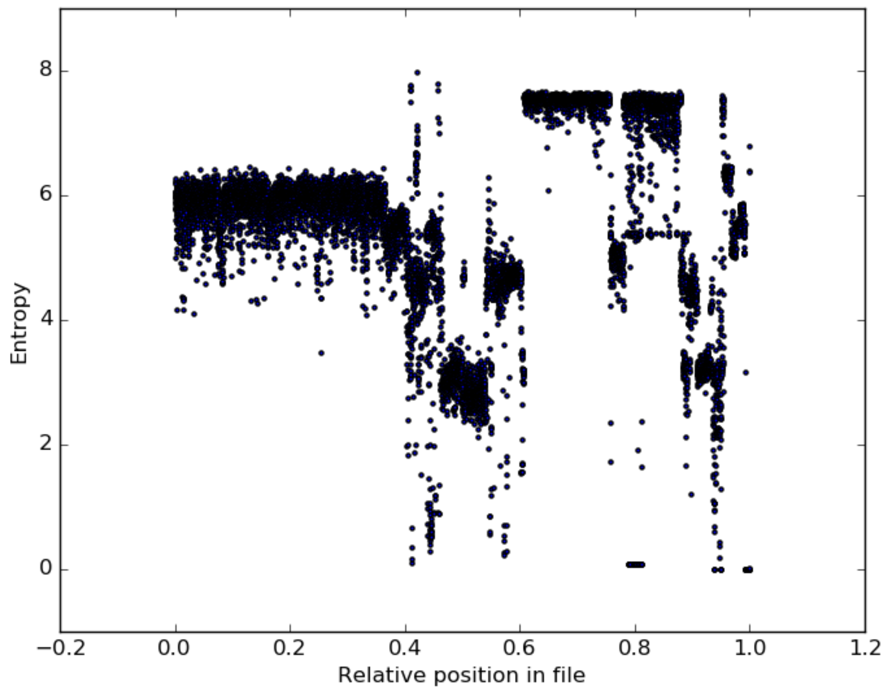


Figure 3: Scatter plot of a random sample of entropies of 512-byte segments versus relative position in a file.

More specific details can be found by examining individual files. The figures below show three versions of the file pacman.exe which were judged to be malware by at least one of the five tools used in (Rowe, 2016) of Bit9, OpenMalware, VirusShare, Symantec, and ClamAV. Figure 4 shows a typical pattern for an executable with a header including several bytes of zeros, machine instructions, and then data starting around location 180000. Figure 5 shows an executable with fewer machine instructions and more data. It also a very short header unlike the file in Figure 4; DLL formats have consistent headers, but these are EXEs and they are more varied. Figure 6 by contrast shows a pattern typical of encoded or compressed code (subjected to an "executable packer") with a short header and very high entropies for the rest of the file. It is important to recognize encoded and encrypted code because byte comparisons between two such files find only spurious matches. Since these three files were all malware, the pictures suggest that there is significant variation on malware, something true of the other malware files we saw in our corpus.

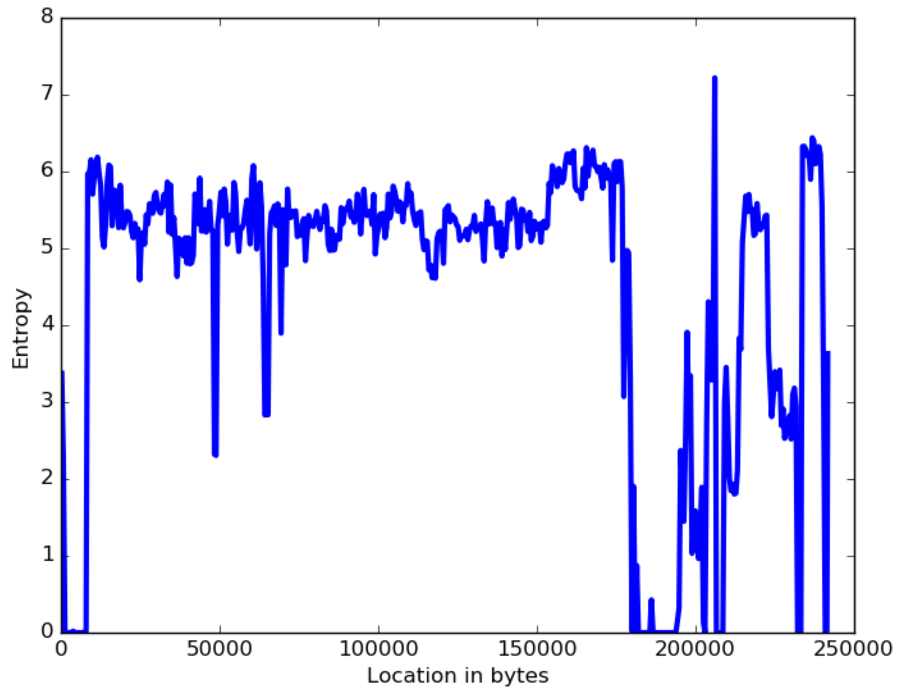


Figure 4: Distribution of 512-byte entropies in one pacman.exe file found in India.

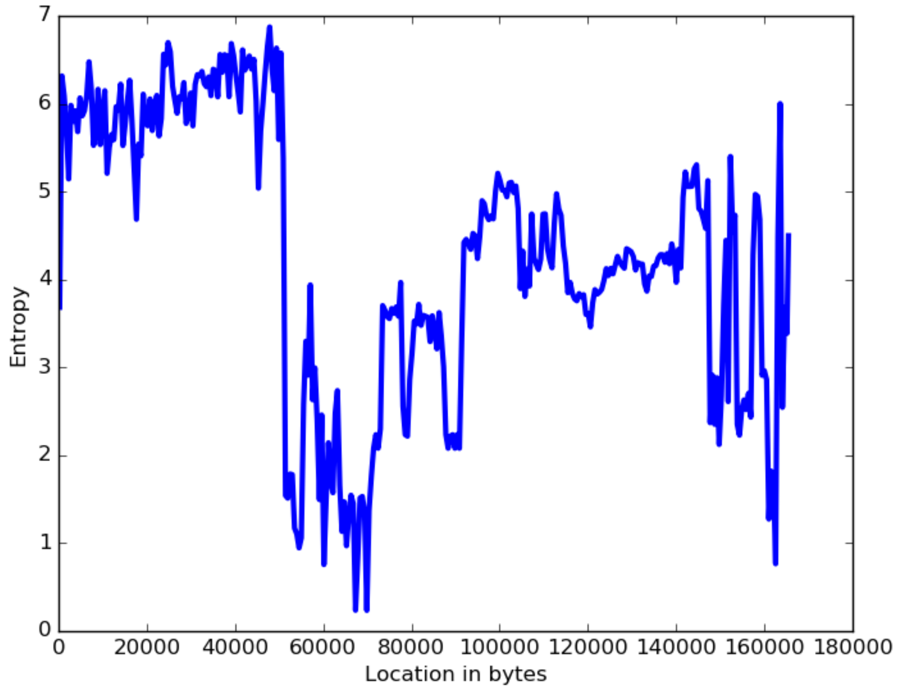


Figure 5: Distribution of 512-byte entropies in a pacman.exe file found in Thailand.

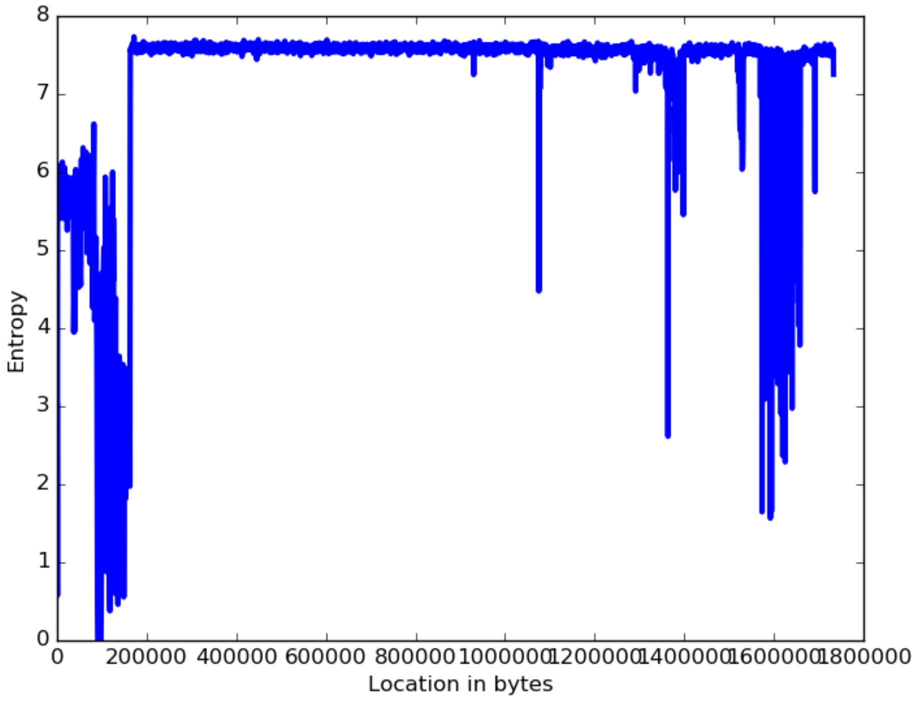


Figure 6: Distribution of 512-byte entropies in a third pacman.exe file found in Thailand.

We explored different window sizes of consecutive on which to calculate entropies. Figure 7 plots an msrdc.dll file with a window size of 512 and Figure 8 plots it with a window size of 4096. The larger window size clearly smooths the data but loses detail that could be important.

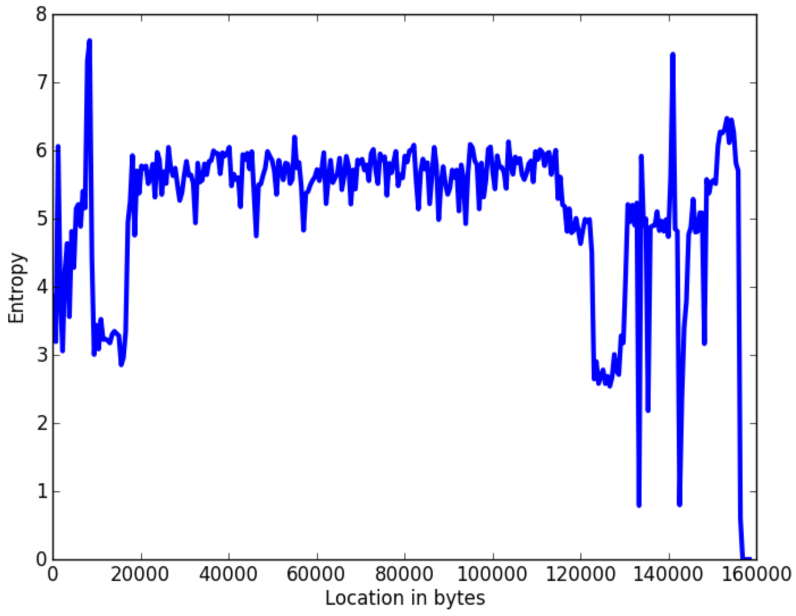


Figure 7: Entropies of an msrdc.dll file using 512-byte distributions.

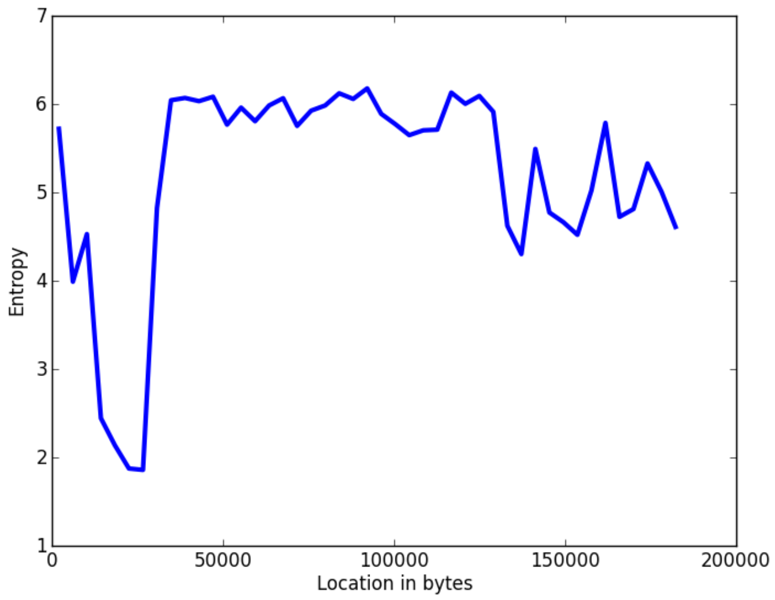


Figure 8: Entropies of an msrd.dll file using 4096-byte distributions.

