Visualizing Differences in Versions of Software Files

Neil C. Rowe[1][0000-0003-2612-0062]

[1] U.S. Naval Postgraduate School, Monterey CA, 93943 USA
ncrowe@nps.edu

Abstract. Understanding differences between versions of the same file is important in digital forensics for analyzing update trends and recognizing anomalous activity. We examined a sample of 20,753 files with 582 distinct filenames having at least five different versions, drawn from a large corpus, with a focus on executable files for which version changes can have major implications. Most current file comparisons focus on block similarities, which are generally unhelpful with executables (unlike source code) for which updates are usually scattered throughout the file. This work first tested an alternative approach that sums weak evidence for offsets of matches between two files, then scanned the files to find matching sequences at the strongest offsets; the fraction of matches to the smaller file measured the similarity of the files. Several ways to make sequence matching more efficient were examined. This work then developed useful visualizations of the differences between two files and the evolution of software versions over time, often showing update patterns of every year or every two years. These methods provide a new tool independent of documentation and path names to identify normal updates, distinguish fraudulent (Trojan) software, and distinguish file near-copies and near-caches. This approach can provide new insights into processes of software updates.

Keywords: Software, Versions, Forensics, Comparison, Executables; Changes, Offsets, Visualization, Malware.

This paper appeared in the 17th EAI International Conference on Digital Forensics & Cyber Crime, 8-10 September, 2026, Reykjavík, Iceland.

1. Introduction

Digital forensics frequently must compare related files, most commonly different versions of the same file. Version comparison has been used to detect malicious code, predict code-maintenance problems [1], and suggest analogous updates [2]. However, it can also provide broader understanding of software development and use.

Versions of executable files are especially valuable to study, but versions of other software-related files can also provide insights. Versions of source code are easier to compare, but they are rarely found on digital systems since they are valuable intellectual property. Changes to executables are harder to find since registers and addresses can change significantly after recompilation for even a minor update. However, instruction

codes (“opcodes”) and constants change less with updates, and they should be matchable. Matching enables seeing how much was added or subtracted from the executable with each version to get an idea of the magnitude of an update. Our research sponsor was particularly interested in recognizing unusual versions with substantial changes from any known version, which are potentially malicious, and versions with very few changes from previous versions, which are possible theft of intellectual property. These are concerns in software acquisition by organizations.

This paper first discusses previous work on comparing and visualizing software versions, then discusses how we assembled a test set from forensic images. We then discuss how we compared versions, which required a different technique from the standard hash-based file-comparison methods since important software-version differences are often spread through a file. Our technique incrementally builds matching sequences and can exploit several pruning methods for poor matches. We then discuss how file matches can be visualized using a novel branching plot, comment on detecting malware, and draw some conclusions.

2. Previous work

Software development is complex, and many projects have tried to aid software engineers with tools to track changes between versions during design. Usually the tools focus on source code, and follow standard algorithms for comparing versions of text like the Linux “diff” command [3]. Changes can be abstracted by generalizations [4]. Most useful approaches focus on the changes to components [5].

Especially useful to software engineering is visualization of relationships in software [6]. Methods have developed visualizations of version clusters [7], tree diagrams of versions [8, 9, 10, 11, 12], diagrams using color and position to encode features [13, 14, 15], and diagrams of versions over time [16]. Work has graphed mappings between software components [17, 18], and parts of software [19, 20, 13], the last of which is closest to our approach. Some methods for comparing text documents also apply to software [21, 22].

Tracking versions of other files besides source code and text has been much less explored. Digital forensics often uses hash values to compare parts of storage media and files. Hashes can be compared on large segments of files, drive sectors, or whole drives to find large-block similarities that show patterns of copying [23]. This approach is unhelpful for versions of software-related files because they usually exhibit many non-contiguous similarities due to recompilation and re-encoding; recompilation changes many registers and addresses after even a minor update (though relative addressing reduces their number), and software-support files like configuration files often also change in scattered locations. Of three classic algorithms [24], block matching still needs significantly long byte sequences; rare-feature matching is only suited for very short features; and comparing blocks to a small set of prototype blocks only works well for data with similar patterns throughout. Matching using fuzzy hashes [24] only works when matching sequences longer than those found in executables, and is primarily useful for searching a drive, not for comparing files [25].

Classic algorithms for subsequence matching [26] are mostly ineffective in comparing software files because the subsequences sought cannot be enumerated in advance. Edit distance could be a more useful concept [27] as it provides a flexible way to recognize fine-grain differences between two strings by computing the changes necessary to map one file to another. While computing it is exponential in complexity in the worst case, dynamic-programming and hierarchical algorithms [28] make it more efficient for strings that have few differences. However, most edit-distance algorithms try to completely explain the differences between two strings, and this level of detail requires considerable time on large files and is unnecessary for most digital-forensic needs. Decompiling executable files can approximate their source code for comparisons, but it is difficult, requires detailed knowledge of a specific machine language, and is inconsistent in its results.

Files can be quickly compared on simple statistics such as entropy and mean values for blocks, but preliminary experiments of ours found this was unhelpful for executables since many files with significantly different contents had similar statistics. A similar problem was observed with overall histograms of short sequences of byte values as compared using the classic technique of cosine similarity, where very little correlation was found between these similarities and the number of sequences matching between the files found using the methods to be described.

