Chapter 9

CLASS-AWARE SIMILARITY HASHING FOR DATA CLASSIFICATION

Vassil Roussev, Golden Richard III and Lodovico Marziale

Abstract This paper introduces "class-aware similarity hashes" or "classprints," which are an outgrowth of recent work on similarity hashing. The approach builds on the notion of context-based hashing to create a framework for identifying data types based on content and for building characteristic similarity hashes for individual data items that can be used for correlation. The principal benefits are that data classification can be fully automated and that *a priori* knowledge of the underlying data is not necessary beyond the availability of a suitable training set.

Keywords: Similarity hashing, class-aware similarity hashing, classprints

1. Introduction

The problem of identifying the type of data inside a container (e.g., file or disk image) has been studied for several years with few positive results. Indeed, the ability to identify the underlying type of the data without the help of file system metadata is very useful in data recovery operations (file carving), especially as a means for validating the attempted data recovery. For example, if a tool runs into text data while attempting to carve a JPEG file, it is clear that the process is not on the right track. This is important because data carving is routinely applied to target images to recover (fragments of) deleted data and is often a valuable source of information.

Another related problem is automated data correlation. Targets often contain several terabytes of data, making it necessary to quickly separate potentially relevant data from irrelevant data. The best strategy is to use prior accumulated data to make the separation. Traditional forensic investigations use large, sophisticated databases (e.g., for fingerprints and DNA) to quickly zero in on relevant data. In digital forensics,

success has come from using databases of hash values of known system and application files, such as those maintained by NIST [6]. But it is debatable if this approach will work when the databases contain billions of hash values – would it be necessary to compute clusters just to perform hash searches?

Traditional, file-based (cryptographic) hashing is useful but fragile; it needs the exact binary representation of all versions of the objects of interest. Several schemes have been proposed to address this issue. Kornblum [5] has proposed a context-based approach that dynamically splits a file into individually hashable chunks from which a composite hash is produced. While the use of a hash-based context – which can be traced to early research in information retrieval [1, 3] and is derived from Rabin's original work [8] – is a proven technique, the rest of the scheme lacks robustness. Recently, we proposed a more robust approach [9] based on Bloom filters [2], but it lacks an elegant mechanism for splitting up arbitrary targets.

In [10] we refined the approach to create the multi-resolution similarity (MRS) hashing scheme that can be applied to arbitrary targets. The scheme clearly identifies similarities in data files that would be classified by a human as being related (e.g., different drafts of the same document). Also, the MRS hashing scheme can identify the presence of a contained file (e.g., JPEG) inside a larger target (e.g., raw drive image) without metadata or any other assistance from the file system.

An MRS hashing tool has significant performance advantages stemming from the fact that it requires only a single sequential pass over an image. In contrast, other file-based tools require access to file metadata, which results in non-sequential disk access patterns.

Figure 1 illustrates the effects of non-sequential access on the throughput of a modern hard drive, as measured by Intel's IOMeter tool. As little as 2% randomness in the workload can produce a 30% performance penalty; 5% randomness can cut performance in half. With hard drive capacities outpacing bandwidth and latency improvements [7], forensic targets are increasing in size faster than the ability of forensic tools to process them in a timely manner.

This paper discusses the use of class-aware similarity hashing to address these issues. Empirical results using a custom tool show that class-defining features can be automatically extracted for several classes of commonly-used file types. In other words, it is practical to define common file types solely based on syntactic features of their binary representations.