Figure 9 shows the histogram of byte entropies on groups of successive 512 byte in our corpus of executables, and Figure 10 shows the portion of this > 0.2 to exclude the sharp peak at zero entropy. The peak at 7.5 represents encoded and compressed data; the peak at 5.9 represents machine instructions; the peak at 3.2 represents Ascii text; and the peaks from 0.5-1.5 represent standard labels (using 16-bit encoding of Ascii with alternate zero bytes). The peak at zero represents empty space, usually zero bytes, that are for data to be filled in later or to pad the file to a byte boundary that is a power of 2. Since these usages are incompatible with one another, it makes sense to partition the entropies into five ranges: 0-0.2, 0.2-2, 2-4, 4-6.7, and 6.5-8. Then it only makes sense to compare segments in the same entropy range.

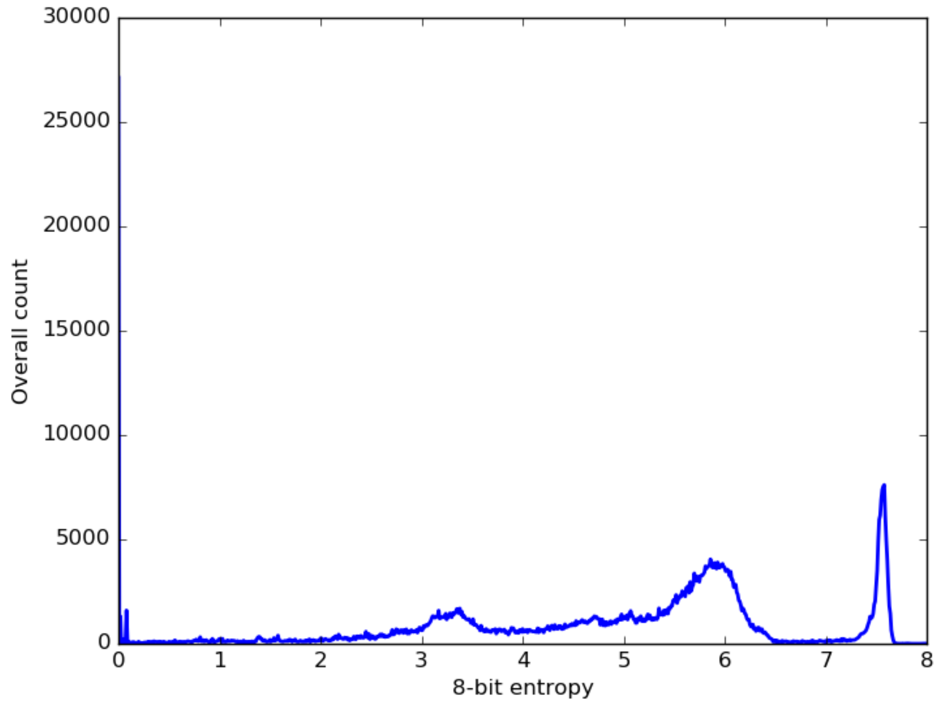


Figure 9: Histogram of average byte entropies on 512 successive bytes in our corpus of executables.

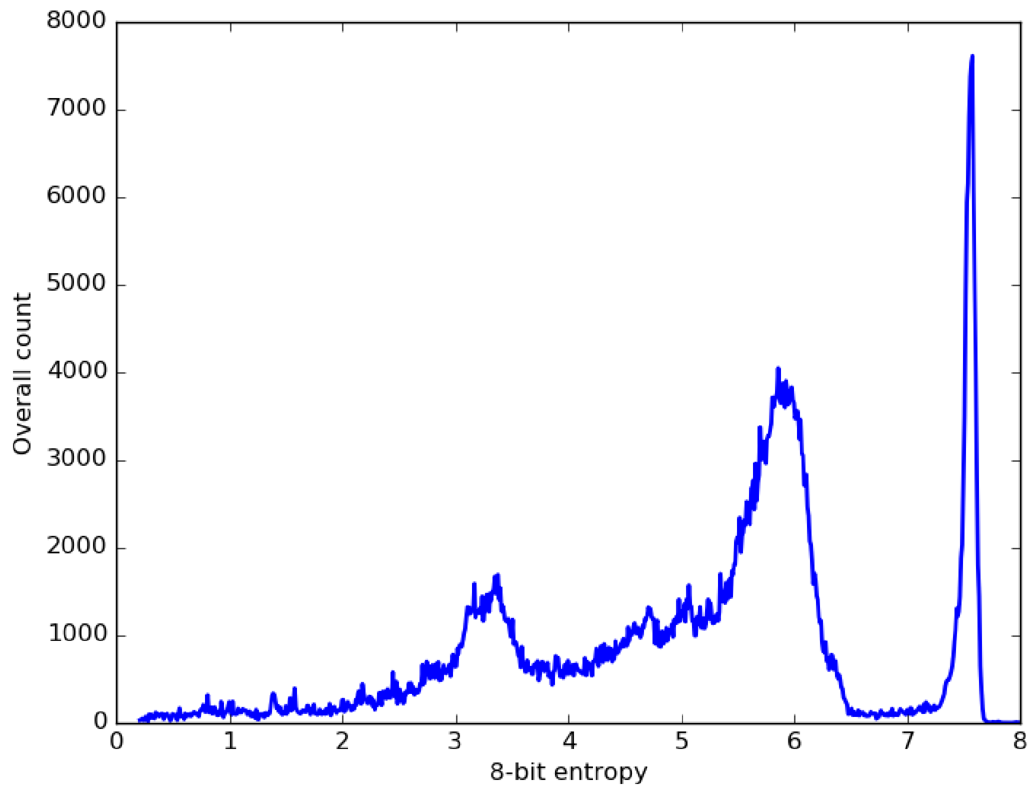


Figure 10: Histogram of average byte entropies > 0.2 on 512 successive bytes in our corpus of executables.

Methodology: Byte Comparison Methods

Randomly chosen executable files tend to have few similarities except for their (short) headers. Even executables from the same family were observed not to have many similarities because of the changes that occur with recompilation mentioned in section 1, though there are exceptions. However, most instruction codes and data constants will remain the same through different versions. To compare executable file versions, then, we need to look carefully to find those features of the files that we can match. The most basic and thorough approach is to compare bytes between two files to try to find matches. We used this as the "ground truth" for subsequent experiments with attempts to find faster methods. There are 256 possible byte values and we can index all their occurrences in two files we wish to compare. Then we can find all possible matches of the same byte value from one file to the other. Some will be coincidental since the probability of two random bytes matching is 1/256. That means that two random files of lengths N and M will have NM/256 matches on the average. However, if we can find a long sequence in one file that matches to a corresponding sequence in the other file, that can be good evidence beyond chance that we have found a block of code that has been shifted in location between the versions. The degree of shift in position from the first file to the second is termed an "offset". We can find likely good offsets by creating histograms of evidence for all offsets, and looking for the peaks of those histograms. We can then search for sequences having those offsets.

Figure 11 shows that the overall distribution of the byte values for the 56 file families in our test data is not uniform, with nonrandom patterns likely due the relative popularity of different instruction codes and operand patterns. We can use this to weight matches between files because matches on rare byte values are less ambiguous than matches on common bytes. We weighted counts of matches between the two files by the inverse of the number of occurrences of the byte value in the smaller file since the smaller file has more control on the match possibilities. We did completely exclude in this first phase any zero (00000000) bytes and all-one

(11111111) bytes because they are often used to indicate unused space, and unused space is not specific enough to provide good matches.

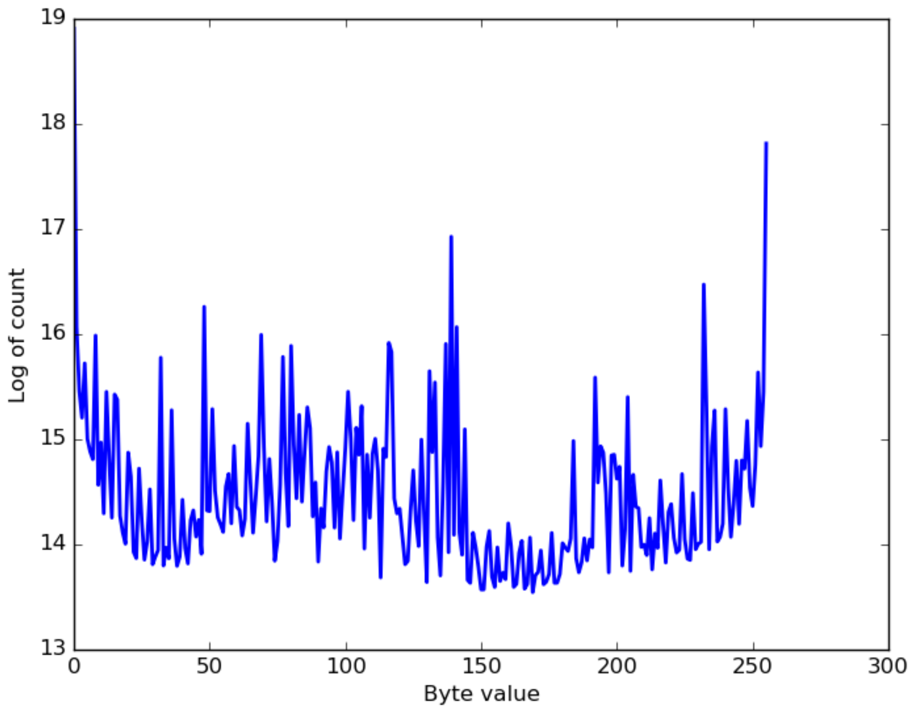


Figure 11: Distribution of byte values in all files in our test sample.

This gave us a distribution of weighted counts for each possible value of offset for a pair of files. We computed the mean and standard deviation of the offset counts. Since with only 256 possible byte values, many matches between two files are spurious, we recomputed the mean and standard deviation of the weighted offset counts within two standard deviations of the mean, then excluded everything less than two standard deviations above this mean. Since meaningful block matches will have significantly higher weighted counts than the average, this excludes many useless matches, reducing the data by roughly a factor of 100. The mean was computed twice to exclude the largest values the second time, since we found many meaningful matches of blocks that had considerably higher offset counts than the average offset.

Then for the most popular offsets, we go back to the pair of files and look for consecutive bytes at those offsets. Unfortunately, we cannot use any of the classic algorithms for subsequence matching (Bergroth et al, 2000) because the subsequences to be found cannot be enumerated in advance. We also look for matches on every 2nd (alternating) byte, on every 4th byte (which helps identify matching instruction codes on 32-bit machines), and every 8th byte (which helps identify matching instruction codes on 64-bit machines). The minimum length of a sequence considered was 8 matches for each of the four types of matches. The following figures show a visual display of this matching for two files each. Blue indicates blocks of successive bytes that match between the two files, green indicates blocks matching every 2nd byte magenta indicates blocks matching every 4th byte, red indicates blocks matching every 8th byte, and white indicates areas for which no match could be found. Black rectangles indicate the extent of the files. It can be seen that files in Figure 12 do not have much in common as the files in Figure 13.