3. Test data

We extracted a sample of 582 file families containing a total of 21,666 distinct files from the 2600 images in the full-image portion of the Real Drive Corpus, a collection of drive images obtained by purchase of used equipment around the world over 15 years from 2000 to 2015 [29], supplemented by a few versions of recent executables on our systems. Although the data is old, policies on updates to software have not changed much since then, while becoming more difficult to study with the increasing use of cloud storage for software, so our study could be valuable. Families were defined as all occurrences of a single file name (e.g., "mobsync.dll") ignoring its file path. The mean file size was 138,202 bytes with a standard deviation of 73,079. 290 families were of DLL format (7299 files), 84 were of EXE format (2683 files), and 50 were of the executable formats rpm, ocx, pyc, h, drv, pyo, class, obj, o, 8bf, pyd, cls, and scpt (2028 files). The remainder were a sampling of other software-related files including 54 families of document formats txt, rtf, pdf, doc, xls, ott, ppt, and pot (3471 files), 24 families of configuration-related files plist, ini, acm, tsp, zdct, cfg, properties, scp, and sys (1609 files), 10 families of source formats rb, js, vb, py, inc, and src (258 files), and 70 miscellaneous families with formats cab, zip, chm, hlp, htm, html, css, gif, and png (4318 files).

For each selected file name, all its occurrences in the Real Drive Corpus were identified using metadata. This metadata was calculated during initial forensic analysis. The Sleuth Kit "icat" command was used to extract one file for each distinct hash value from the images in EWF format (Expert Witness Format). Since many corpus files could not be retrieved because of faulty drive images (as many computers and devices

we obtained were sold to the used-hardware market because they failed), retrievals were tried for each hash value until one provided a nonempty result. With so many drives, an undamaged copy for each hash value was usually found for common software families. All files were sanitized by storing modified versions inverting the bits to defang malware while not affecting file comparisons.

Families were excluded from our experiments if their files exceeded 50 megabytes in size, since comparing every pair within such families required too much time for our resources. Files marked for deletion by their operating system were excluded, as previous work showed that their file names were often wrong [30]. Nonetheless, some undeleted files could not be loaded either, as several things can go wrong is storing large amounts of data. Modification times were extracted for these files from metadata since modification time usually is set at the last change by the vendor and is independent of the installation. For this we used the earliest modification time of all the instances in the corpus of the hash value; around 5% of files had more than one modification time recorded for the same file contents, although this is poor software practice if the vendor is responsible.

The mean number of files in the families studied was 35.7 with a standard deviation of 54.7. There were a total of 1.3 million file pairs within families. In computing average file similarities for significance tests, using all the file pairs would unfairly bias the large file families against small file families as well as taking a long time. So tests were run on two samples of pairs, an intra-family sample and an inter-family sample, choosing a random 36 file pairs for each family, or all the pairs if they were less than 36; 36 was chosen because roughly half the families had fewer pairs. An intra-family experiment compared two randomly selected files in a file family, and an inter-family (control) experiment compared a file in a family with a file having the same file extension in a different family. 19,636 file pairs were compared in the intra-family experiment and 21,277 file pairs were compared in the inter-family experiment.

4. Efficient sequence comparison of files

A typical update to a software-related file will insert, delete, or modify a few segments of code or data. Recompiled code of a mostly unmodified executable should have many identical instructions to the original compiled code though the address and register numbers may differ. We can count the fraction of unmatched segments of the smaller file as a metric of dissimilarity of two files.

Figure 1 plots two versions of an executable file soundman.exe where colors indicate sequences of identical code at least 24 bytes long found by our algorithm to be described in section 4, where white indicates unmatchable sequences. The top and earlier file came from a VAIO/Applications/Drivers/Audio/WDM directory on an Austrian machine, and bottom file came in a Program Files/Realtek/Audio/InstallShield directory on an Indian machine. Several insertions were made in the code section in the first half of the file, though much of the code was unchanged. By contrast, a single large red block of the data section towards the end is unchanged except for a few data differences in its second half. This is a typical pattern of changes during routine updates to an

executable. Note that the matched pairs vary considerably in size, so an algorithm that picks a single block size for hash-based block comparison does poorly.

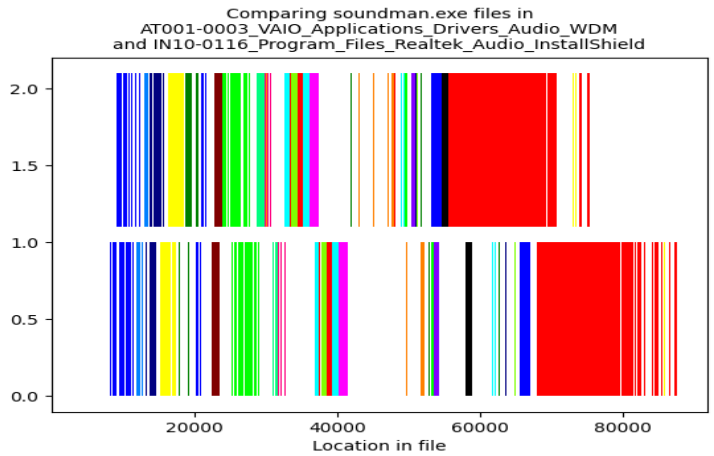

Figure 1: Sequence matches between two versions of an executable file soundman.exe, with colors indicating matched sequences, and the earlier version on top.

4.1 Finding offsets of matching sequences

We defined in this work the degree of similarity between two files as the fraction of the bytes in the smaller file in byte sequences ("spans") that both files contain. An important tuning parameter is the minimum length of a significant span. 24 bytes was chosen after experiments since it can include three 64-bit instructions, and nearly all spurious matches were found in spans less than 24.