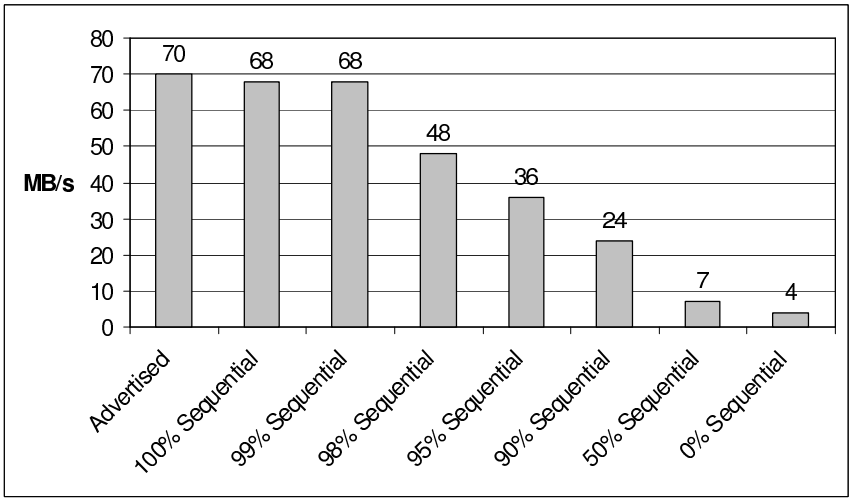

Figure 1. Hard drive throughput for WDC WD5000KS (500 GB).

2. Similarity Hashing

This section briefly summarizes recent work on similarity hashing. Interested readers are referred to [10] for additional details.

Block-level hashing is the most basic scheme for determining the similarity of binary data. The technique generates and stores cryptographic hashes for blocks of a chosen fixed size (e.g., 512 bytes). Block hashes from two different sources can then be compared and, by counting the number of blocks in common, a measure of similarity may be determined. The principal advantages of this scheme are that it is supported by existing hashing tools and that it is computationally efficient; in fact, the hash computations are faster than disk I/O.

However, block-level hashing has certain limitations when applied to discover file similarity. The success of the technique depends heavily on the physical layout of the files being similar. However, the insertion, deletion or modification of just one character at the beginning of a file could render all the block hashes different. Also, block hashes do not help identify if an object (e.g., JPEG image) is embedded in a file (e.g., Microsoft Word document). In short, block hashing is too fragile and negative results do not reveal any useful information.

Kornblum [5] proposed context-triggered piecewise hashing to address the limitations of block-level hashing. The idea is to identify content markers, called "contexts," within (binary data) objects and to store the sequence of hashes for each of the pieces (or chunks) in between

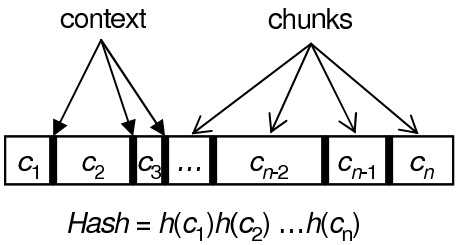

Figure 2. Context-based hashing or "shingling."

contexts (Figure 2). In other words, the boundaries of the chunk hashes are not determined by an arbitrary fixed block size but are based on object content. The hash of the object is simply a concatenation of the individual chunk hashes. Thus, if a new version of the object is created by localized insertions and deletions, some of the original chunk hashes will be modified, reordered or deleted, but enough will remain in the new composite hash to identify the similarity.

To identify a context, Kornblum's `ssdeep` implementation uses a rolling hash over a window of c bytes that slides over the target. If the t lowest bits of the hash (the trigger) are all equal to one, a context is detected, the hash computation of the preceding chunk is completed and a new chunk hash is started. The value of t depends on the size of the target because `ssdeep` generates a fixed-size result. Intuitively, a larger t value produces less frequent context matches and reduces the granularity of the hash.

We recently proposed Bloom filter similarity hashing [9], a scheme utilizing Bloom filters to derive object similarity. This scheme uses the (known) structure of an object to break it into components, which are individually hashed and placed in a Bloom filter. Using the mathematical properties of filters, we demonstrated analytically and empirically that the bitwise comparison of filters yields a useful measure of the similarity between the binary representations of two or more objects.

In subsequent work [10], we combined Bloom filter similarity hashing with context-based object decomposition ("shingling" [3]) to handle arbitrary binary data. We also devised a standardized multi-resolution scheme called MRS hashing that allows objects of arbitrary size to be hashed without loss of resolution. Moreover, the scheme allows different-sized objects to be compared; for example, it is possible to search for the remnants of a 1 MB file inside a 100 GB target.

