Chapter 1

CLASSPRINTS: CLASS-AWARE SIMILARITY HASHES

Hash-based Classification of Data

Vassil Roussev, Golden G. Richard III, and Lodovico Marziale
Department of Computer Science, University of New Orleans
New Orleans, Louisiana 70148, USA
vassil,golden,vico@cs.uno.edu

Abstract In this paper, we introduce the notion of *class-aware similarity hashes*, or *classprints* which is an outgrowth of recent work on similarity hashing. Specifically, we build on the notion of context-based hashing to design a framework both for identifying data type based on content, and for building characteristic similarity hashes for individual data items that can be used for correlation.

The most important feature of the presented work is that the process can be fully automated and no prior knowledge of the underlying data is necessary, beyond the selection of a training set of objects. The approach relies entirely on these representative sets to characterize a particular data type. We present an empirical study which demonstrates the practicality of this work on real data and sketch out a complete implementation.

Keywords: Digital forensics, similarity hashing, classprints, class-aware similarity hashing

1. Introduction

The problem of identifying the type of data inside a container, such as a file or disk image, has been studied since the very beginning of digital forensics, yet very few positive results have been published. The ability to identify the underlying type of the data without the help of the file system metadata comes in very handy in data recovery (file carving) operations to either validate or invalidate the currently attempted recov-

ery. For example, if a tool is trying to carve out a *JPEG* file and runs into plain text data, it is clear that the process is not on the right track. Data carving is routinely applied to target images to recover (fragments of) deleted data and is often a critical source of information.

Another related line of research that is automated data correlation. With the exponential capacity growth, targets can easily encompass multiple terabytes of data so the ability to quickly separate the potentially relevant from the clearly irrelevant information will have a great impact on the length and accuracy of a forensic inquiry. One of the most powerful tools in that regard is the ability to use prior accumulated dat to make that separation. In traditional (physical) forensics this includes a large and sophisticated set of different databases that can help an investigator quickly zero in on the relevant. Unfortunately, in the digital world, we are well behind the curve of what is needed. Currently, the only success story is the use of sets of file hashes of known system and application files, such as the ones maintained by NIST [6] and commercial vendors. Yet those hashes are a drop in the bucket and it is unclear how long this approach can be extended into the future as more and more hashes are added–are we going to need compute clusters just to do hash searches?

Traditional, file-based (cryptographic) hashes have their place but are also a very fragile tool–they must know the *exact* binary representation of all versions of the objects of interest. Recently, a few schemes have been proposed that approach the issue of finding similarity among objects. In [5], Kornblum proposed a context-based approach to dynamically split up the file into individually hashable chunks from which a composite hash is produced. While the use of hash-based context (which can be traced back to early work in information retrieval such as [1] and [3], and is ultimately derived from Rabin's original work [8]) is a proven idea, the rest of the scheme lacks robustness. At the same time, we proposed a significantly more robust approach based on Bloom filters [2], [4] but lacked an elegant mechanism to split up arbitrary targets.

In [10] we combined those two ideas with a sizeable body of experimental results and came up with the idea of *Multi-Resolution Similarity* (MRS) hashing that can be applied to arbitrary targets. Indeed the results allowed us to quite clearly relate data files that would be classified by a human as related, such as different drafts of the same document. Also, we were able to identify the presence of the content of a file (e.g. a JPEG) inside a larger target (raw drive image) without any knowledge or assistance from the file system.

The latter property, in addition to making the tool generic, also carries significant performance advantages stemming from the fact that a single

sequential pass over the image is required. In contrast, any file-based tool requires access to file metadata, which results in a non-sequential disk access pattern. Figure 1 illustrates the effects of non-sequential access on the throughput of a modern hard drive, as measured by Intel's IOMeter tool (iometer.org). As little 2% randomness in the work load can cost 30% in performance penalty while 5% can cut performance in half. Currently, forensic tool design appears oblivious to this issue.