To match files like executables where operands may change between versions but many operations will not, sequences should be considered where only one in every 2, 4, or 8 bytes need match; every 4th byte would be appropriate in matching 32-bit machine instructions, and every 8th byte in matching 64-bit instructions. This is called the "granularity" of the match in this work. For instance, the sequence of bytes with values 105, 107, 120, 115, 32, 68, 99, 102 matches a sequence of bytes of 105, 32, 120, 115, 32, 87, 99, 107 with granularities of 2, 4, and 8 but not 1.

Updates to executables found in our corpus almost always increased the size of the executable, as with Figure 1. Hence updates more often insert than delete, and insertions shifted the positions of matches in the earlier version to later in the newer version. If changes between two versions of software were not extensive, we should be able to see consistent evidence of a few particular shifts or "offsets" between their two files. More precisely, an offset can be defined as the integer that must be added to a byte address in the first file to get the corresponding byte address in the second file. So rather than comparing each pair of bytes between files, one can look for the predominant offsets for particular byte values, and find contiguous sequences with those offsets.

Counting evidence for offsets can be done by first indexing the locations of each of the 256 possible byte values for each file. We can then count the number of possible

occurrences of each offset by subtracting the pair of indexes for the same byte value in the two files, and keeping running counts in a hash table. On the average, this approach does 2N retrievals to construct the indexes for files of N bytes, $256 * \left(\frac{N}{256}\right) * \left(\frac{N}{256}\right) = N^2/256$ increments by 1 to the offset counts if evenly distributed in the worst case; then it does $2Nlog_2(2N)$ comparisons to sort the 2N possible offsets to find the K best, for $2N + \frac{N^2}{256} + 2Nlog_2(2N)$ comparisons total. For files of 50,000 bytes, this increases speed by an estimated factor of 210 over exhaustive comparison of bytes.

The offset counts show much noise in this weak evidence, much like the Hough transform in computer vision for finding broken lines. Byte values are not equally likely in real forensic data; Figure 2 shows that the histogram of raw byte values in our full corpus, with big peaks for both 0 and 255. So it is important to exclude matches involving all-zero bytes and all-one bytes in sequence matching since they are common and create many spurious offsets. The figure shows that the remaining byte values vary significantly in occurrence rate, so their counts were weighted in rating possible offsets by $1/(n_{i1} + n_{i2})$ where n_{ij} is the count of byte value i found in file j, to provide fairer comparisons between sequences.

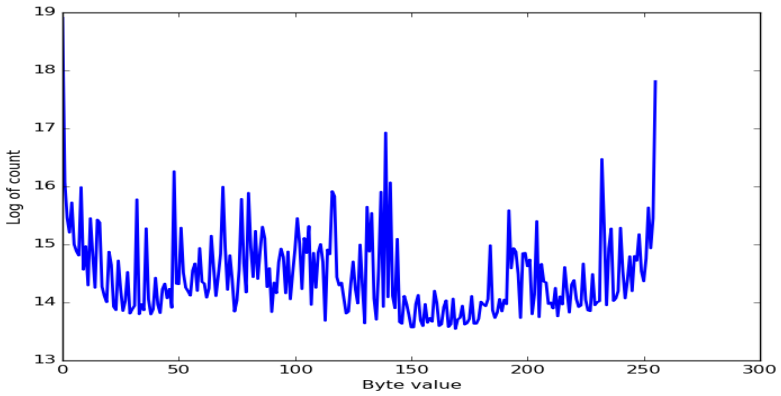

Figure 2: Log histogram of integer byte values on a random sample of the corpus using raw bytes.

4.2 Inference of spans of matching bytes

Changes to an executable during an update usually cluster. Offsets with high weighted counts between two files must next be confirmed to predominantly occur in contiguous spans before they can be considered evidence of insertions or deletions. Comparison can be done by scanning the two files simultaneously at a particular offset value, counting consecutive matches. Now the occasional 0 and 255 byte values can be considered because we have eliminated most of their spurious matches. However, we found it desirable to require that spans contain at least one non-zero byte and have no zero-byte sequences of longer than half the minimum span length.

Because spans need not match every byte, spans found can conflict with one another about their offset recommendations for the same bytes. Three principles were used to exclude less-likely spans:

1. A longer span that overlaps a shorter span has priority in the area of their overlap, since the longer spans have more evidence. Length of a span was defined as the number of bytes it covered divided by its granularity, so it measured the number of actual bytes matched.
2. Spans than match with a smaller granularity (of 1, 2, 4, and 8) have priority over spans which map with a larger granularity in the area of their overlap, since their matches are more dense.
3. Shortened spans created by the first two principles that are less than the minimum span length (24 bytes) are eliminated.

For example in comparing two files, if one span with offset 100 covers locations 150 to 350 of a file on every byte (a granularity of 1), and another span with offset -50 covers locations 140 to 500 of a file on every second byte (a granularity of 2), then 150 to 350 of the second span should be eliminated as well as its too-short left side from 140 to 150. We get two spans with offset 100: 150 to 350 with a granularity of 1, and 350 to 500 with a granularity of 2.

It is important to find overlaps between spans in both directions of comparison. So for instance, for two spans with a granularity of 1 in a first file, one with an offset of 100 from 500 to 650 and one with an offset of -100 from 700 to 800, the two spans will overlap from 600 to 700 in the second file, and the shorter second span can be eliminated.