MRS hashing is very memory efficient due to the use of Bloom filters; hash values are no more than 0.5% of target size. Thus, the complete MRS hash of a 500 GB hard drive can fit in the main memory of a mod-

ern workstation. From the point of view of performance, MRS hashing is no more expensive than block-level MD5 hashing, even when the unoptimized version of MD5 is used. The comparison step is very efficient and can be sped up by using lower resolution for large targets and/or delegating comparisons to a graphics processor (e.g., NVidia G80); this can speed up the process twenty times.

3. Class-Aware Similarity Hashing

As discussed in the preceding section, MRS hashes provide a sensitive and tunable means for finding similarities among binary data objects. But why are these objects similar? From our previous work, it appears that MRS hashing works reasonably well for user-generated artifacts (e.g., .jpg, .doc and .pdf files) in that the objects identified as being similar stand out from other objects in their class.

However, this is not the case for other classes of objects such as applications and system libraries. When applied in its original form, MRS hashing finds too many applications/libraries to be similar, which limits its usefulness. Note that these matches are not false positives; the binary representations of the objects are indeed similar. The observed syntactic similarities are generally artifacts of the particular file format (common headers, etc.) used by the compiler and statically-linked libraries. For example, we discovered (to our surprise) that most of the libraries sampled had repetitive functions. In other words, the same function code was present multiple times. These functions tend to be small and are likely compiler artifacts. Nonetheless, they increase the binary similarity, but are not necessarily indicative of semantic similarity.

Therefore, the fundamental problem is: Is it possible to effectively separate the class-common features (hashes) of an object from its characteristic individual features? Solving this problem would permit the definition of an object class (e.g., Microsoft Word documents) as a set of context-based hashes that are commonly found in such objects. Furthermore, it would lead to at least three important applications:

- The data recovery process is enhanced by eliminating at least some of the false positive results that plague virtually all file carving tools.
- The similarity hashing scheme is enhanced by separating the class-common hashes from object-specific hashes; this would yield more focused similarity results.
- An unstructured target can be searched to estimate the number of objects of different types without reading the file system. Informa-

tion can be obtained after a single sequential pass over the target; partial results could be presented while the operation is underway. This would help in a triage process, which is often faced with a large volume of data.

In addition to aiding regular digital forensic investigations, the latter two applications could help in tricky legal situations where search and seizure must be balanced against privacy concerns. The judicial system has not as yet directly addressed the bounds of what is a reasonable search in the digital world. Nevertheless, the capabilities listed above could provide cause for search, e.g., a disk contains a file that is similar to something relevant or the drive contains a large number of pictures. Just as important, the capabilities could help rule out unlikely candidates.

This paper focuses on the validation of the concept of class-aware similarity hashing. In particular, it attempts to verify the existence of class-specific features that can be captured via hashing, to quantify the number and coverage of these features, and to cross-validate the features by comparing their performance for other classes of objects.

4. Empirical Study

The empirical study used a custom tool that implemented a counting Bloom filter with a single hash function. This is equivalent to using a hash table whose values correspond to the number of data chunks that hash to the particular hash key. The procedure used is a variant of the original MRS hashing scheme.

For each file, given parameters c and t:

1. Hash a sliding window of size c with the `djb2` hash function.
2. If the t rightmost bits are all set to 1, declare a new context match and compute an MD5 hash of the data chunk between the previous context and the current one, and place it in the counting Bloom filter; advance the window by the minimum chunk size (2^{t-2}) and go to Step 1.
   Otherwise, slide the window by one position.
3. If the end of file is reached, exit.
   Otherwise, go to Step 1.

In the case of low-entropy data, a single file often contributes the same hash value multiple times. To address this problem, a local filter is created for each file and the number of hash value contributions is limited to one per key (this is added to the total in the master table). Note that this problem is not due to MRS hashing because it does not use a counting filter.