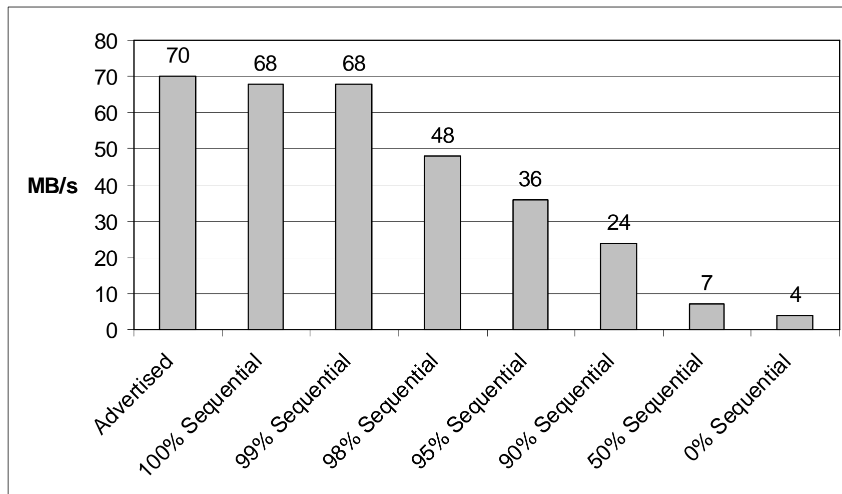

Figure 1. Observed HDD throughput for WDC WD5000KS (500GB)

With capacity growth outpacing both bandwidth and latency improvements [7], forensic target are, in fact getting bigger relative our capacity to process them on time. Therefore, building performance-consious tools should be a priority for researchers in the field.

The rest of the paper is laid out as follows. First, we briefly review the similarity hashing techniques relevent to this work. Next, we outline the ideas and approaches designed to extend it. Finally, we present some experimental results in support of our conjectures, and summarize the results.

2. Background–Similarity Hashing

In this section we *briefly* summarize our recent work on similarity hashing; for a more in-depth discussion, please refer to [10].

*Block hashing.*The most basic scheme that can be used for determining similarity of binary data is block-based hashing. In short, crypto hashes are generated and stored for every block of a chosen fixed size (e.g. 512 bytes). Later, the block-level hashes from two different sources

can be compared and, by counting the number of blocks in common, a measure of similarity can be determined. The main advantage of this scheme is that it is already supported by existing hashing tools and it is computationally efficient–the hash computation is faster than disk I/O.

The disadvantages become fairly obvious when block-level hashing is applied to discover file similarity. Success heavily depends on the physical layout of the files being very similar. For example if we search for versions of a given text document, a simple character insertion/deletion towards the beginning of the file could render all block hashes different. Similarly, block-based hashes will not tell us if an object, such as a JPEG image, is embedded in a compound document, such as an MS Word document. In short, the scheme is too fragile and a negative result does not reveal any information.

Context-triggered piecewise (CTP) hashing. In [5], Kornblum proposed an approach that overcomes some of the limitations of block-based hashes and presents an implementation called *ssdeep*. The basic idea is to identify content markers, called *contexts*, within a (binary data) object and to store the sequence of hashes for each of the pieces (or chunks) in between 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 the content of the object. 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.

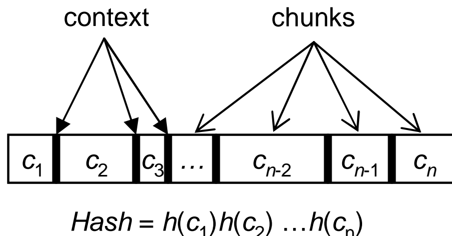

Figure 2. Context-based hashing (a.k.a. *shingling*)

To identify a context, *ssdeep* uses a rolling hash over a window of $c = 7$ bytes, which slides over the target. If the lowest t 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 exact value of t depends on the size of the target as the tool generates a fixed-size result. Intuitively, a bigger t produces less frequent context matches and reduces the granularity of the hash.