Table 1 shows statistics on the intra-family sample mentioned in section 3. Matches at granularities greater than 1 were uncommon, though they were important for some files and should not be ignored. The median number of spans found in an executable comparison was 68, and only 15.7% of the spans 24 bytes or longer found in all files were over 128 bytes. Nonetheless, the median length of the longest span for executables was 2373, suggesting that some longer sequences matched well despite considerable fragmentation in the matches. The last two rows were computed by finding the distance, for each span, of the nearest span in a comparison of two files that had at least two spans of at least 24 bytes; this method then averages the nearest distances over all spans in the comparison, and computes the median over all file comparisons. These numbers indicate significant gaps between matched parts of different versions of the same file, and suggest that block matching will likely do poorly with software files.

4.3 Other criteria for avoiding file and sequence comparisons

The methods of the last section still find many spurious matches between two files because of frequently reused programming constructs; this makes it hard to visualize the overall pattern of matching. The criterion that 5% of the smaller file's sequence were shared with the larger file was chosen as the minimum reasonable similarity of two files after experiments examining a sample of file pairs. In addition, many possible file pairs of a family collected over a long time are unlikely to be similar based on a large size disparity, and this eliminates the need to compare their bytes. Figure 3 plots

the average similarity versus the log of the ratio of the larger file size to the smaller file size, and this justifies using a 5% size difference as a threshold for exclusion.

Table 1: Statistics on the file comparisons in the intra-family sample.

Statistic	**Value**
Mean file size in bytes	136,838
Mean number of bytes in spans of granularity 1	26,137
Mean number of bytes in spans of granularity 2	1,189
Mean number of bytes in spans of granularity 4	558
Mean number of bytes in spans of granularity 8	141
Mean maximum offset rating for file pair	19.89
Mean offset rating for file pair before filtering	0.108
Mean number of spans found per file-pair comparison	140.5
Mean fraction of bytes that match with spans of any granularity	0.2615
Mean 8-bit entropy	3.705
Number of executable pairs with at least one matching span	10,048
Number of non-executable pairs with at least one matching span	3257
Number of spans of 128 bytes or less in all files	25,954,000
Number of spans of more than 128 bytes in all files	483,000
Median number of spans in executable pairs, excluding pairs with 0	68
Median number of spans in non-executable pairs with at least one span	13
Median length in bytes of the longest span over all executable pairs	2,373
Median length in bytes of the longest span over all non-executable pairs	351
Median of mean nearest gap for each span over all executable pairs	50.5
Median of mean nearest gap for each span over all non-executable pairs	63.6

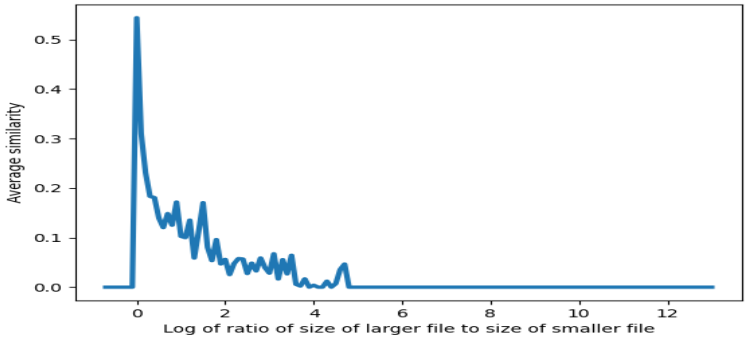

Figure 3: Mean similarity versus ratio of compared file sizes for the intra-family sample.

It is also desirable to avoid comparing files of high entropy since they are likely encoded or encrypted, and would lack clear insertions or deletions. Figure 4 shows the average similarity in the intra-family data as a function of the maximum of the 8-bit entropies of the two files; similar but less clear results were obtained from the minimum and average of the entropies. Very-high and very-low entropies fall below the 0.05 average similarity threshold, and we found maximum entropies < 0.35 and > 5.25 were almost certain indicators of insufficient similarity.

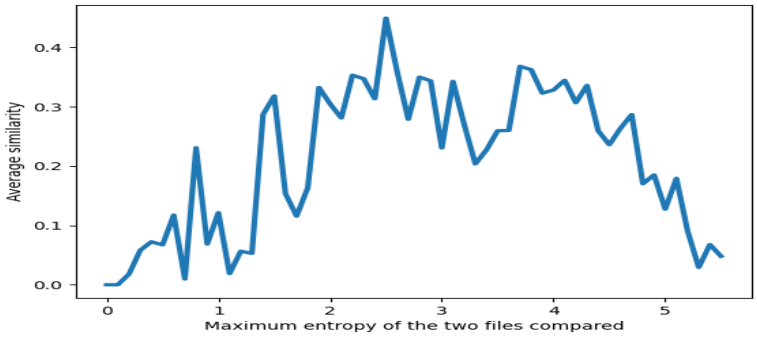

Figure 4: Mean similarity versus the maximum of the entropies of the files compared in the intra-familiy sample.

Another possible pruning criterion for file matching is when file modification times are far apart. When we compared epoch times (seconds since January 1, 1970) for every pair in every family in our corpus, only 14.1% of 18,130 closest-matching file pairs involved the most immediately previous files in terms of modification times. This means that file pairs in a family that are adjacent in time often have different operating systems and hardware, and have very different contents. As Figure 5 shows for executables alone, similarities occur for files in a broad range from 0 to 15 on the graph (corresponding to a maximum of 37 years apart), and only then do similarities decrease significantly. This suggests that major updates occur no more than yearly (17.2 on the horizontal axis), but some software takes many years to get updated. Since file pairs in our corpus were rarely more than 15 years apart, modification timestamps did not provide useful pruning data.

