available at www.sciencedirect.com
journal homepage: www.elsevier.com/locate/diin

Multi-resolution similarity hashing

Vassil Roussev[*], *Golden G. Richard III*, *Lodovico Marziale*

Department of Computer Science, University of New Orleans, New Orleans, LA 70148, United States

Keywords:
Hashing
Similarity hashing
Digital forensics
Multi-resolution hash
File correlation
Data correlation

ABSTRACT

Large-scale digital forensic investigations present at least two fundamental challenges. The first one is accommodating the computational needs of a large amount of data to be processed. The second one is extracting useful information from the raw data in an automated fashion. Both of these problems could result in long processing times that can seriously hamper an investigation.

In this paper, we discuss a new approach to one of the basic operations that is invariably applied to raw data – hashing. The essential idea is to produce an efficient and scalable hashing scheme that can be used to supplement the traditional cryptographic hashing during the initial pass over the raw evidence. The goal is to retain enough information to allow binary data to be queried for similarity at various levels of granularity without any further pre-processing/indexing.

The specific solution we propose, called a *multi-resolution similarity* hash (or MRS hash), is a generalization of recent work in the area. Its main advantages are robust *performance* – raw speed comparable to a high-grade block-level crypto hash, *scalability* – ability to compare targets that vary in size by orders of magnitude, and space *efficiency* – typically below 0.5% of the size of the target.

1. Introduction

One of the main performance bottlenecks of (digital) forensic processing, especially in large cases, is the disk I/O time. For example, for a commodity Seagate 750 GB Barracuda ES SATA hard drive it would take approximately 2:40 h to read the entire drive at the advertised *sustained* (sequential) transfer rate of 78 Mbits/s. (It is worth noting that manufacturers like to draw attention to the bandwidth of the buffer-to-host connection, which is up to 3 GB/s but, for large reads, it is the disk-to-buffer rate that is the limiting factor.) In other words, creating a working copy of the original target would take at least that much time. At the end of the operation, virtually nothing of forensic interest will be known about the target. Thus, at least one more pass will be performed by the investigator's forensic tool of choice before any real work can begin rounding out a full workday with no real work accomplished. Unfortunately, reality is much worse – the forensic pre-processing of a target of this size is likely to take several days using current tools. Part of the problem lies not with the tools per se but with the file-centric processing, which generates a much more randomized I/O workload than a straight sequential read.

The primary goal of this work is to develop a scheme that, eventually, allows for digest information to be extracted from the target *while* it is being copied. Unlike traditional hashes, which can only answer yes/no comparison questions, we would like to provide a measure of *similarity* among raw data objects. This in turn can be used to find versions of the same object, find related objects, or to automatically classify related information.

The exact same mechanism can also be applied to compare files of various types, as our experiments show. We should

[*] *Corresponding author.*
E-mail addresses: vassil@cs.uno.edu (V. Roussev), golden@cs.uno.edu (G.G. Richard III), lmarzial@cs.uno.edu (L. Marziale).

1742-2876/$ – see front matter © 2007 DFRWS. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2007.06.011

emphasize that this kind of work is designed to *supplement*, not replace type-specific processing. It is useful to think of this mechanism as being one of first and/or last resort. In other words, it can be used like traditional hashes in the beginning as a quick way to narrow down the scope of the investigation, but it can also be applied to binary data for which no type-specific tools are available (e.g. raw data dumps).

2. Related work

Over the last two decades, there has been a significant amount of work in the area of information retrieval (IR) that deals with approximate string matching. Yet, from a forensic perspective, there are a host of challenges that have not been addressed as they are not a concern in the IR field. Those tend to fall in two categories: execution time and target domain. Generally, IR systems have all the time in the world to perform any pre-processing they need and the solutions are targeted at *specific* domains, most often text. In the case of web search engines, there is also explicit information that links objects together. In forensics, there is the distinct need to have *generic* tools that work *fast* and help sift through terabytes of raw data. Naturally, we cannot expect such tools to have the fidelity of domain-specific solutions but should be effective in finding binary similarity among digital artifacts.

Much of the IR work has focused on web data and with the goal of either finding near-identical document, or to identifying content overlap. Brin et al. (1995) pioneered the use of word sequences to detect copyright violations. Follow-up work by Shivakumar and Garcia-Molina has focused on improving and scaling up the approach (Shivakumar and Garcia-Molina, 1995, 1996, 1998).

Currently, the state-of-the-art is represented by Broder et al. (1997) and Charikar (2002) and all other techniques use them as benchmarks and/or starting point for further development. Broder uses a sample of representative "shingles" consisting of 10-word sequences as a proxy for the document, whereas Charikar constructs locality sensitive hash functions to use in similarity estimation. A detailed explanation of these (and related) algorithms is beyond the scope of this paper. However, it is worth noting that almost invariably, the approaches are optimized for text documents. Evaluation/comparison studies (e.g. Monostori et al., 2002) of these and other methods have focused primarily on accuracy and recall measurements and, to some degree, the number of documents compared. We could not find any execution time numbers although we conjecture that the author of Henziger (2006) was in fact able to get reasonable performance by using Google's massively parallel infrastructure (Dean and Ghemawat, 2004).