Bloom filter similarity hashing. In [9], we developed a scheme, which utilizes Bloom filters to derive object similarity. The basic idea is to use the (known) structure of an object to break it into components which are individually hashed and placed into a (Bloom) filter. Using the mathematical properties of filters, we demonstrated both analytically and empirically that the bitwise comparison of filters can yield a very useful measure of the similarity between the binary representations of two (or more) objects.

In [10] we further developed this idea by combining it with context-based object decomposition (or *shingling* in the terminology of [3] to handle arbitrary binary data. We also devised a standardized multi-resolution scheme which allows: a) objects of arbitrary sizes to be hashed without loss of resolution; b) objects of various size to be effectively compared, for example, it is practical to search for (the remnants of) a 1MB file inside a target that is over 100GB.

Another important property is that, due to the use of Bloom filters as a basic builing block, the resulting hashes are extemely memory efficient–they require no more than 0.5% of the size of the target. Thus, the complete multi-resolution hash of a 500GB hard drive can fit in the main memory of a modern workstation.

Performance-wise, the MRS hash generation scheme is no more expensive than a block-based MD5 hash, even in its early (unoptimized) version. The comparison step is very efficient and can be sped up by using lower resolution for large targets and/or delegating it to the graphics processor which, in our experience, can speed up the process 20 times on an NVidia G80 processor.

3. Class-aware Similarity Hashing

As discussed in the preceding section, MRS hashes are a very sensitive and tunable tool in terms of finding similarities among binary data objects. However, what is not clear so far is *why* are the objects similar? From our previous work, it appears that for user-generated artifacts (e.g. jpg, doc, pdf files) the existing MRS scheme works reasonably well in that the identified similar objects stand out from the rest of the objects of the same 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. We should note that these are *not* false positives–the binary representations of these objects are indeed similar. The observed syntactix similarities are generally artifacts of the particular file for-

mat (common headers, etc.), the compiler used, and statically-linked libraries. For example, in some early experiments, we identified (much to our surprise) that most of the libraries we sampled had repetitive functions. In other words, the *exact* 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 higher semantic similarity of the compared objects, which is the typical goal of an investigation.

Thus, the main question we focus on in this work is: Is it possible to effectively separate the class-common features (hashes) of an object from its characteristic individual features? Solving this problem would allows us to define an object class (e.g. MS Word documents) as a set of (context-based) hashes that are commonly found in such objects. A positive outcome has at least three forensically-importnant applications:

- We can enhance the data recovery process by helping to eliminate at least some of the false positive results that currently plague virtually all file carving tools in existence.
- We can enhance the similarity hashing scheme by splitting up the class-common from the object-specific hashes, which would yield more focused similarity results.
- We can search an unstructured target to estimate the number of objects of different types without resorting to reading the file system. This is a significant advantage as we can obtain the information after a single sequential pass over the target (partial results could, of course, be presented while the operation is under way). This could help in the triage process when faced with a large volume of data.

Quite apart from aiding in regular investigatations, the latter two application could help in some tricky legal situations where search and seizure must be balanced against privacy concerns. While the judicial system has not yet directly addressed the bounds of what is a reasonable search in the digital world, the above capabilities could provide cause for search, e.g., the disk contains file that is similar to something relevant, or the drive contains a large number of pictures. Conversely, it could help rule out unlikely candidates.

The main thrust of this paper is to validate the concept of class-aware similarity hashing. In other words, we must verify the existence of class-specific features that can be captured through hashing, quantify the number and coverage of these features, and cross-validate them by comparing them with other classes.

4. Empirical Study

The actual experiments are based on a custom tool, which utilizes a counting Bloom filter with a single hash function. (This is equivalent to a hash table which stores as values the number of data chunks that hash to the particular hash key.) The procedure is a variant on the original MRS hashing scheme.

For each file, given parameters c and t:

- Hash a sliding window of size c with the *djb2* hash function.
- If the t rightmost bits are all set to 1, declare a new context match and *md5*-hash 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 back to *djb2*-hashing;
- Otherwise, slide the window by one position and go back to *djb2*-hashing.