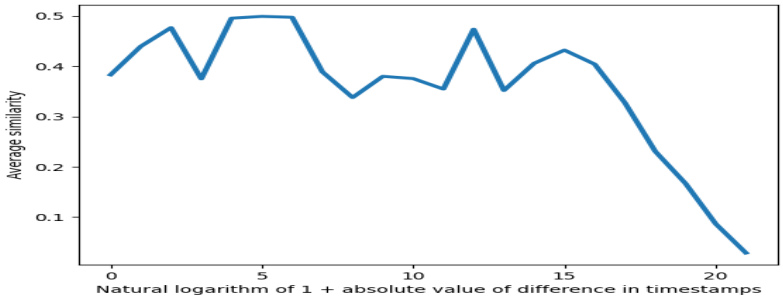

Figure 5: Mean similarity of executable files in the intra-family sample versus the logarithm of one plus the absolute value of the difference of their timestamps.

File-comparison speed can be further improved by excluding unlikely offsets below a rating. For rating, we used the similarity fractions (fraction of the smaller drive matching the larger drive). Figure 6 shows the distribution of the average logarithm of the ratio of intrafamily-similarity to the average interfamily-similarity for each family in the corpus; the horizontal axis is the logarithm of the intrafamily similarity. The instability on both left and right is due to small counts and is not statistically significant. As expected, high similarity values tend to occur within families, and low similarity values occur between families. Some intrafamily comparisons had low similarity values for major updates, and some interfamily comparisons had high similarity values for reused code. Nonetheless, a clear phase change occurs around -1 in the plot, and it is reasonable to say that no significant difference occurs in counts between the two experiments for offsets where the count ratio logarithm is greater than -1, which happens at -0.66 on the horizontal axis, corresponding to an offset rating greater than 0.52. This was used for pruning in subsequent experiments.

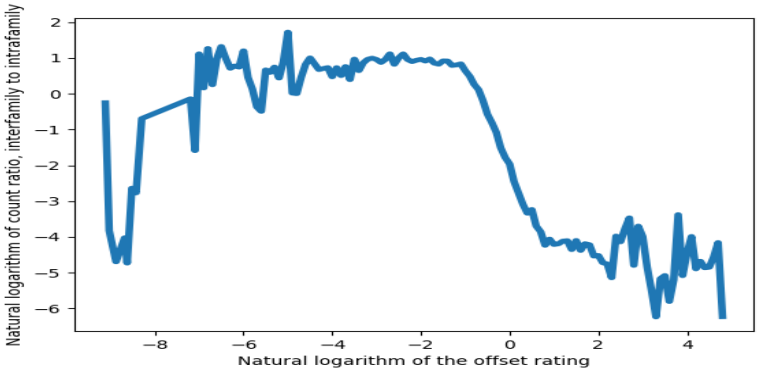

Figure 6: Count ratio of spans bytes found versus offset rating for interfamily and intrafamily offset ratings.

We also tested other pruning criteria for matches:

1. Exclude all but the K highest-ranking offsets for some value of K in comparing two files. This was excluded because it made too many errors, suggesting that small changes in updates are common.
2. Compute the mean offset weight for all the offsets found in comparing two unrelated (different-family) files, and only further consider offsets whose weight was more than three standard deviations greater than this mean. This was even worse that criterion 1.
3. Exclude parts of files unlikely to match. We found that code sections of an executable had a 5.9 byte entropy, text sections 3.2, and encoded-data sections 7.9. Executable formats like Microsoft PE [31] segregate executable files into code and data which could be exploited to rule out sequence matches. But this did not provide much pruning power in experiments.

5. File family visualization

The main goal of this work was to provide good visualization of the evolution of file versions (families). This can help learn the similarity patterns between versions in normal software schedules, suggest software-development anomalies, and find close copies of software, all without requiring hashes or signatures. It can also identify software components that could be fraudulent or a theft of intellectual property. The metric of dissimilarity of file sequences between two file versions introduced in section 4 is analogous to a distance in a plot of the space of file versions.

Related files can be visualized as connections in a graph. Earlier in this work, the Gephi software (https://gephi.org) for visualizing graphs was used, but we found it is mainly useful for graphs with thousands of nodes, more than in any of the families tested. It is also a non-metric visualization where edge lengths do not measure degree of similarity. Metric visualizations can better indicate important software updates, so we implemented one. In these plots, nodes represent files, the horizontal axis represents the modification time, and the vertical distances represent cumulative degree of dissimilarity for each file to its most-similar earlier file, plus 0.15 to prevent nodes from crowding vertically. Green dotted lines connect a file to its most-similar predecessor, except that files whose similarities to all other files in their family were less than 0.05 were assigned a vertical coordinate of 0.

Figure 7 shows the example visualization of the 45 members of the family of dunzip32.dll, an unzipping executable provided by Microsoft and used by a variety of software. One major line of development is clear, with a few outshoots, and six outliers are at the bottom. Updates were frequent at first, and then slowed, a typical pattern for the lifetime of software. Connecting lines become flatter (as 31-40, 31-43, and 31-44) as changes between versions became less extensive over time. Near-vertical alignments of nodes such as 2-3-4 and 16-18-19 suggest versions released at nearly the same time for different operating systems or hardware.