The next step is to build a histogram which, for a given number k, gives the number of filter locations that have a count k (i.e., k files contain that hash). Based on the histogram, a notion of "coverage" is defined for threshold r – the number of files that contain a hash that has a count of at least r in the master table. Intuitively, it is desirable to obtain maximum coverage with the fewest number of features, so the search starts at the highest frequency and goes down in order. This approach does not guarantee minimal coverage in terms of the number of hashes, but it works fairly well in practice. Two other terms, "relative coverage" and "coverage size," are defined. The "relative coverage" is the fraction of objects covered by hashes with count of at least r. The "size" of a coverage is the number of hashes participating in the coverage.

Seven file sets were used in the empirical study. The first three file sets, whose contents were obtained at random from the Internet, were also used in our previous work [10]. The remaining four file sets contain standard system files as described below.

- *doc:* This set contains 355 files varying in size from 64 KB to 10 MB (total 298 MB).
- *xls:* This set contains 415 files varying in size from 64 KB to 7 MB (total 257 MB).
- *jpg:* This set contains 737 files varying in size from 64 KB to 5 MB (total 121 MB).
- *win-dll:* This set contains 1,243 files (total 141 MB) from a fully-patched WindowsXP `system32` directory varying in size from 3 KB to 640 KB.
- *win-exe:* This set contains 343 files (total 46 MB) from the WindowsXP `system32` directory varying in size from 1 KB to 17 MB.
- *cyg-bin:* This set contains 1,272 files (total 192 MB) from the `bin` directory of Cygwin 2.4 (including all executable files) varying in size from 3 KB to 7.6 MB.
- *ubu-bin:* This set contains 445 files (total 63 MB) from the `/usr/bin` directory of a fully-patched Ubuntu 6.06; the files varied in size from 16 KB to 3.85 MB.

4.1 First-Order Analysis

The first task was to verify the hypothesis that data from different file types exhibits common features that can be captured via context-based

Table 1. First-order analysis of user data.

doc			*xls*			*jpg*		
Hashes	Cov %	Cover	Hashes	Cov %	Cover	Hashes	Cov %	Cover
1	52	188	1	59	245	1	28	212
2	54	195	3	83	345	4	52	388
3	59	212	4	92	382	5	54	400
4	91	325	5	94	394	10	59	439
5	91	325	6	97	403	38	72	536
6	93	331	**7**	**97**	**406**	**42**	**75**	**557**
8	93	333	23	100	415	65	78	579
9	93	333				81	79	585
10	94	334				90	81	604
12	97	346				122	85	629
15	97	347				405	88	653
20	**99**	**352**				3857	98	729
774	100	355						

hashing. One feature is a hash value that is common to a set of data objects of a specific class. The coverage of this feature includes all the objects that contain the feature at least once. Ideally, a relatively small set of features should cover as much as possible of the reference set.

First, we ran our custom tool against a set of 600 files (256 KB each) of random data. The results showed that only two features were common to five different files; all the other features were common to no more than two files. This result is expected – random data should not exhibit any features. High-entropy data objects (e.g., compressed and/or encrypted objects) should exhibit similar results.

Table 1 summarizes the results for three common types of user-created data: Microsoft Word documents (*doc*), Microsoft Excel spreadsheets (*xls*) and JPEG images (*jpg*). All the hash values were generated using similarity hashing as described in Section 4 with the parameters $c = 8$ and $t = 5$. The first column presents the number of hashes in the cover, the second provides the relative coverage (percentage of the file set covered) and the third gives the absolute number of files covered. Thus, the row {5, 91, 335} means that the top five ("most popular") hashes cover 335 files, which constitute 91% of the files in the reference set. Note that several intermediate rows are not shown for reasons of space; only data that represents important trends is presented. Also, the rows presented in boldface represent the coverage chosen for the cross-analysis study in the next section.