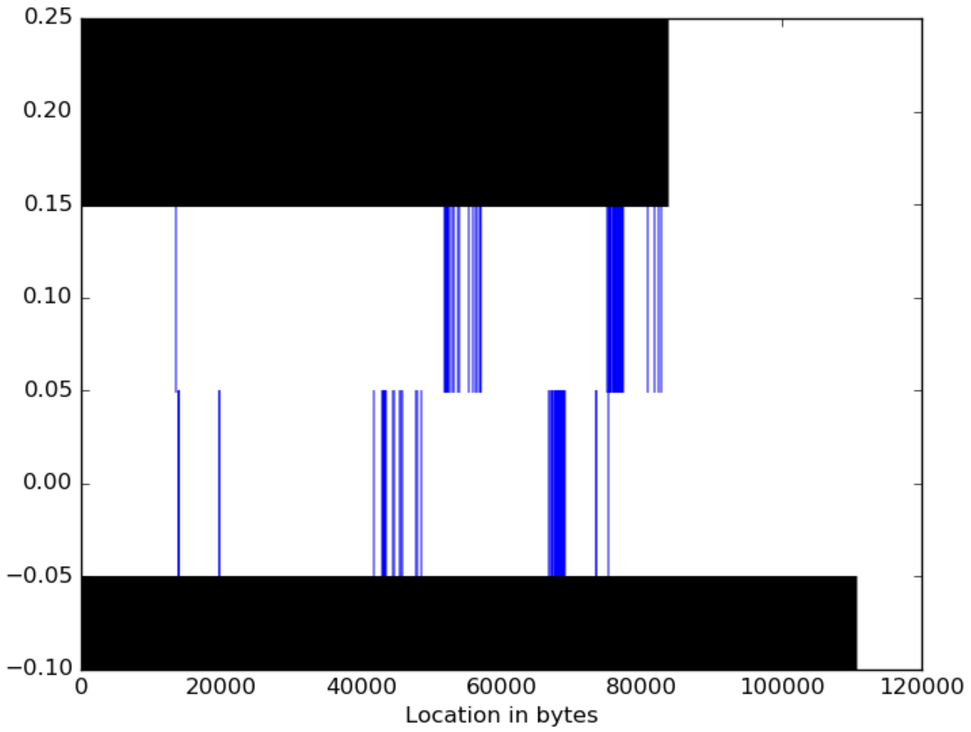


Figure 12: Byte matches between two versions of a file that do not have many similarities.

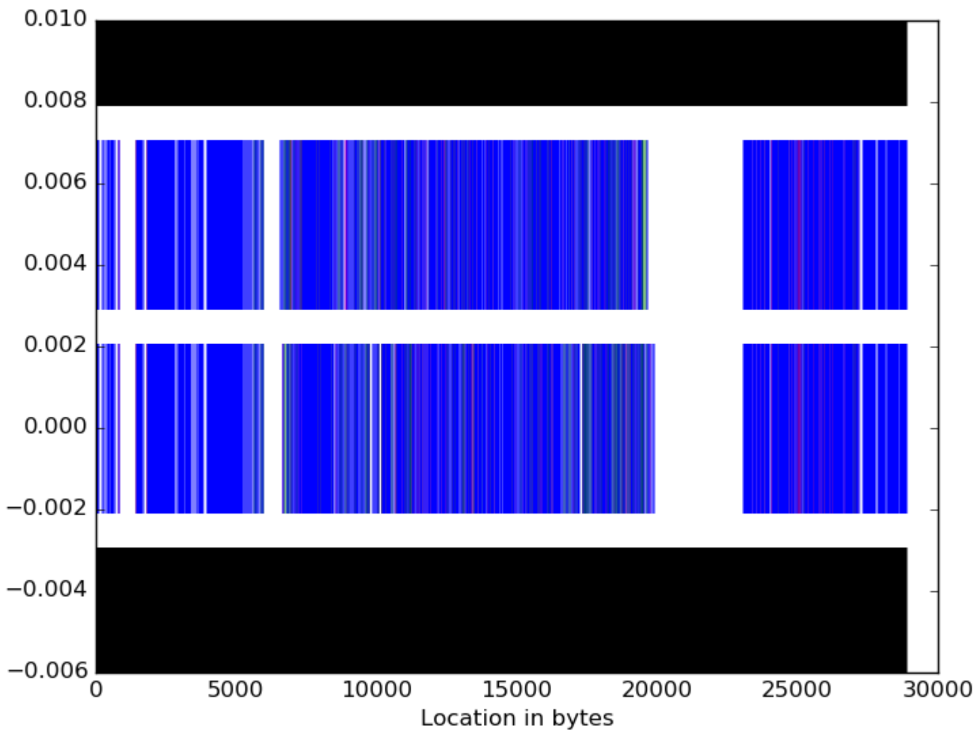
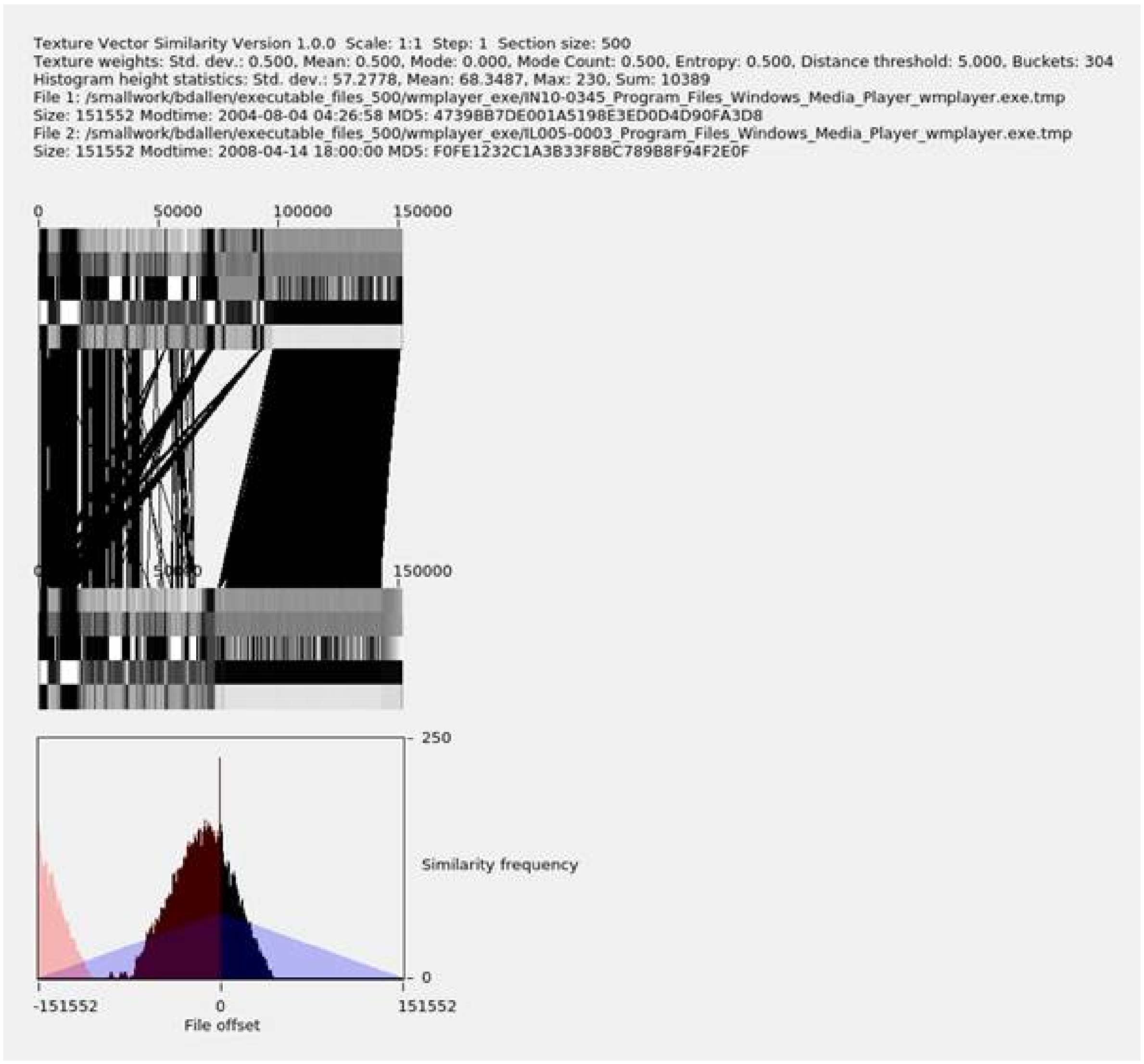


Figure 13: Byte comparison between two similar versions of the same file.

A disadvantage of the previous diagrams is they fail to indicate what matches exactly, so we developed an alternative visualization (Figure 14) (Allen, 2019). The first five and last five rows of the image visualize five “texture-vector” components. These components were the entropy of the segment byte values, mean byte value, standard deviation of the byte values, mode (most common frequency) of the byte values, and frequency of the mode. Later experiments also used the median. Distances were computed as the Euclidean distance between two texture vectors; similarities are the inverses of distances. Lines between the two bars representing files indicate strong similarities between the texture vectors in the files.



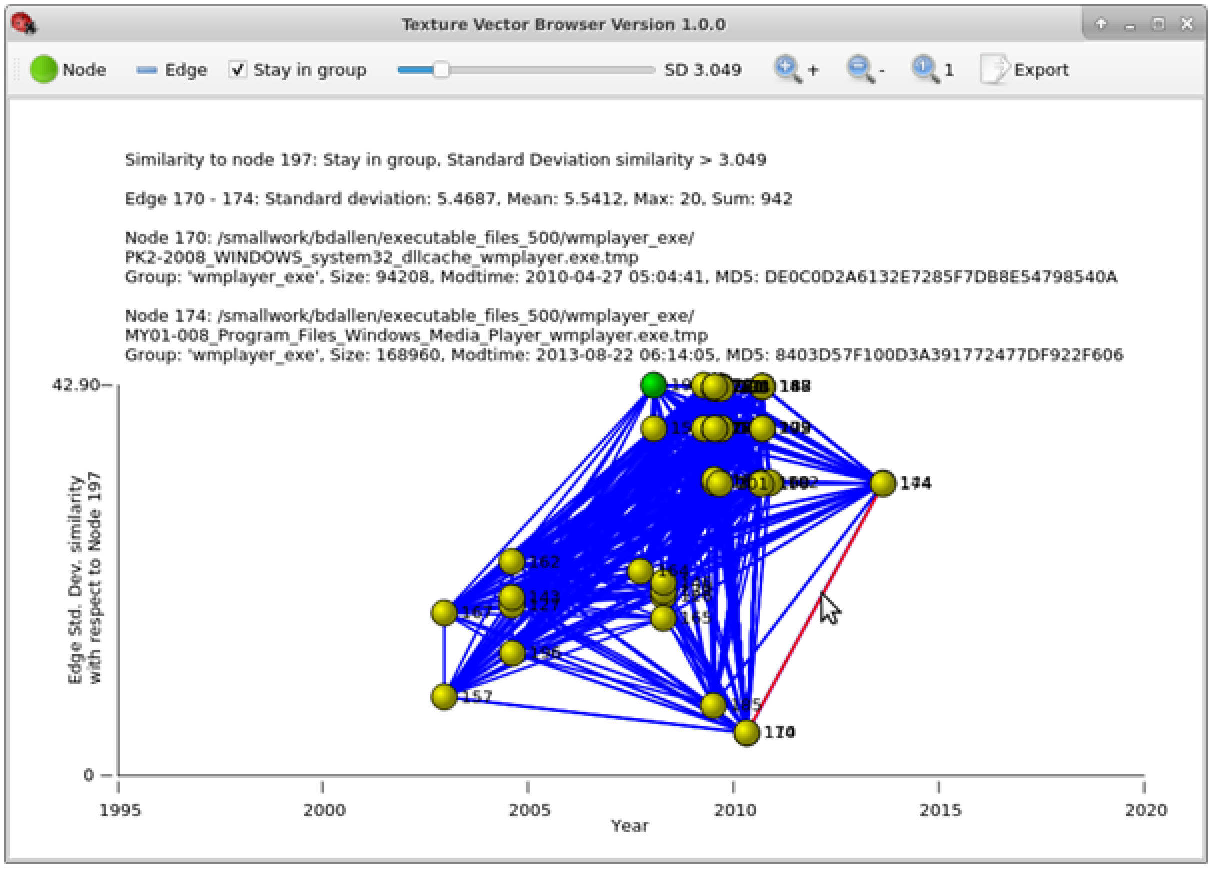
```
Texture Vector Similarity Version 1.0.0  Scale: 1:1  Step: 1  Section size: 500
Texture weights: Std. dev.: 0.500, Mean: 0.500, Mode: 0.000, Mode Count: 0.500, Entropy: 0.500, Distance threshold: 5.000, Buckets: 304
Histogram height statistics: Std. dev.: 57.2778, Mean: 68.3487, Max: 230, Sum: 10389
File 1: /smallwork/bdallen/executable_files_500/wmplayer_exe/IN10-0345_Program_Files_Windows_Media_Player_wmplayer.exe.tmp
Size: 151552 Modtime: 2004-08-04 04:26:58 MD5: 4739BB7DE001A5198E3ED0D4D90FA3D8
File 2: /smallwork/bdallen/executable_files_500/wmplayer_exe/IL005-0003_Program_Files_Windows_Media_Player_wmplayer.exe.tmp
Size: 151552 Modtime: 2008-04-14 18:00:00 MD5: F0FE1232C1A3B33F8BC789B8F94F2E0F
```

Figure 14: Example comparison of two files using their texture vectors and showing connections between file portions.