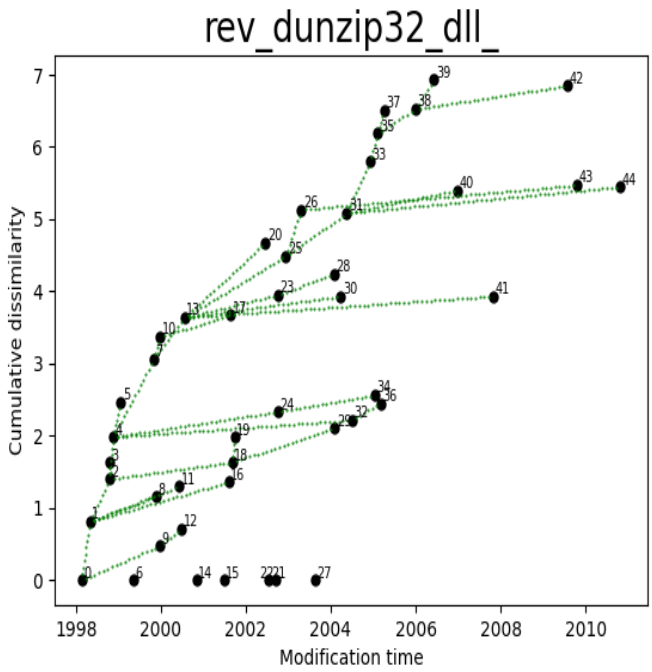

Figure 7: Automatically generated graph visualization of the 45 versions of of dunzip32.dll in the corpus.

Numbers in Figure 7 represent the 45 distinct hash values of the 348 files with file-name “dunzip32.dll; the numbers are assigned in order of last modification time. The most common version was 44 at the furthest right in the figure, which occurred 142 times under Program Files/Real/RealPlayer in some of the last-collected drives. Other popular versions were 3 (fourth up on the left side) which occurred 30 times exclusively on Mexican drives under WINNT/system32, version 13 (eighth up on the left side) which occurred 7 times on a variety of drives under Corel Draw, version 29 (in 2004 at height 2) which occurred 25 times under Program Files/Real/RealPlayer, and version 33 (2006 at a height of 6) which occurred 12 times under Program Files/Auto CAD. Several versions (0, 2, 16, 32, 38, 42) related to anti-malware programs. It appears that many vendors use older versions of this executable for compatibility with the rest of their software; such executables are excellent sources for historical research even though drive imaging often gets only the latest version for each drive.

Figure 7 shows that the most-similar previous version of an executable is usually close in time, but not necessarily the closest. This was true for most of our file families, and is good support for our similarity algorithm. Earlier work of ours visualized simple statistics such as mean and standard deviation on blocks of files to indicate visual similarities, and made good pictures, but they correlated little with file matches.

Figure 8 shows a similar diagram for googlelicense.rtf, the Google license file, which has quite a few variants. It appears to have had simultaneous updates to variants in 2008, 2010, 2011 (twice), and 2012. This is typical of files intended for many systems.

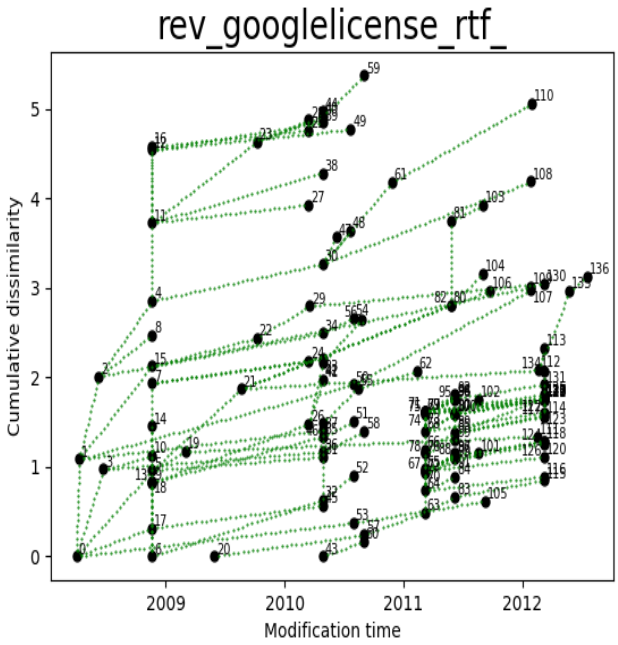

Figure 8: Graph visualization for files of google_license.rtf.

Figure 9 shows the histogram of the natural logarithm of modification-time gaps between the most-similar but different files found in each family. The horizontal axis is the logarithm of 0.0001 plus the time gap in years. A big peak occurs for minimal differences in the modification time for variants released simultaneously for different operating systems, a peak at two weeks (-3 on the horizontal axis) possibly for critical updates, and peaks at one (0.1) and two years (1.2) for routine updates.

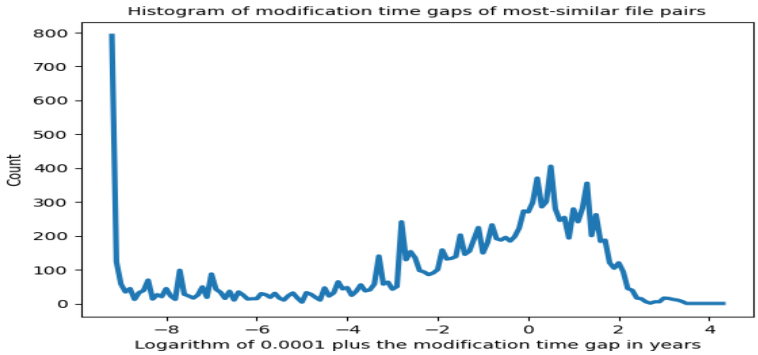

Figure 9: Distribution of time gaps between most-similar file versions in the corpus.