Table 2. First-order analysis of system executables.

win-dll			***win-exe***			***cyg-bin***			***ubu-bin***		
Hashes	Cov %	Cover	Hashes	Cov %	Cover	Hashes	Cov %	Cover	Hashes	Cov %	Cover
1	41	510	1	44	151	1	11	146	1	53	239
2	58	733	3	46	158	2	22	285	2	64	285
4	68	853	4	77	265	36	30	384	3	78	351
9	71	886	5	78	267	49	36	458	4	82	365
17	75	933	6	79	271	90	41	529	6	84	377
43	**80**	**1004**	7	80	273	105	49	624	9	85	379
122	85	1061	8	86	295	144	55	706	**33**	**91**	**407**
541	90	1120	11	87	296	276	61	778	50	91	409
2478	95	1193	**12**	**89**	**305**	**654**	**67**	**853**	1100	92	412
5390	97	1215	56	90	306	1947	72	921	3208	93	416
14208	98	1228	139	91	310	3332	75	958	5820	93	417
36716	99	1237	332	95	324	7013	80	1022	6648	94	419
			453	95	325	16913	86	1096	9192	95	424
			987	96	329	29985	89	1138	42238	97	435
			9873	98	334	65119	93	1190			

The results show that *doc* and *xls* files have compact and easily identifiable feature hash sets or "classprints" that represent the types. In the case of *doc* files, only 20 feature hashes are required to provide 99% coverage. The top four give 91% coverage, so choosing the cut-off point can be somewhat subjective. The results are not as good for *jpg* files, where a substantially larger feature set is required to cover the reference files. Intuitively, the larger the feature set, the more instance-specific the features it includes.

In all cases, the feature set was kept relatively small and the inflection point was chosen so that the rate at which features need to be added was greater than the rate at which coverage was increased. For example, in the *jpg* case, the increase from 10 to 38 hash values yields an increase in coverage from 59% to 72%; the next step, from 38 to 42 is relatively small and yields a correspondingly modest improvement from 72% to 75%. However, the increase from 42 to 65 only yields an improvement of 75% to 78%. Therefore, 42 was chosen as the cut-off point for the experiments in the next section.

The analysis of system executables shows some interesting results (Table 2). The sets were chosen so they had various degrees of commonality. Specifically, all the sets primarily contain executable code for the Intel x86 architecture. Although other resources could be bundled into an

Table 3. Feature set intersection.

	doc	xls	jpg	win-dll	win-exe	cyg-bin	ubu-bin
doc		3 (17%)					
xls	3 (43%)						
jpg							
win-dll					9 (21%)	1 (2%)	
win-exe				9 (75%)			
cyg-bin				1 (0.2%)			1 (0.2%)
ubu-bin						1 (3%)	

executable, these are relatively small system utilities that are unlikely to contain much beyond code. The *win-dll*, *win-exe* and *cyg-bin* file sets all contain Microsoft Windows code. The *cyg-bin* files correspond to the Windows portion of the utilities under Unix/Linux, which are contained in the *ubu-bin* file set. Both these types of files are compiled using `gcc`.

The main observation is that it is easy to identify the inflection points for the *win-dll*, *win-exe* and *ubu-bin* file sets, but not for the *cyg-bin* set. Part of the reason could be that *cyg-bin* contains more files than two of the other sets; however, *win-dll* has about the same number of files and does not have the same problem. The reference cover that was picked has substantially more hash values (654) than for any of the other sets, still the coverage is much lower – only 2/3 of the reference set.

In summary, the observed data shows that it is possible to define a class-common feature set based on similarity hashes. The next task is to establish whether or not these features are "class-defining," i.e., they are generally not present among the features of other classes.

4.2 Second-Order Analysis

Clearly, if the class-common features that are discovered are shared by multiple classes, their analytical value is significantly diminished. A second-order analysis was undertaken because there were reasons to believe that some of the chosen sets may share features.

For completeness, all 21 possible (unordered) pairs of feature sets were compared, and their intersections were computed in relative and absolute terms. The results are presented in Table 3, which only shows the non-zero elements. The table is symmetric in terms of the absolute numbers; the figures in parentheses correspond to the intersections as a fraction of the total number of features for the associated set (row). For example, the *xls* and *doc* sets have three features in common, which

represents 43% of all features for the *xls* files and 17% of the features for the *doc* files.

The results indicate that the {*doc*, *xls*} and {*win-dll*, *win-exe*} file set pairs cannot be considered independent, which is not entirely unexpected. Nevertheless, just one feature from the intersection can provide a useful hint about the content of a target because it helps eliminate a large number of possibilities.