THIS PAGE LEFT INTENTIONALLY BLANK

Results: File Family Visualization

We also developed visualization techniques for families of files. Figure 15 shows a first version (Allen, 2019) giving a summary diagram for the file versions in the wmplayer.exe family. Yellow dots represent file versions, and lines connecting them indicate degrees of similarity of those files sufficiently similar. Green nodes and red edges are those that have been highlighted for additional study by the user with the interface. This red edge was hovered over by the cursor spanning nodes 170 and 174, along with detail about this edge and these two nodes. All nodes are within the same file family because "Stay in group" is selected. The similarity slider is set so that edges with a similarity measure less than 3.049 are not shown. The similarity value used was the texture-vector similarity described previously.



Similarity to node 197: Stay in group, Standard Deviation similarity > 3.049

Edge 170 - 174: Standard deviation: 5.4687, Mean: 5.5412, Max: 20, Sum: 942

Node 170: /smallwork/bdallen/executable_files_500/wmplayer_exe/
PK2-2008_WINDOWS_system32_dllcache_wmplayer.exe.tmp
Group: 'wmplayer_exe', Size: 94208, Modtime: 2010-04-27 05:04:41, MD5: DE0C0D2A6132E7285F7DB8E54798540A

Node 174: /smallwork/bdallen/executable_files_500/wmplayer_exe/
MY01-008_Program_Files_Windows_Media_Player_wmplayer.exe.tmp
Group: 'wmplayer_exe', Size: 168960, Modtime: 2013-08-22 06:14:05, MD5: 8403D57F100D3A391772477DF922F606

Figure 15: Example visualization of a file family.

Figure 16 shows a screenshot of the results of a zoom on the above graph, omitting edges with a similarity measure of less than 28.373. In this screenshot, node 152 is hovered over. The display shows information about node 152, but this information is not visible because the view is scrolled down.

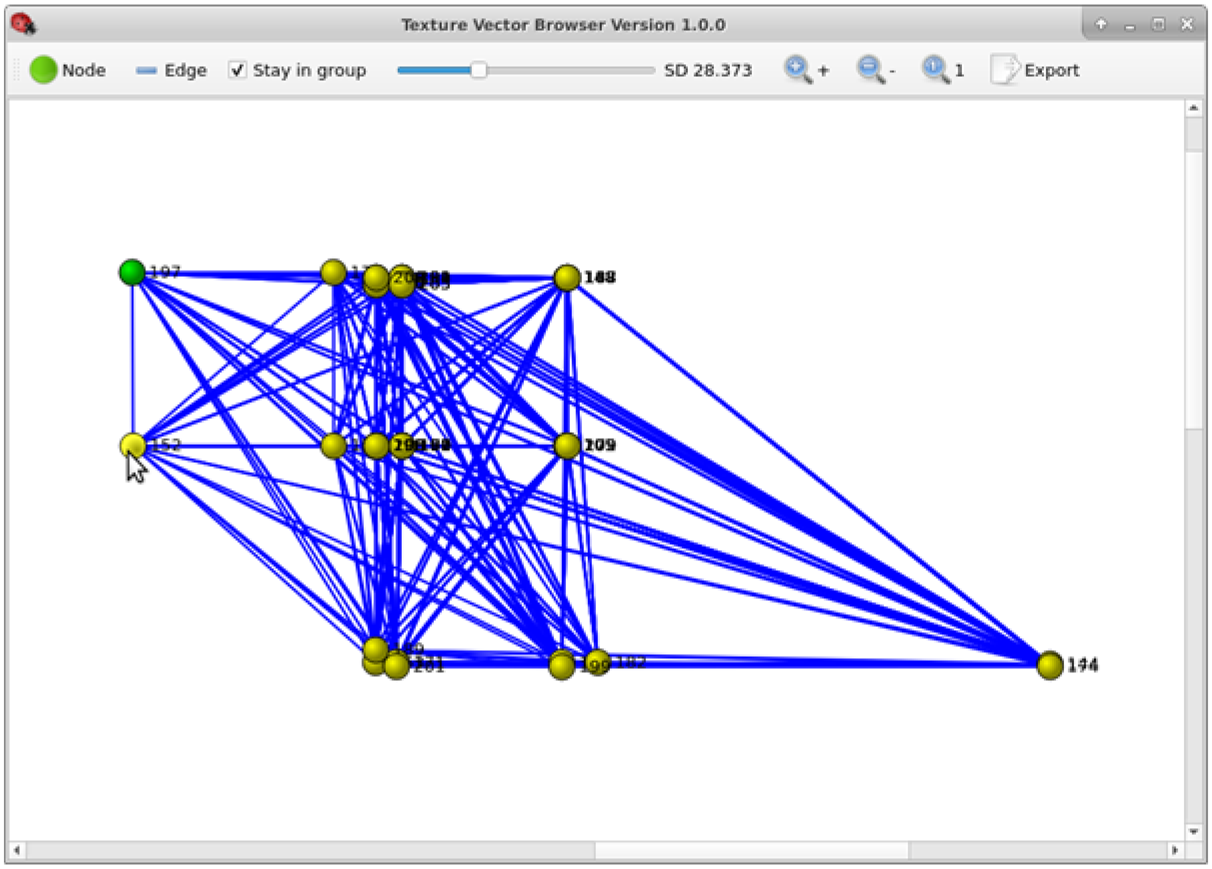
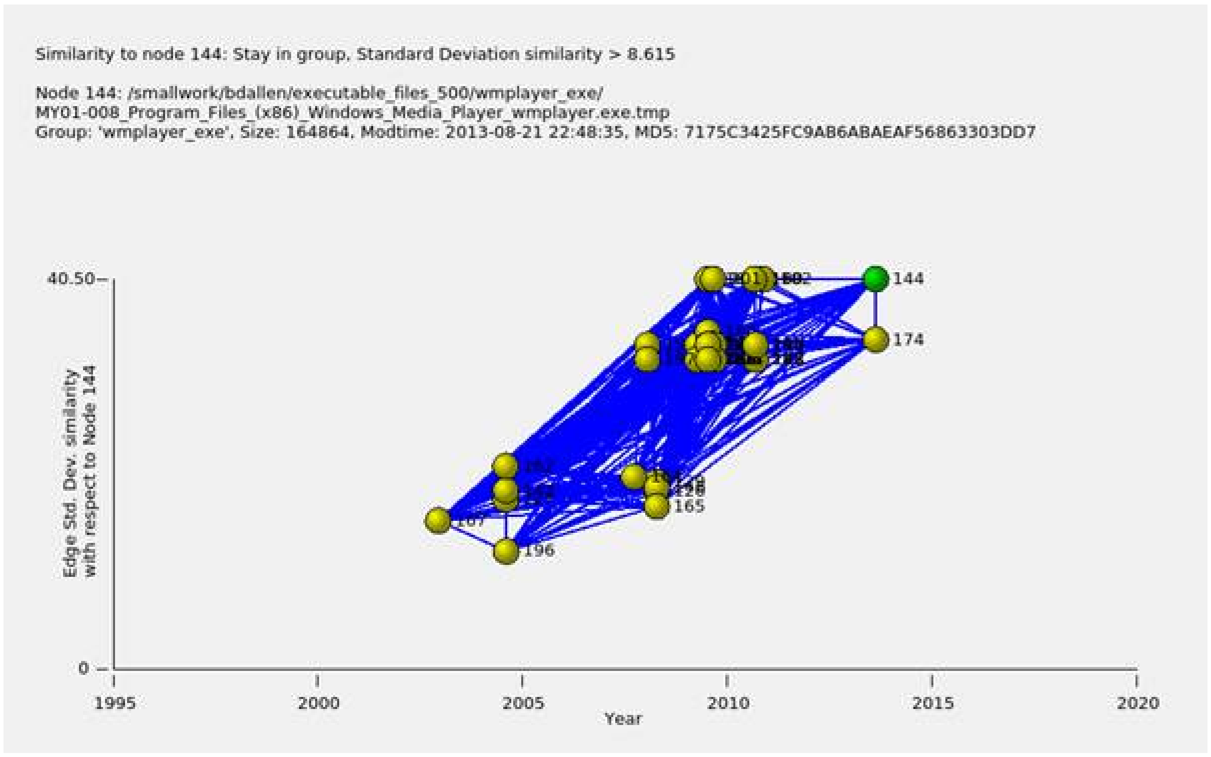


Figure 16: Results of a zoom on the previous figure.

Figure 17 shows what happens when we click on node 144. It becomes green and similarity along the y-axis is with respect to node 144. By clicking on "Export" we get this exported view showing the timeline similarity graph for node 144, which we tuned to reject similarities less than 8.615.


Similarity to node 144: Stay in group, Standard Deviation similarity > 8.615

Node 144: /smallwork/bdallen/executable_files_500/wmplayer_exe/
MY01-008_Program_Files_(x86)_Windows_Media_Player_wmplayer.exe.tmp
Group: 'wmplayer_exe', Size: 164864, Modtime: 2013-08-21 22:48:35, MD5: 7175C3425FC9AB6ABAEAF56863303DD7

Figure 17: Details of files similar to node 144.

Figure 18 shows a second version of our visualization as a tree, for the family of cdfview.dll, a common Microsoft Windows file. Modification times should be equal for files with the same hash value since they are the times when the vendor last modified the file, so we have plotted only one file for each hash value. Here the horizontal axis is time and the vertical axis represents cumulative similarity calculated by adding similarities for related descendants in each family. In the Figure, lines connect file versions which are related, defined as those having more than a threshold similarity which was 0.5 for this diagram. Red dots indicate malicious files as judged by at least one of five malware detection tools: Bit9, OpenMalware, VirusShare, Symantec, and ClamAV. More background on our malware-analysis infrastructure is given in (Rowe, 2015).

Similarity here was defined differently than for the previous visualizations, using a sigmoid function on the sum of the sizes of the ten longest sequence matches between the two files, $s = x/(x + 2000)$ where $x = \sum_{i=1}^{10} l_i$ where l_i is the length of the ith longest sequence of consecutive byte matches between the two files. The sigmoid function was important to flatten the effects of extreme values of matches since the sum varied widely from 0 to millions with our file families. We started using this similarity measure when further experiments (in Section 7) questioned the valued of texture-vector similarity.