To avoid the potential problem of a single file contributing the same hash multiple times (a real issue with low-entropy data), we create a local filter for each file and limit the number of contributions to one per key and then add them to the total in the master table. (This is not a problem with the actual MRS hash because it does not use a counting filter.)

After this step, we build a histogram which, for a given number k, gives us the number of filter locations that have a count of k (that is, k files contain that hash). Based on the histogram, we can define a notion of *coverage* for threshold r–the number of files that contain a hash that has a count of at least r in the master table. Intuitively, we would like to obtain maximum coverage with the fewest number of features, so we start with the highest frequency and go down in order. It is not difficult to see that this approach does not guarantee *minimal* (in the number of hashes) coverage but it works fairly well in practice. We also define *relative coverage* as the fraction of objects covered by hashes with count of at least r. The *size* of the coverage is the number of hashes participating in the coverage.

4.1 Reference File Sets

Below are brief descriptions of the file sets we used in the experiments, along with their corresponding mneumonic abbreviation used in the result presentation. Note that the first three ones were also used in our

previous work [10] and were obtained at random from the Internet. The rest are standard sets of system files, as described.

- *doc* The sample contained 355 files varying in size from 64KB to 10MB for a total of 298MB of data.
- *xls* 415 files, 64KB to 7MB, 257MB total.
- *jpg* 737 files, 64KB to 5MB, 121MB total.
- *win-dll* 1,243 files from a fully-patched WindowsXP's **system32** directory ranging between 3KB and 640KB, 141MB total.
- *win-exe* 343 files from the WindowsXP's **system32** directory between 1KB and 17MB, 46MB total.
- *cyg-bin* 1,272 files from the **bin** directory of Cygwin 2.4 (this includes all executable files); sizes: 3KB-7.6MB, 192MB total.
- *ubu-bin* 445 files from the **/usr/bin** directory of a fully-patched Ubuntu 6.06, 16KB-3.85MB, 63MB total.

4.2 First Order Analysis: File Set Features

Our first order of business is to establish our hypothesis that data from different file type does indeed exhibit common features that can be captured via context-based hashing. A feature in this context is a hash that is common to a set of data objects of a specific class. The coverage of this feature comprises of all the objects that contain that feature at least once. Ideally, we would like to see a relatively small set of features cover as much as possible of the reference set.

As a simple sanity check, we ran our code first against a set of 600 files (256KB each) of random data. The results showed that only two features were common to five different files, with all the rest common to no more than two files. This is precisely what we expected–random data should not exhibit any features. By extension, high-entropy data objects (compressed and/or encypted) cannot be analyzed in this manner. Figure 3 summarizes our findings with respect to three common types of user-created data: MS Word documents (doc), MS Excel spreadsheets (xls), and JPEG images (jpg). For each type, the first column gives the number of hashes in the cover, the second provides the relative coverage (percent 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 5 ('most popular') hashes cover 335 files, which constitutes 91% of the total number of files in the reference set. Note that, both in this figure

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						

Figure 3. First-order analysis of user data

and the next, a good number of intermediate rows have been deleted to reduce space requirements. We have picked points that represent the overall trends. We should also mention that all hashes are generated as described in the Similarity Hashing section with parameters $c = 8$ and $t = 5$.

It is quite clear that for *doc* and *xls* files there are compact and easily identifiable feature hash sets, or *classprints* that represent the types. In the case of *doc* files, we only need 20 feature hashes to provide 99% coverage. It is notable that the top four give 91% coverage so choosing the cut-off point can be somewhat subjective. (The rows in bold represent the coverages we have chosen for the cross analysis in the next section.) For *jpg* files things are a bit more problematic as we need substantially larger feature set to cover the reference files. Intuitively, the larger the feature set the more instance-specific the features it includes. In all cases, we have tried to keep the feature set relatively small and we chose the inflection point where the rate at which we need to add features is greater than the rate at which we increase coverage. For example, in the *jpg* case, the jump from 10 to 38 hashes, 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 following increase from 42 to 65 only yields an improvement of 75 to 78%, therefore, the 42 was chosen as the cut off point for the experiments in the next section.