4.3 Estimating Drive Content

The next test involved the application of the *doc* feature set to estimate the number of `.doc` files in a 7.2 GB Windows partition residing on a personal laptop. First, the reference set was examined and the average number of features matched by each file was computed. Next, the number of matches against the unknown target was used to estimate the number of `.doc` files in the Windows partition.

As it turned out, the original reference set was not ideal for this purpose – it contained many files that had a very large number of feature matches (the "top" file had 547 matches). Upon closer review, it was discovered that this file contained a huge amount of repetitive information. Clearly, a more systematic approach for selecting reference sets would help avoid problems in such pathological cases.

Nonetheless, the median of nine feature matches per file was taken and applied to the target Windows partition that had yielded 298 feature matches. Thus, it was estimated that there were $298/9 = 33$ Microsoft Word documents on the partition. The actual count was 68, so the estimate was off by a factor of two.

The approach has some potential, but more research is needed to improve and validate this technique. Still, it is notable that features from a training set were applied to a completely unknown and unrelated target; this is evidence that the identified features are generic class features.

Another interesting point pertains to the throughput of the operation. The single-threaded, unoptimized version of the code was able to perform the search in 2 hours and 44 minutes, corresponding to a rate of 45 MB/s. This is significant because the code is parallelizable so 2-4 threads on a dual- or quad-core processor should keep up with the sustained 80-100 MB/s transfer rate of current large-capacity HDDs. In other words, valuable information could be obtained during the initial cloning of a target without incurring any latency overhead. Furthermore, the operation is constrained by hash value generation, so estimates for multiple types of data could easily be performed in a single run with virtually no impact on performance.

5. Conclusions

Class-aware similarity hashing is an attractive technique for automatically extracting class-defining feature sets (classprints) and for identifying data types based on content. Our empirical study demonstrates that classprints can be generated for several common file types; in other words, the file types can be defined solely in terms of syntactic features of their binary representation. The overall scheme requires a modest amount of storage during the extraction phase and a negligible amount for the classprints. Experiments indicate that hashing rates above 1 Gbit/s can be sustained; this exceeds the transfer rates of current generation high-capacity (500 GB+) hard drives. The hashing scheme also enables investigators to ask very generic questions about targets without violating privacy concerns. In fact, it is possible to discover whether or not a drive contains documents (or document remnants) of a particular type without examining file names or metadata.

References

[1] S. Brin, J. Davis and H. Garcia-Molina, Copy detection mechanisms for digital documents, *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 398–409, 1995.

[2] B. Bloom, Space/time tradeoffs in hash coding with allowable errors, *Communications of the ACM*, vol. 13(7), pp. 422–426, 1970.

[3] A. Broder, S. Glassman, M. Manasse and G. Zweig, Syntactic clustering of the web, *Proceedings of the Sixth International World Wide Web Conference*, pp. 391–404, 1997.

[4] A. Broder and M. Mitzenmacher, Network applications of Bloom filters: A survey, *Internet Mathematics*, vol. 1(4), pp. 485–509, 2005.

[5] J. Kornblum, Identifying almost identical files using context triggered piecewise hashing, *Proceedings of the Sixth Digital Forensic Research Workshop*, 2006.

[6] National Institute of Standards and Technology, National Software Reference Library, Gaithersburg, Maryland (www.nsrl.nist.gov).

[7] D. Patterson, Latency lags bandwidth, *Communications of the ACM*, vol. 47(10), pp. 71–75, 2004.

[8] M. Rabin, Fingerprinting by Random Polynomials, Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, 1981.

[9] V. Roussev, Y. Chen, T. Bourg and G. Richard III, md5bloom: Forensic file system hashing revisited, *Proceedings of the Sixth Digital Forensic Research Workshop*, 2006.

[10] V. Roussev, G. Richard III and L. Marziale, Multi-resolution similarity hashing, *Proceedings of the Seventh Digital Forensic Research Workshop*, 2007.