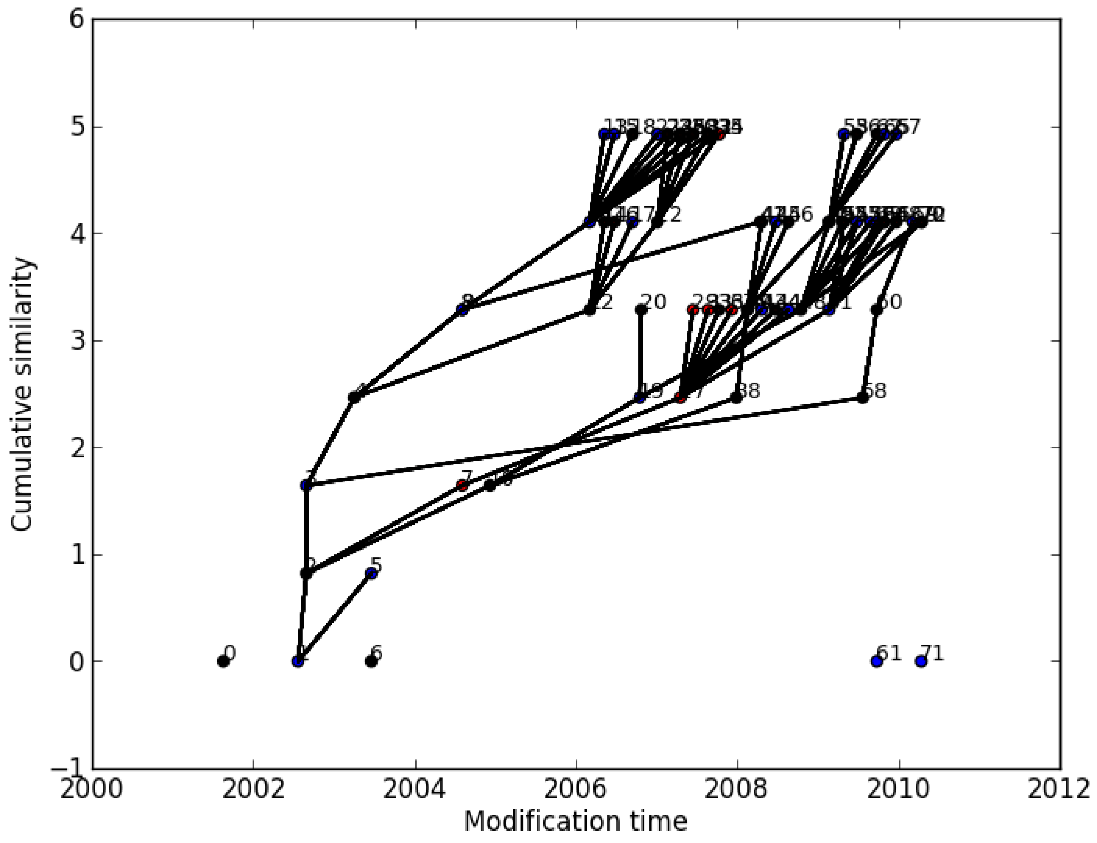


Figure 18: Tree visualization of the 73 distinct files of files named cdfview.dll in our corpus.

The complete list of 73 files retrieved and analyzed for the cdfview.dll family is given below. The index numbers given below for the files are also used in the Figure but some are hard to see. The first part of each file name is its drive name starting with 2-character country code, and the rest of the filename is the path to the file in the directory structure of the operating system.

0: SG1-1066/WINNT/system32/cdfview.dll
1: HK99-6001/WINNT/$NtServicePackUninstall$/cdfview.dll

```
2: HK99-6001/WINNT/$NtUninstallKB896688-IE6SP1-
20051004.130236$/cdfview.dll
3: IL3-0207/WINNT/ $NtUninstallKB972260-IE6SP1-
20090722.120000$/cdfview.dll
4: DE001-0003/Muell/$NtUninstallKB912812-IE6SP1-
20060322.182418$/cdfview.dll
5: HK99-6001/WINNT/ServicePackFiles/i386/cdfview.dll
6: SG1-1066/WINNT/ServicePackFiles/i386/cdfview.dll
7: AE10-0025/WINDOWS/system32/cdfview.dll
8: MA1001-0001/WINDOWS/system32/dllcache/cdfview.dll
9: DE001-0003/Muell/$NtUninstallKB912812$/cdfview.dll
10: HK99-6001/WINNT/system32/dllcache/CDFVIEW.DLL
11: DE001-0003/Muell/$NtUninstallKB916281$/cdfview.dll
12: DE001-0003/Muell/$hf_mig$/KB912812/SP2QFE/cdfview.dll
13: DE001-0003/Muell/$NtUninstallKB918899$/cdfview.dll
14: DE001-0003/Muell/$hf_mig$/KB916281/SP2QFE/cdfview.dll
15: DE001-0003/Muell/$NtUninstallKB922760$/cdfview.dll
16: DE001-0003/Muell/$hf_mig$/KB918899/SP2QFE/cdfview.dll
17: DE001-0003/Muell/$hf_mig$/KB922760/SP2QFE/cdfview.dll
18: DE001-0003/Muell/$NtUninstallKB928090$/cdfview.dll
19: PS01-069/WINNT/$NtUninstallKB925454-IE6SP1-
20061116.120000$/cdfview.dll
20: PS01-069/WINNT/$NtUninstallKB928090-IE6SP1-
20070125.120000$/cdfview.dll
21: DE001-0003/Muell/$NtUninstallKB931768$/cdfview.dll
22: DE001-0003/Muell/$hf_mig$/KB928090/SP2QFE/cdfview.dll
23: DE001-0003/Muell/$NtUninstallKB933566$/cdfview.dll
24: DE001-0003/Muell/$hf_mig$/KB931768/SP2QFE/cdfview.dll
25: DE001-0003/Muell/$NtUninstallKB937143$/cdfview.dll
26: DE001-0003/Muell/$hf_mig$/KB933566/SP2QFE/cdfview.dll
27: IL005-0006/WINDOWS/$NtUninstallKB937143$/cdfview.dll
28: DE001-0003/Muell/$NtUninstallKB939653$cdfview.dll
29: IL005-0006/WINDOWS/$NtUninstallKB939653$/cdfview.dll
30: DE001-0003/Muell/$hf_mig$/KB937143/SP2QFE/cdfview.dll
31: DE001-0003/Muell/$hf_mig$/KB939653/SP2QFE/cdfview.dll
32: DE001-0003/Muell/$NtUninstallKB942615$/cdfview.dll
33: IL005-0006/WINDOWS/$NtUninstallKB942615$/cdfview.dll
34: DE001-0003/Muell/$hf_mig$/KB942615/SP2QFE/cdfview.dll
35: DE001-0003/Muell/system32/dllcache/cdfview.dll
36: IL005-0006/WINDOWS/$NtUninstallKB944533$/cdfview.dll
37: IL005-0006/WINDOWS/$NtUninstallKB947864$/cdfview.dll
38: IN10-0317/WINDOWS/system32/dllcache/cdfview.dll
39: IL005-0006/WINDOWS/$NtUninstallKB950759$/cdfview.dll
40: IL005-0006/WINDOWS/$hf_mig$/KB947864/SP2QFE/cdfview.dll
41: MA1001-0003/WINDOWS/system32/dllcache/cdfview.dll
42: IL005-0006/WINDOWS/$hf_mig$/KB950759/SP2QFE/cdfview.dll
```

43: IL005-0006/WINDOWS/$NtUninstallKB953838$/cdfview.dll
44: IL005-0006/WINDOWS/$NtUninstallKB956390$/cdfview.dll
45: IL005-0006/WINDOWS/hf_mig/KB953838/SP2QFE/cdfview.dll
46: IL005-0006/WINDOWS/hf_mig/KB956390/SP2QFE/cdfview.dll
47: IL005-0006/WINDOWS/$NtUninstallKB958215$/cdfview.dll
48: IL005-0007/WINNT/$NtUninstallKB963027-IE6SP1-20090303.120000$/cdfview.dll
49: IL005-0007/WINNT/$NtUninstallKB969897-IE6SP1-20090501.120000$/cdfview.dll
50: IL005-0006/WINDOWS/hf_mig/KB963027/SP2QFE/cdfview.dll
51: IL005-0006/WINDOWS/$NtUninstallKB969897$/cdfview.dll
52: IL005-0007/WINNT/$NtUninstallKB972260-IE6SP1-20090722.120000$/cdfview.dll
53: IL005-0006/WINDOWS/hf_mig/KB969897/SP2QFE/cdfview.dll
54: IL005-0006/WINDOWS/$NtUninstallKB972260$/cdfview.dll
55: IL005-0007/WINNT/$NtUninstallKB974455-IE6SP1-20090925.120000$/cdfview.dll
56: IL005-0006/WINDOWS/hf_mig/KB972260/SP2QFE/cdfview.dll
57: IL005-0006/WINDOWS/$NtUninstallKB974455$/cdfview.dll
58: IL3-0207/WINNT/system32/dllcache/CDFVIEW.DLL
59: IL005-0007/WINNT/$NtUninstallKB976325-IE6SP1-20091027.120000$/cdfview.dll
60: IL3-0207/WINNT/SoftwareDistribution/Download/d5d55eaac3e837d022d68f8271168a8d/rtmgdr/cdfview.dll
61: IL005-0007/WINNT/system32/CDFVIEW.DLL
62: IL005-0006/WINDOWS/hf_mig/KB974455/SP2QFE/cdfview.dll
63: IL005-0006/WINDOWS/$NtUninstallKB976325$/cdfview.dll
64: IL005-0007/WINNT/$NtUninstallKB982381-IE6SP1-20100414.120000$/cdfview.dll
65: IL005-0006/WINDOWS/hf_mig/KB976325/SP2QFE/cdfview.dll
66: IL005-0006/WINDOWS/$NtUninstallKB978207/$cdfview.dll
67: IL005-0006/WINDOWS/hf_mig/KB978207/SP2QFE/cdfview.dll
68: IL005-0006/WINDOWS/$NtUninstallKB980182$/cdfview.dll
69: IL3-0207/WINNT/SoftwareDistribution/Download/9169eacb0c4e7dc5028638599c92ac2e/rtmgdr/cdfview.dll
70: IL005-0007/WINNT/SoftwareDistribution/Download/952d4c1e3e0c27aba62f38820c3fe08a/rtmgdr/cdfview.dll
71: IL005-0007/WINNT/system32/CDFVIEW.DLL
72: IL005-0006/WINDOWS/system32/cdfview.dll

However, there are many files with identical contents (hash codes) in our corpus; 1426 files in the corpus had name cdfview.dll or had a hash value of a file having this file name. For example, here is the full list of files matching Hashcode 27 of 2595DCEF6BDAAD19B3A7F825CD5493DC. In order, three are from Israel on three different drives, eight are from Palestine on six different drives, five are from Singapore on five different drives, and six more are from Israel on four different drives. The A0004153.dll and A0004174 are files matching on a hash value rather than a name.

```
IL005-0006/WINDOWS/$NtUninstallKB937143$/cdfview.dll
IL006-0004/WINDOWS/$NtUninstallKB937143$/cdfview.dll
IL3-0205/WINDOWS/$NtUninstallKB937143$/cdfview.dll
PS01-015/WINDOWS/system32/cdfview.dll
PS01-015/WINDOWS/system32/dllcache/cdfview.dll
PS01-018/WINDOWS/$NtUninstallKB937143$/cdfview.dll
PS01-018/Program Files/SAP/FrontEnd/SAPgui/Lang/lgndllCS.txt
PS01-023/WINDOWS/$NtUninstallKB937143$/cdfview.dll
PS01-030/WINDOWS/$NtUninstallKB937143$/cdfview.dll
PS01-
070/WINDOWS/SoftwareDistribution/Download/493760be868721503b9abd615f
71e312/SP2GDR/cdfview.dll
PS01-076/WINDOWS/$NtUninstallKB937143$/cdfview.dll
SG001-7020/WINDOWS/$NtUninstallKB937143$/cdfview.dll
SG001-7021/WINDOWS/$NtUninstallKB937143$/cdfview.dll
SG1-1052/WINDOWS/$NtUninstallKB937143$/cdfview.dll
SG1-1063/WINDOWS/$NtUninstallKB937143$/cdfview.dll
SG1-1064/WINDOWS/$NtUninstallKB937143$/cdfview.dll
il2-0034/System/System Volume Information//restore{B8B901F5-EACA-4604-
957F-6EF34D7ECA70}/RP54/A0004153.dll
il2-0034/System/System Volume Information//restore{B8B901F5-EACA-4604-
957F-6EF34D7ECA70}/RP54/A0004174.dll
il2-0034/System/WINDOWS/$NtUninstallKB933566$/cdfview.dll
il3-0118/WINDOWS/$NtUninstallKB937143$/cdfview.dll
il3-0172/WINDOWS/$NtUninstallKB937143$/cdfview.dll
il3-0181/WINDOWS/$NtUninstallKB937143$/cdfview.dll
```

Figure 19 shows a different kind of visualization for this data, the modification times of all 1426 files having name cdfview.dll or a hash value matching one such file. There are several incorrect reported times, but these did not appear in the above data because the drives they came from were generally faulty and we could not retrieve files from them. Color encodes the last digit of the file number.