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			

Figure 4. First-order analysis of system executables

The analysis of the system executables (Figure 4) shows some interesting results. The sets were chosen so they had various degrees of commonality. First, all of them represent primarily executable code for the Intel x86 architecture. Although other resources could be bundled into an executable, these are relatively small system utilities that are unlikely to contain much beyond code. Next, the *win-dll*, *win-exe*, and *cyg-bin* all represent code for MS Windows. Finally, the *cyg-win* files are a Windows port of the same utilities under Unix/Linux, as represented by the *ubu-bin* set, and are compiled with the same compiler–*gcc*.

The main observation is that it is very easy to identify the inflection points for the *win-dll*, *win-exe*, and *ubu-bin* sets but not the *cyg-bin* one. Part of the reason could be that it contains more files than two of the other sets, however, *win-dll* has about the same number of files and exhibits no such issues. The reference cover we picked has substantially more hashes than for any of the other sets (654) yet the coverage is substantially lower–only 2/3 of the reference set.

In summary, the observed data shows that it is, indeed, possible to define a class-common feature set based on similarity hashes. The next important question is to establish whether this features are class-*defining* in that they are generally not present among the features of other classes.

4.3 Second Order Analysis: Cross-Set Correlation

It is quite clear that if the class-common features discovered are shared by multiple classes, their analytical value will be significantly diminished. Among the chosen sets, there are reasons to believe that, at least some of these set, share features. For completeness, we compared all 21 possible

	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%)	

Figure 5. Feature set intersection

(unordered) pairs of feature sets and calculated their intersection both in relative and in absolute terms. The results are presented on Figure 5 with only the non-zero elements shown. The table is symmetrical in terms of the absolute numbers, however, in parenthesis we have put the intersection as a fraction of the total number of features for the *row* set. For example, the *xls* and *doc* sets have 3 features in common, which represents 43% of all features for the *xls* files and 17% of the features for the *doc* files.

It is clear that the {*doc*, *xls*} and {*win-dll*, *win-exe*} set pairs cannot be considered independent, which is hardly an unexpected result. Yet, even a feature from the intersection can be a useful hint as to the content of a target as it helps eliminate a large number of other possibilities.

4.4 Example Use: Estimating Drive Content

After completing the above analysis we decided to apply the collected *doc* feature set to a 7.2GB Windows partition residing on the personal laptop of one of the authors. The basic idea is to look back at the reference set and calculate how many features (on average) each of the files mathes. Then, using the number of matches against the unknown target we can roughly estimate the number of *doc* files present.

As it turns out, the original reference set was not ideal for this purpose– it contained a number of files that had a very high number of feature matches with the 'top' file containing 547 feature set matches. Upon

closer review, the file contained a huge amount of repetitive information. Evidently, a more systematic approach to selecting reference sets would be helpful in avoiding such pathological cases.

Nonetheless, we took the median of 9 feature matches per file and applied to the target Windows partition which had yielded 298 feature matches. Thus, our best guess would be that there are approximately $298/9 = 33$ MS Word documents on the partition. The actual count was 68 so our estimate was off by a factor of two. While more work is needed to improve and empirically validate this approach, we see some potential here.

It is also notable that, implicitly, we applied the features from our training set to a completely unknown and unrelated target, which is further evidence that the identified features are generic class features.

Another interesting detail is the throughput of the operation. The single-threaded, unoptimized version of the code was able to perform the search in 2:44min, or at the rate of 45MB/s. This is significant because the code is readily paralellizable so 2-4 threads on a dual- or quad-core processor should be quite capable of keeping up with the sustained 80-100MB/s tranfer rate of current generation of large-capacity HDD. In other words, this kind of information could be obtained, for example, during the initial cloning of a target without incurring any latency overhead. Further, the operation is hash-generation constrained so estimates for multiple types of data could easily be performed in a single run with virtually no effect on performance.