7. Identifying potential malware and bug fixes

Some files in the corpus were tagged as malware by one of five anti-malware tools in 2015: Bit9 (https://carbonblack.com), Open Malware (https://dannyquist.github.io/about/), VirusShare (https://virusshare.com), Symantec (https://broadcom.com), and Clam AV (https://www.clamav.net) [33]. We were curious if this malware had different time patterns than normal software. The corpus was purchased around the world and many drive owners were not diligent about cybersecurity, so there were around 350,000 instances (including duplicates) of malware in the full corpus from which the families were derived.

The visualizations in section 5 did provide a way to spot some kinds of anomalous software versions by marking nodes in red those that were tagged as malware by our tools. Figure 10 shows an example where malicious versions of webclnt.dll (a Web interaction tool) do form a separate vertical sequence of related variants from the legitimate variants in 2001-2005. We also discovered 7969 file pairs with identical hash values for the Counter Strike game where one member of the pair had "_xtreme_edition_" in its path and the other did not, suggesting illegal copies of a game. However, many software versions identified as malware by the five tools did not appear in any obvious patterns; as we noted in our previous work with this malware data, these five tools rarely identified the same files as malware, so many of those malware identifications may be spurious.

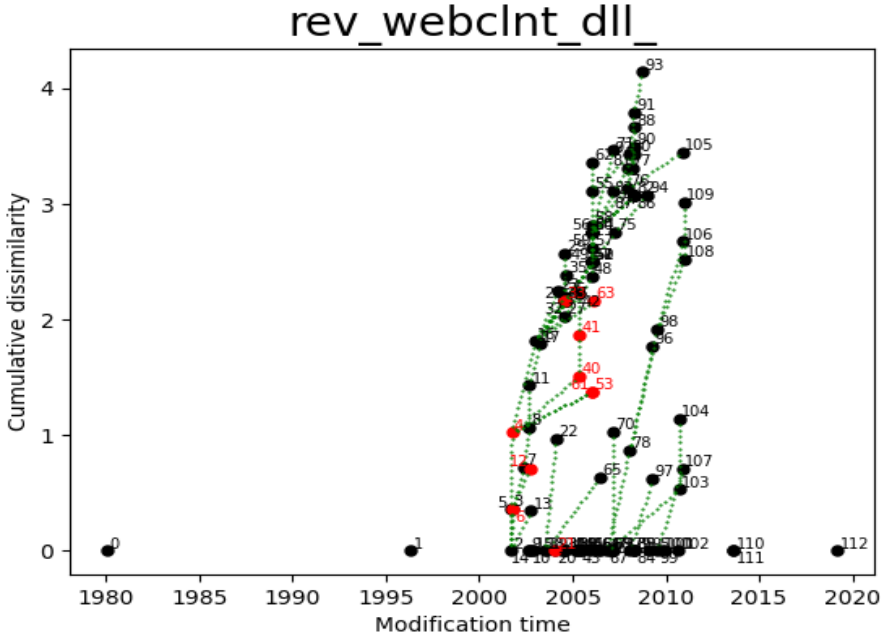

Figure 10: Graph of versions of webclnt.dll showing malicious versions in red.

Current work is focusing on distinguishing the two major categories of updates, bug fixes and feature additions [34]. They are rarely distinguished in their file names, but can be distinguished by dissimilarity measure to the most recent version (bug fixes are

more similar) and time gap (bug fixes have shorter and more atypical gaps consistent with the left side of Figure 10). We tested binary classification of updates based on applying a threshold to a weighted sum of those features, and got promising results.

8. Conclusions

This work developed a powerful tool for understanding patterns in the normal version history of software files. The software industry makes updates periodically, but until now we have not had a good tool to see what they have done and to distinguish normal from abnormal updates, nor detect distinctive characteristics of a particular vendor's update policy. A new algorithm was developed in this work for finding matching sequences between software files, a method that is made efficient by exploiting several kinds of pruning of spurious matches as shown by testing on a representative corpus. Tests on real executables also showed they could be usefully compared even with changes to addresses, registers, and constants between versions because enough clues are left to match the remaining bytes. This analysis yields dissimilarity measures which can be usefully visualized to see tree-like structures of updates. They also enable learning the directories used in normal patterns of updates, the relative frequency of insertions, deletions, and changes, and recognizing abnormal behaviors of several kinds that suggest bottlenecks, fraud, or malware. As software is increasingly distributed and assembled as needed, this study of already assembled software files will be increasingly difficult to duplicate, and its results will thus become increasingly valuable.

Acknowledgements

This work was supported by the Acquisition Research Program at the Naval Postgraduate School. Bruce Allen and Adam Montgomery helped with details. The data studied here can be obtained from the author.