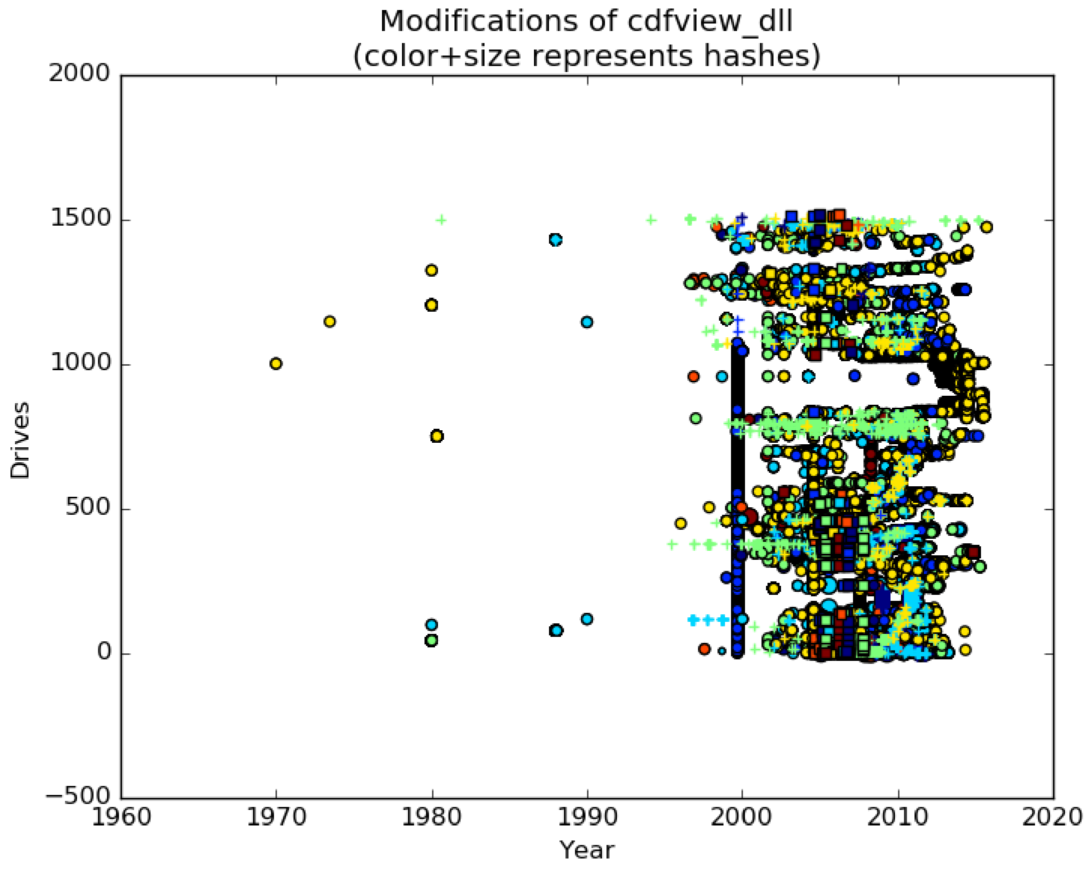


Figure 19: Modification times reported for all instances of cdfview.dll and other files having hash values of cdfview.dll.

Figure 20 shows yet another visualization, of the relationships between the files based solely on similarity. Distance between files was approximated by one minus the similarity of the files. However, finding 2N coordinates in two dimensions for N*N files is an overdetermined problem and needed an approximation achieved by optimization. We minimized the absolute value of the logarithm of the ratios of the achieved distanced to the desired distance, as described in (Rowe, 2018).

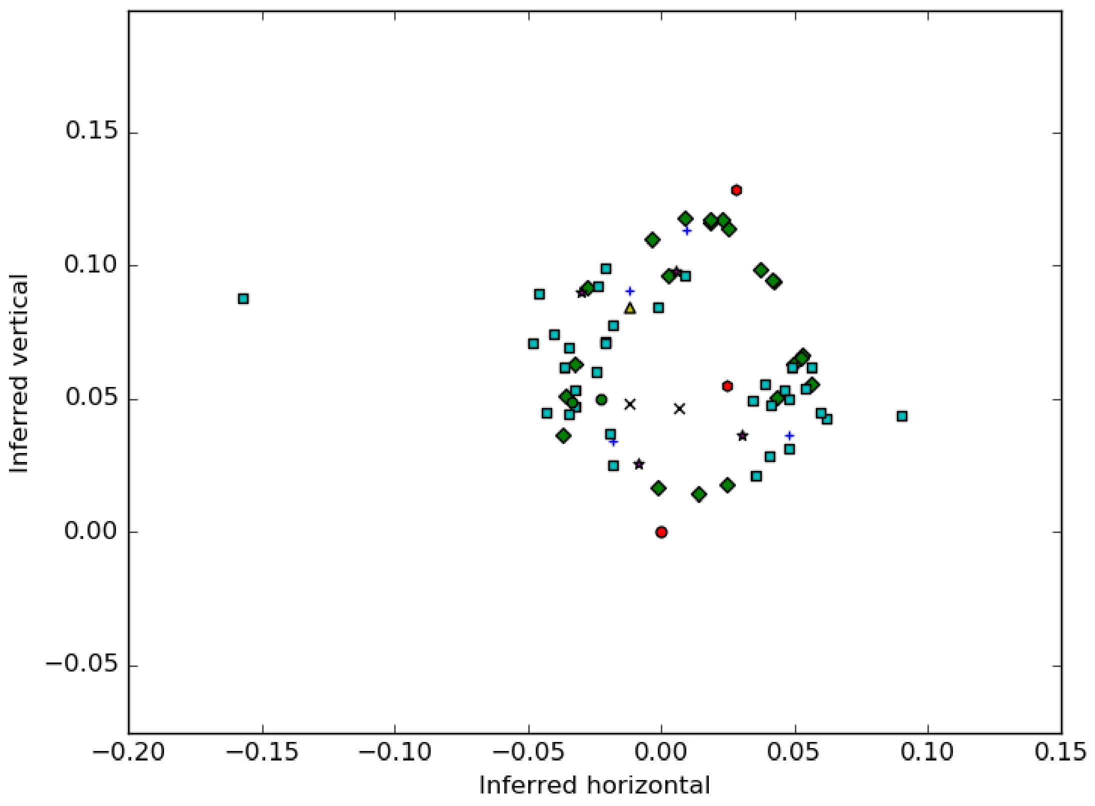


Figure 20: Visualization of the relationships between the cdfview.dll versions where distance is approximated by 1 minus similarity.

Here are two more tree diagrams of file families. Figure 21 shows the file family of ntevt.dll in our corpus. There was one main product line in the middle of the period, plus an earlier line and a later line. They were apparently for different operating systems because they overlap in time. Two long vertical sequences involve different machines for each hashcode, so apparently a suite of somewhat different versions was created for different operating systems or hardware. Most of the dots at height 0.0 appear to be based on incorrect assignment of “ntevt.dll” as the filename by faulty drives. Figure 22 shows the file family of w2k_lsa_auth.dll in our corpus. This shows one predominant product line with a few variations on it.

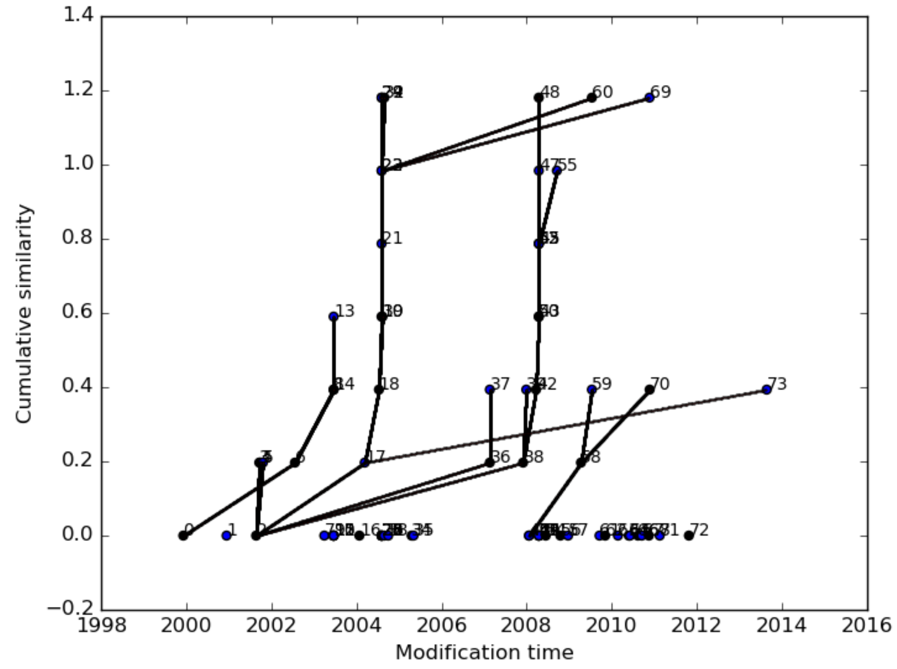


Figure 21: Tree visualization of the 74 files of nvevt.dll in our corpus.

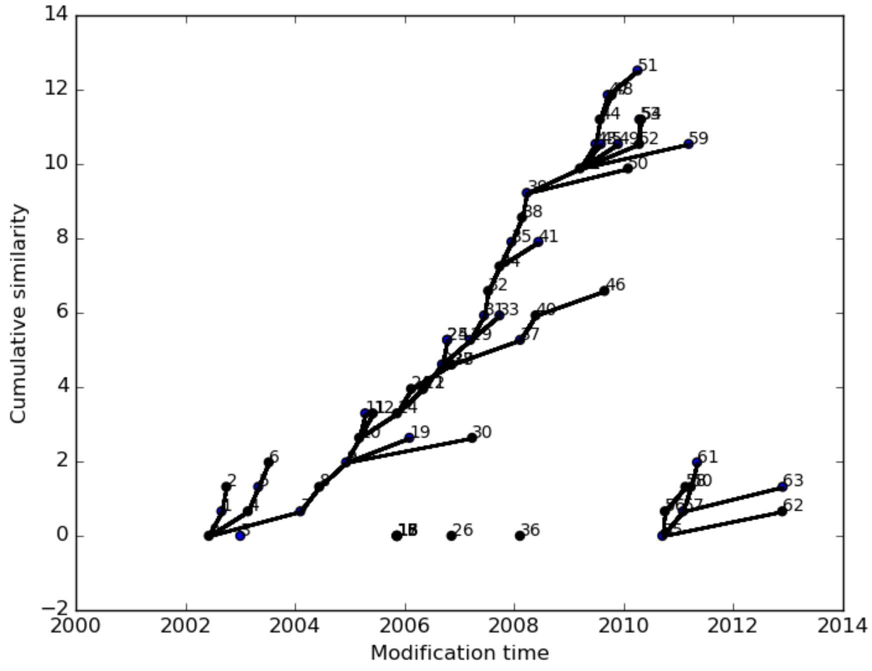


Figure 22: Tree diagram of 64 files of w2k_lsa_auth.dll in our corpus.

Results: Attempts to Improve Processing Speed

Since the “ground truth” approach of obtaining every possible match between bytes of two files and looking for offset patterns required considerable processing time, we experimented with alternative methods to find matches between bytes of files. These can also be used in preprocessing to rule out obviously dissimilar pairs of files.

One approach is to skip the time-consuming process of matching offsets and just use the heights of the most-likely offsets to indicate overall pair similarity. Figure 23 shows a plot of the sum of counts of the evidence for the 10 highest offsets in file comparisons versus the ground-truth (sequence) similarity in a random sample of 2000 file pairs. The figure shows that offset heights increase with ground truth but not smoothly, though high values of both were correlated. It also shows that this “peak” metric seems useful for preprocessing filtering but is not sufficient alone to indicate similarity of two files.