5. Conclusions

In this paper we motivated and justified the introduction of an enhanced similarity hashing scheme called *class-aware similarity hashing*. We established empirically that, for several classes of commonly-used file types, it is possible to automatically extract class-defining feature sets using context-based hash generation. In other words, we showed that it is practical to define common file types based solely on syntactic features of their binary representation. The proposed approach has the following properties:

- *Genericity and Scalability.* The approach can be applied to any data sets of practical size and arbitrary type. By relying solely on the binary object representation without any knowledge of the object's structure we can apply the scheme to, for example, case-specific data that is not supported by standard tools.

- *Automation.* The scheme allows complete automation–all it needs is reference groups of files representing the different user-defined

types. After the training is completed, the derived feature sets can be automatically to the raw data that needs to be classified.

- *Ease of Use.* The tool does not require any specific qualifications from the user and needs no deep understanding of the underlying methods to obtain and interpret the results.

- *Space Efficiency.* Our experiments show that the typical feature set consists of a few dozen features. Using Bloom filters, it could be represented in 256 bytes and have false positive rates of less than 1 in 10,000.

- *High Performance.* The generation of feature sets requires a single hashing pass over the reference sets (a second one maybe required if a more sophisticated feature selection algorihtm is used). The actual observed speed for a single-threaded implementation of 45MB/s shows that an improved version should be able to able to keep up with the sequential transfer rates of modern large-capacity hard drives.

- *Privacy Preservation.* The use of hashes as proxies for the actual data enables some generic inquiries to be performed without reading (interpretting) the actual data. This is likely to help in many delicate situations arising at the beginning of many investigation and would allow legitimate privacy concerns to be addressed.

6. Future Work

The presented work is only the first step in what we see as a long-term project with the ultimate goal to bring as much as possible of established information retrieval techniques into the forensics domain. So far the field has been hobbled by the fact that most such techniques are designed to work on text. However, we showed it is possible (with some likely limitations) to apply many of the notions, such as statistically unlikely features to quickly discern likely related objects without before interpreting them (through an application).

Shorter term, we would like to come up with a better feature selection algorithm that minimizes the feature set while maximizing coverage, modify the MRS hashing tool to separate out the class-common from the instance specific features so those could be examined separately. On the experimental side, we would like to perform a much larger scale experiment to gain more insight into the practical aspects of the developed scheme.

References

[1] S. Brin, J. Davis, H. Garcia-Molina, Copy detection mechanisms for digital documents, *Proceedings of the ACM SIGMOD Annual Conference*, San Francisco, CA, May 1995.

[2] B. Bloom, Space/time tradeoffs in hash coding with allowable errors, *Communications of the ACM*, vol 13 no 7, pp. 422-426, 1970.

[3] A. Broder, S. Glassman, M. Manasse, and G. Zweig, Syntactic clustering of the web, *Proceedings of the 6th International WWW Conference*, pages 1157-1166, Santa Clara, CA, Apr. 1997.

[4] A. Broder and M. Mitzenmacher, Network applications of Bloom filters: a survey, *Internet Mathematics*, vol. 1. no. 4, pp. 485-509, 2005.

[5] J.Kornblum, Identifying almost identical files using context triggered piecewise hashing, *Proceedings of the 6th Annual DFRWS Conference*, Aug 2006, Lafayette, IN.

[6] National Software Reference Library, http://www.nsrl.nist.gov/index.html.

[7] D. Patterson, Latency Lags Bandwidth, *Communications of the ACM*, Vol. 47(10), 2004.

[8] M. Rabin, Fingerprinting by random polynomials. *Technical Report TR-15-81*, Center for Research in Computing Technology, Harvard University, 1981.

[9] V.Roussev, Y.Chen, T.Bourg, G.G.Richard III, md5bloom: Forensic filesystem hashing revisited, *Proceedings of the 6th Annual DFRWS Conference* (DFRWS'06). Aug 2006, Lafayette, IN.

[10] V. Roussev, G.G. Richard III, L. Marziale, Multi-resolution similarity hashing, *Proceedings of the 7th Annual DFRWS Conference* (DFRWS'07). Aug 2007, Pittsburgh, PA.