References

1. Farago, C., Hegedus, P., Ladanyi, G., Ferenc, R.: Impact of version history metrics on maintainability. In: Proc. 8th Intl. Conf. on Advanced Software Engineering and Its Applications, pp. 30-35 (2015).
2. Zimmermann, T., Weissgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: Proc. 26th Intl. Conf. on Software Engineering (2004).
3. MacKenzie, D., Eggert, P., and Stallman, R.: Comparing and merging files with Gnu diff and patch. Network Theory Ltd., Clifton, UK (2003).
4. Zeller, A., Snelting, G.: Unified versioning through feature logic. ACM Transactions on Software Engineering and Methodology, 6 (4), 398-441 (October 1997).
5. Gergic, J.: Towards a versioning model for component-based software assembly. In: Proc. Intl. Conf. on Software Maintenance (2003).
6. Merino, L., Ghafari, M., Anslow, C., Neirstrsz, O.: A systematic literature review of software visualization evaluation. Journal of Systems & Software, 144, 165-180 (2018).
7. Beyer, D., Hassan, A.: Animated visualization of software history using evolution storyboards. In: Proc. 13th Working Conference on Reverse Engineering (2006).

8. Arbuckle, T.: Visually summarising software change. In: 12th International Conference Information Visualization (2008).
9. Seemann, J., von Gudenberg, J.: Visualization of differences between versions of object-oriented software. In: Proc. Euromicro Conference on Software Maintenance and Reengineering, Florence, Italy (March 1998).
10. Novais, R., Lima, C., Carneiro, G., Junior, P., Mendonca, M.: An interactive differential and temporal approach to visually analyze software evolution (2011).
11. Elsen, S.: VisGi: Visualizing Git branches. In: Proc. IEEE Working Conference on Software Visualization (2013).
12. Novais, R., Santos, J., Mendonca, M: Experimentally assessing the combination of multiple visualization strategies for software evolution analysis. Journal of Systems & Software, 128, 56-71 (2017).
13. Voinea, L., Telea, J. van Wijk, A.: Cvsscan: Visualization of code evolution. In: Proc. 2005 ACM Symposium on Software Visualization, pp. 47-56 (2005).
14. Kuhn, A., Stocker, M.: Code timeline: Storytelling with versioning data. Proc. ICSE, Zurich, SW (2012).
15. Hanjalic, A.: ClonEvol: Visualizing software evolution with code clones. Proc. IEEE Working Conference on Software Visualization, Eindhoven, NL (October 2013).
16. Aghajani, E., Mocci, A., Bavota, G., Lanza, M.: The code time machine. In: Proc. IEEE 25th International Conference on Program Comprehension (ICPC) (2017).
17. Kim, M., Notkin, D.: Program element matching for multi-version program analyses. Proc. MSR, Shanghai, CN, pp. 58-64 (May 2006).
18. Kaur, P., and Singh, H.: A model for versioning control mechanism in component-based systems. ACM SIGSOFT Software Engineering Notes, 36 (5) (September 2011).
19. Burch, M., Diehl, S., Weißgerber, P.: Visual data mining in software archives. In: Proc. ACM Symposium on Software Visualization, pp. 37–46 (2005).
20. North, K., Sarma, A., Cohen, M.: Understanding Git history: A multi-sense view. In: Proc. SSE, Seattle, WA, US (November 2016).
21. See, A., Monnich, M., Fischer, M: Enhancing binary code similarity analysis for software updates: A contextual diffing framework. In: Proc. 20th ACM Asia Conference on Computer and Communications Security, pp. 1724-1740 (2025)
22. Shannon, R., Quigley, A., Nixon, P.: Deep diffs: Visually exploring the history of a document. Proc. AVI 20, Rome, IT (May 2010).
23. Young, J., Foster, K., Garfinkel, S., and Fairbanks, K.: Distinct sector hashes for target file detection. Computer, 45 (12), pp. 28-35 (December 2012).
24. Lee, A., Atkison, T.: A comparison of fuzzy hashes: Evaluation, guidelines, and future suggestions. In: Proc. ACM Southeast Conference, Kennesaw, GA, US, pp. 18-25 (April 2017).
25. Drucker, S., Petschnigg, G., Agrawala, M.: Comparing and managing multiple versions of slide presentations. In: Proc. UIST, Montreux SU, pp. 47-56 (2006).
26. Bergroth, T., Hakonen, H.: A survey of longest common subsequence algorithms. In: Proc. Intl. Symposium on String Processing Information Retrieval, pp. 39–48 (2000).
27. Cormode, G., Muthukrishnan, S.: The string edit distance matching problem with moves. ACM Transactions on Algorithms, 3(1), Article 2 (February 2007).
28. Fluri, B., Wursch, M., Pinzger, M., Gall, H.: Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Transactions on Software Engineering, 33(11) (November 2007).
29. Garfinkel, S., Farrell, P., Roussev, V., and Dinolt, G.: Bringing science to digital forensics with standardized forensic corpora. Digital Investigation, 6, pp. S2-S11 (August 2009).

30. Rowe, N.: Identifying forensically uninteresting files in a large corpus. EAI Endorsed Transactions on Security and Safety, 16(7), article e2 (2016).
31. Microsoft: PE Format. Accessed at https://docs.microsoft.com/en-us/windows/win32/debug/pe-format (August 11, 2020).
32. Rowe, N.: Efficiently comparing versions of software files. Technical Report, NPS, https://nps.edu/web/research/calhoun/, in preparation.
33. Rowe, N.: Finding contextual clues to malware using a large corpus. In: ISCC-SFCS Third International Workshop on Security and Forensics in Communications Systems, Larnaca, Cyprus (July 2015).
34. Mockus, A., Votta, L., Identifying reasons for software changes using historic databases. In: Proc. Int. Conf. Software Maintenance (ICSM), pp. 120-130, doi: 10.1109/ICSM.2000.883028 (2000).