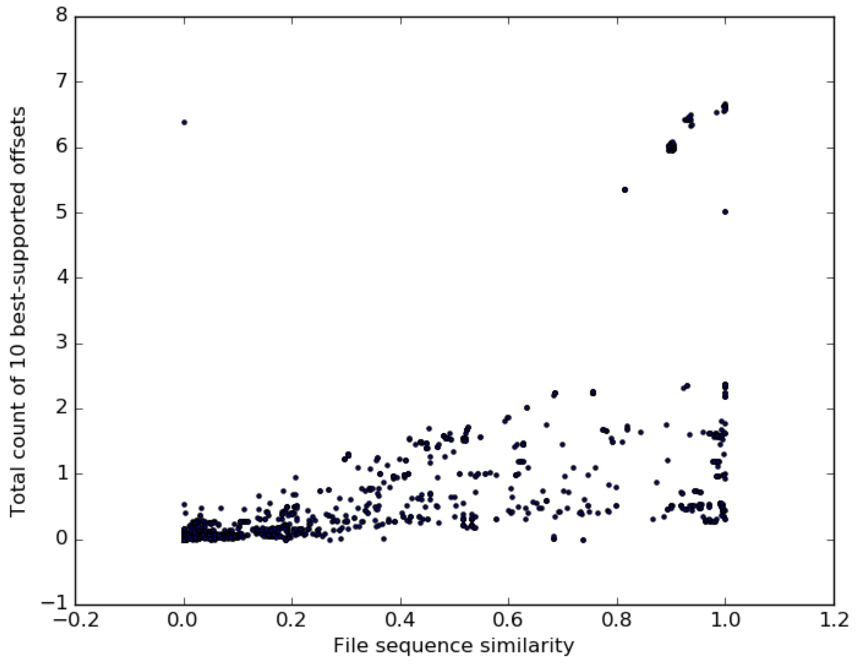


Figure 23: Plot of the sum of the 10 largest offset-evidence counts versus ground truth of similarity between a sample of files.

A quickly computable measure of similarity is to compare histograms of byte values between two files using the classic technique of cosine similarity,

$$\sum_{i=1}^{N} f(i,1) * f(i,2) \Bigg/ \left(\sqrt{\sum_{i=1}^{N} f(i,1)^2} * \sqrt{\sum_{i=1}^{N} f(i,2)^2} \right)$$

We have applied cosine similarity previously to many kinds of file attributes (Rowe, 2018). With only 256 values the cosine similarities are very high (close to their maximum of 1), so we took the 10th power of the values to spread them out better (Figure 24). Nonetheless, there is not much correlation with the sequence similarity between two files, so we did not consider this further for purposes of initial filtering.

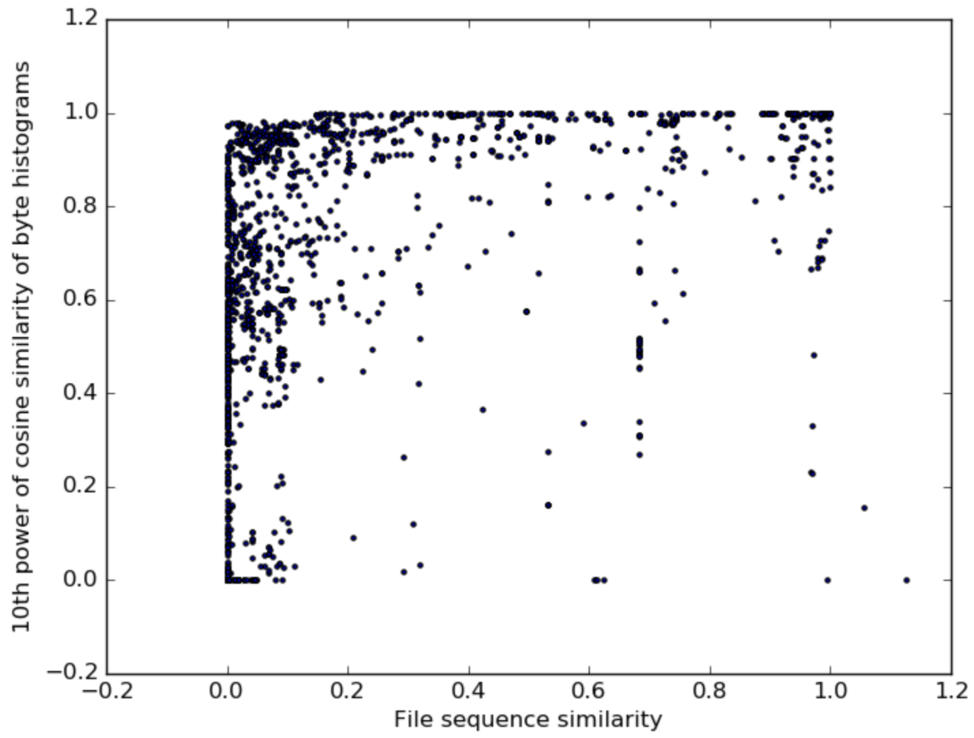


Figure 24: 10th power of cosine similarity of byte distributions versus ground-truth similarity for a sample of files.

Another approach we used was to compare the “texture vectors” on 512- byte segments of the two files, using the best match for each segment of the smaller file where “best” was defined as the weighted distance between the statistics of the two segments. Figure 25 shows the correlation of the five-element vector similarity with the sequence similarity. Although the correlation is not good, there is an upward trend from left to right. Still, it is not as good as the correlation with offsets peaks in Figure 23.

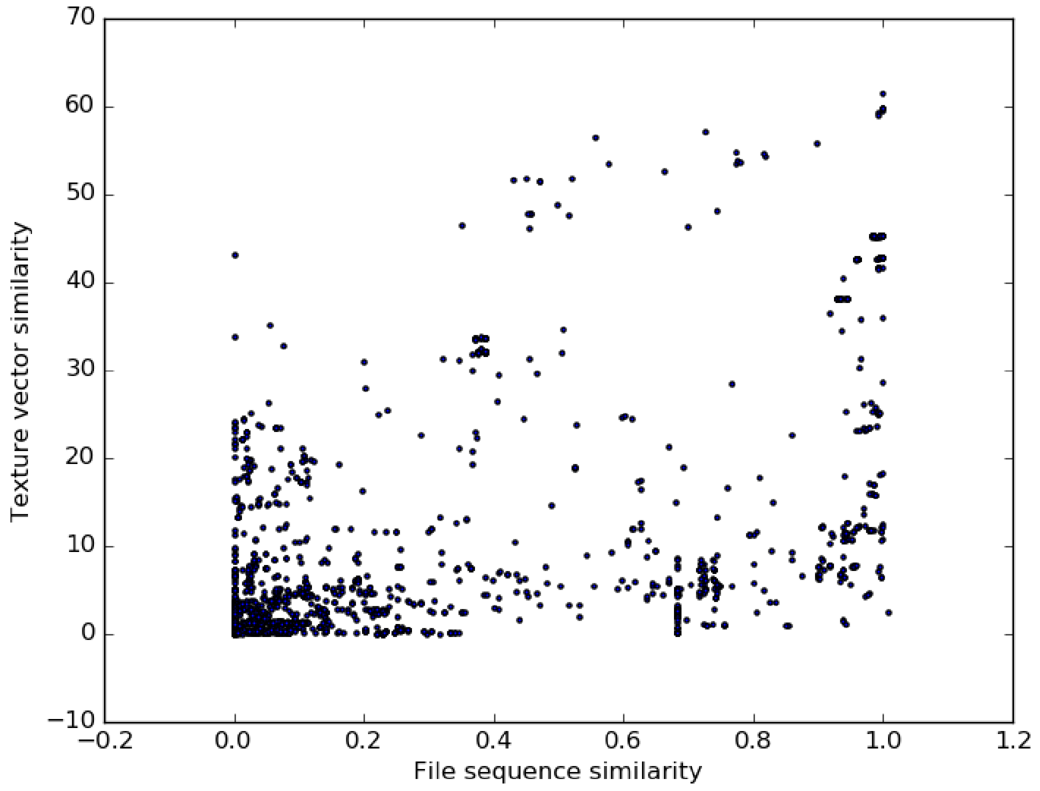


Figure 25: Texture-vector similarity of two files versus ground-truth (sequence) similarity for a sample of files.

We explored whether some of the five components of our texture vector could provide better correlations with our ground truth of file-sequence similarity (Figure 26, Figure 27, Figure 28, and Figure 29). Unfortunately, none of them showed much correlation. We conclude that there is no shortcut to measuring similarities between executables besides comparing large numbers of bytes between the two files.

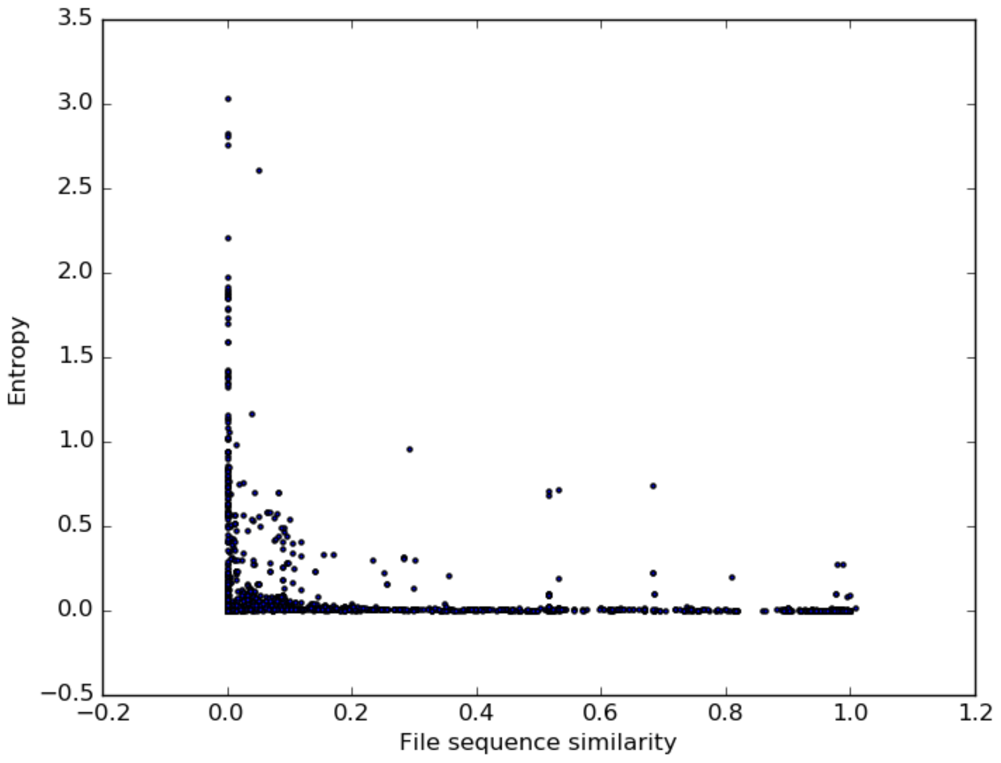


Figure 26: Average similarity of 512-byte entropies versus ground truth.

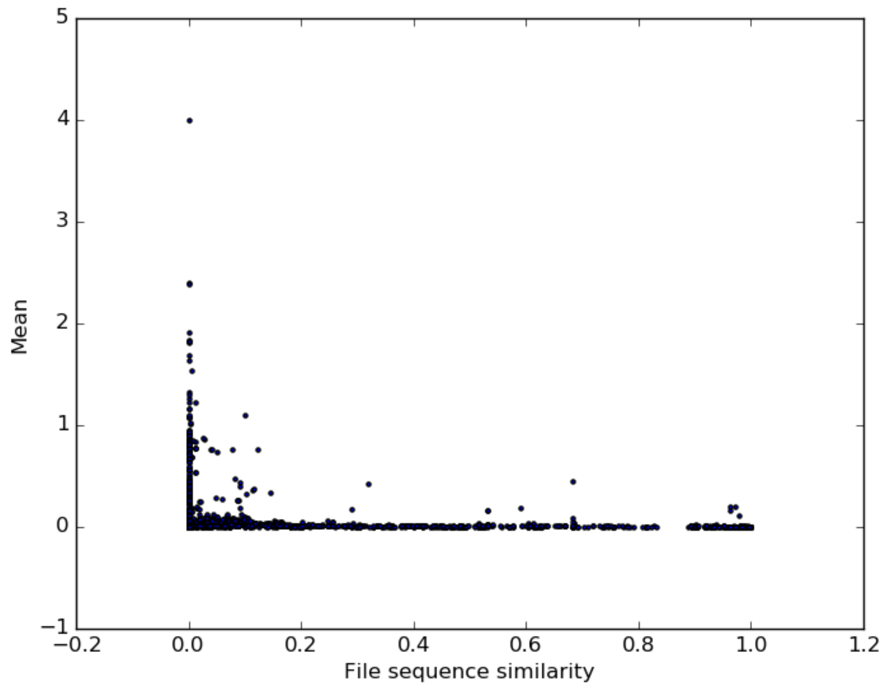


Figure 27: Average similarity of 512-means versus ground truth.

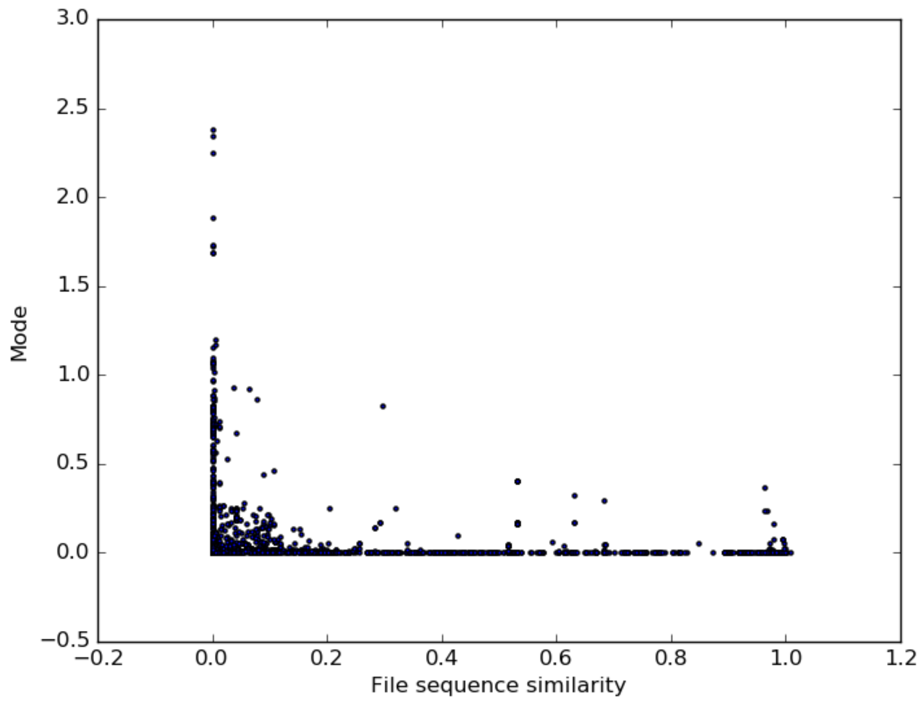


Figure 28: Average similarities of most-common byte value in 512 bytes versus ground truth.

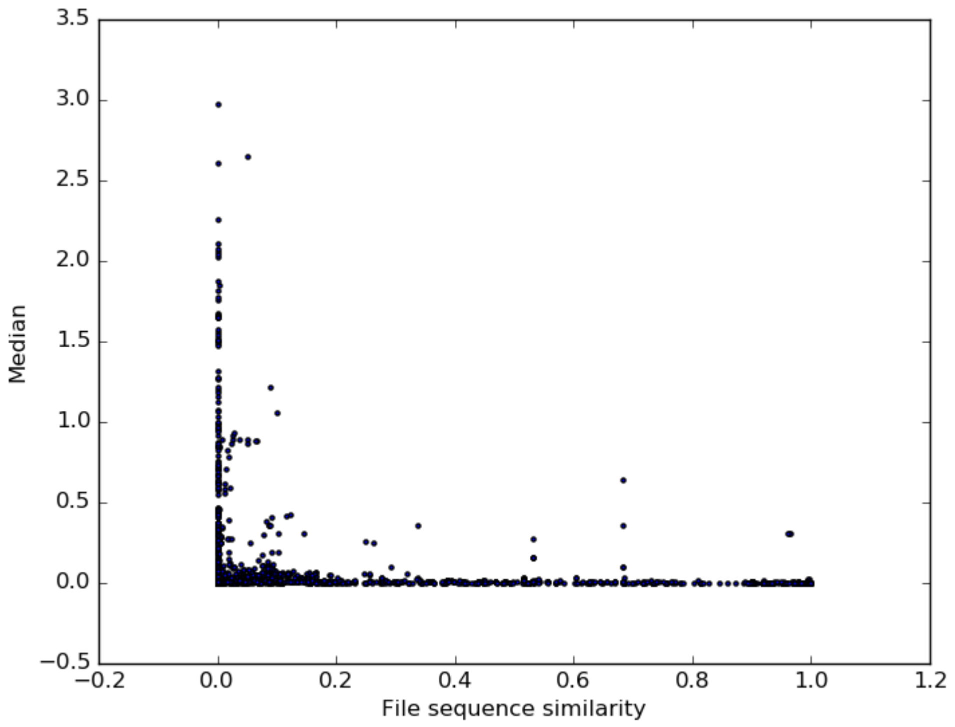


Figure 29: Average median similarity for 512-byte segments versus ground truth.

Results: Suspicious File Versions

The visualizations in section 6 provide a good way to spot anomalous software versions. A fraudulent or malicious version would likely not conform to the patterns of update shown in diagrams like that of Figure 18. It would be difficult for a fraudulent version to get its spacing in time and similarity to other versions to fit in with the normal update framework. So it will often be noticeable as being too early or too late compared the next update since updates tend to be quite regular for most software. So inspecting our diagrams will likely reveal the more obvious kinds of fraudulent activity.

Malware is a special case of fraudulent versions. Some of the executables in our sample were flagged as malware by one of five anti-malware tools used in (Rowe, 2015): Bit9, Symantec, ClamAV, OpenMalware, and VirusShare. We were curious about the similarity of these to non-malicious versions of the software. One approach to creating malware, used by many viruses, is to modify a few lines of code. However, we saw no evidence of this with our data. The malware that did occur in our sample was quite different from the legitimate executables. That suggests that malware authors are becoming increasingly brazen in their counterfeits of executables, and it also suggests they are not using viruses much anymore.

THIS PAGE LEFT INTENTIONALLY BLANK

Recommendations

Our experiments suggest this approach is a good one to analyzing software in the form of executables when source code is not available. We could see clear patterns in the gradual evolution of software versions, and fraudulent or malicious activity clearly stood out. Several kinds of visualizations appear to be helpful in doing this analysis, especially matches of segments of two related executables and graphs of the evolution in time of different versions of executables.

THIS PAGE LEFT INTENTIONALLY BLANK

Conclusions

We developed a toolkit to analyze relationships between different versions of compiled program files, also called executables. It finds similar files by matching sequences of bytes within the files. We applied it to data from real files from used computer equipment purchased around the world. Plotting file similarity of files with the same name over time showed interesting tree-like branching patterns showing updates and new variants created by software vendors. A user interface we created allows a user to focus on individual files and their similarities to other files, including showing exactly which bytes match between two files. We also produced separate visualizations focused only on similarity between the files having the same name and only on the times of the files. We provide supplementary information of the distribution of entropy within the files, which enables distinguishing machine code from data and headers, and the complete list of all matching files including duplicates found in our collection and where. We studied methods to speed up processing.

This analysis can be used to identify anomalous behavior with files such as with malware, fraud, or illegal copies because such files will not fit a normal update pattern on our graphs and will stand out. The results of our study showed that this approach can spot fraudulent software, malware, and illegally copied software. Fraudulent software and malware will exhibit minimal similarities with other software, and illegally copied software will exhibit very high similarities.

THIS PAGE LEFT INTENTIONALLY BLANK

References

B. Allen, Using texture vector analysis to identify file similarity, M.S. thesis, Naval Postgraduate School, December 2019.

T. Arbuckle, “Visually summarising software change,” in 12th International Conference Information Visualization, 2008.

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

T. R. L. Bergroth, H. Hakonen, “A survey of longest common subsequence algorithms,” in Proc. Int’l Symposium on String Processing Information Retrieval (SPIRE ’00), 2000, pp. 39–48.

D. Beyer and A. Hassan, Animated visualization of software history using evolution storyboards. Proc. 13th Working Conference on Reverse Engineering, 2006.

M. Burch, S. Diehl, and P. Weißgerber, “Visual data mining in software archives,” in SoftVis ’05 Proceedings of the 2005 ACM symposium on Software Visualization, 2005, pp. 37–46.

Elsen, S., VisGi: Visualizing Git branches. Proc. IEEE Working Conference on Software Visualization, 2013.

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

J. Gergic, Toward a versioning model for component-based software assembly. Proc. Intl. Conf. on Software Maintenance, 2003.

M. Kim and D. Notkin, Program element matching for multi-version program analyses. Proc. MSR, Shanghai, CN, May 2006, pp. 58-64.

A. Hanjalic, ClonEvol: Visualizing software evolution with code clones. IEEE Working Conference on Software Visualization, Eindhoven, NL, October 2013.

J. Jang and D. Brumley, “Bitshred: Fast, scalable code reuse detection in binary code,” in *CMU-CyLab-10-006*, 2009.

P. Kaur and H. Singh, A model for versioning control mechanism in component-based systems. ACM SIGSOFT Software Engineering Notes, Vol. 36, No. 5, September 2011.

A. Kuhn and M. Stocker, Code timeline: Storytelling with versioning data. Proc. ICSE, Zurich, SW, 2012.

H. Koike and H.-C. Chu, “Vrcs: Integrating version control and module management using interactive three-dimensional graphics,” in Graduate School of Information Systems University of Electro-Communications Chofu, Tokvo 182, Japan, 1997.

L. Merino, M. Ghafari, C. Anslow, and O. Neirstrsz, A systematic literature review of software visualization evaluation. Journal of Systems & Software, Vol. 144, 2018,, pp. 165-180.

K. North, A. Sarma, and M. Cohen, Understanding Git history: A multi-sense view. Proc. SSE, Seattle, WA, US, November 2016

R. Novais, C. Lima, G. Carneiro, P. Junior, and M. Mendonca, An interactive differential and temporal approach to visually analyze software evolution, 2011.

R. Novais, J. Santos, and M. Mendonca, Experimentally assessing the combination of multiple visualization strategies for software evolution analysis. Journal of Systems & Software, Vol. 128, 2017, pp. 56-71.

N. Palix, J. Lawall, and G. Muller, Tracking code patterns over multiple software versions with Herodotos. Proc. AOSD, Rennes, FR, March 2010, pp. 169-180.

J. Rho and C. Wu, “An efficient version model of software diagrams,” in Proceedings 1998 Asia Pacific Software Engineering Conference (Cat. No.98EX240), 2-4 Dec. 1988, Taipei, Taiwan, Taiwan.

N. C. Rowe, Finding contextual clues to malware using a large corpus. ISCC-SFCS Third International Workshop on Security and Forensics in Communications Systems, Larnaca, Cyprus, July 2015.

N. Rowe, Identifying forensically uninteresting files in a large corpus. *EAI Endorsed Transactions on Security and Safety*, Vol. 16, No. 7, article e2, 2016.

N. Rowe, Associating drives based on their artifact and metadata distributions. In 10th International EAI Conference, ICDF2C 2018, New Orleans, LA, USA, September 10–12, 2018, Proceedings.

J. Seemann and J. W. von Gudenberg, “Visualization of differences between versions of object-oriented software,” in Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering, 11-11 March 1998, Florence, Italy, Italy.

R. Shannon, A. Quigley, and P. Nixon, Deep diffs: Visually exploring the history of a document. Proc. AVI 20, Rome, IT, May 2010.

L. Voinea, A. Telea, and J. J. van Wijk, “Cvsscan: Visualization of code evolution,” in SoftVis ’05 Proceedings of the 2005 ACM symposium on software visualization, 2005, pp. 47–56.

A. Zeller and G. Snelting, Unified versioning through feature logic. ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 4, October 1997, pp. 398-441.