Overall, we have argued (Roussev and Richard, 2004) that the use of distributed processing is necessary if these and other advanced techniques are to be utilized in routine forensic analysis. Indeed, the economics of such use have improved ever since down to the point where the rent for 1 CPU for an hour has reached $0.10 (http://amazon.com/ec2). Yet, bandwidth limitations (uploading a target) are an obstacle. Thus, realistic forensic use is largely limited to a single computer system and that is the focus of this work.

Currently, we are only aware of only two forensic similarity hashing schemes – Kornblum (2006) and Roussev et al. (2006) – that have been recently proposed at DFRWS 2006. As our further analysis will show, they contain complimentary ideas and serve as a starting point for this work.

2.1. *Block-based 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 files. 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 JPEG image, is embedded in a compound document, such as MS Word document. In short, the scheme is too fragile and a negative result does not reveal any information.

2.2. *Context-triggered piecewise (CTP) hashing*

Kornblum (2006) 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 (Fig. 1). 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.

To identify a context, *ssdeep* uses a rolling hash over a window of 7 bytes, which slides over the target. If the lowest *k* bits

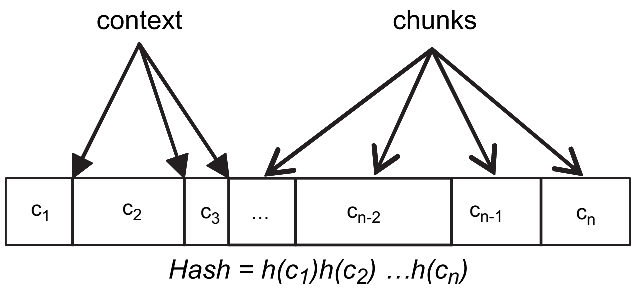

Fig. 1 – Context-based hashing.

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. (This approach can be traced back to Brin et al., 1995.) The exact value of k depends on the size of the target as the tool generates a fixed-size result. Intuitively, a bigger k produces less frequent context matches.

It is difficult to formally analyze and to prove that this context-based approach to hashing yields reliable results. However, experimental results show that, for realistic targets, the basic idea is sound and works well. In Section 3, we provide a more detailed analysis of the specific design and implementation choices made in *ssdeep*. At this point we will note that, while *Kornblum's* work certainly deserves the credit for bringing the idea to forensics, context-based hashing is also a starting point for a number of possible implementations each with their own advantages and shortcomings.

2.3. *Bloom filter hashing*

Traditionally, Bloom filters (Bloom, 1970) have been used for space-efficient set representation. The price for efficiency is the probability that membership queries may return false positive (but not false negative) result. Crucially, the relationship is mathematically quantifiable and, for collision-resistant functions (e.g. MD5), empirical observations closely match the theoretical framework (which depends on the hash functions being perfect).

The original Rabin–Karp string-searching algorithm Karp and Rabin (1987) uses Bloom filters to speed up the search for multiple string matches. It can be viewed as a precursor to the presented forensic use of Bloom filters, with the notable difference that it utilizes them as a helper mechanism, not as proxies for the search of the original data.

Before we present the proposed forensic use of Bloom filters, we provide a *brief* overview of (Bloom) filters in general with some of the most relevant results. Our discussion and notation follow the framework presented in Broder and Mitzenmacher (2005) and Mitzenmacher (2002) and have absolutely no claim to being exhaustive.

In short, a filter B is a representation of a set $S = \{s_1, \ldots, s_n\}$ of n elements from a universe (of possible values) U. The filter consists of an array of m bits, initially all set to 0. The ratio $r = m/n$ is a key design element and is usually fixed for a particular application. To represent the set elements, the filter uses k independent hash functions $h_1, \ldots, h_k$, with a range $\{0, \ldots, m-1\}$. All hash functions are assumed to be independent and to map elements from U uniformly over the range of the function.

To insert an element s from S, the hash values $h_1(s), \ldots, h_k(s)$ are computed and the corresponding k bit locations are set to 1. The same process is repeated for every element in S. Note that setting a bit to 1 multiple times has the same effect as doing it once. Fig. 2 sketches the insertion of two consecutive elements – s_1 and s_2 – into an empty *Bloom* filter.

To verify if an element x is in S, we compute $h_1(x), \ldots, h_k(x)$ and check whether *all* of those bits are set to 1. If the answer is no, then we *know* that x is *not* an element of S, i.e., there are no false negatives. Otherwise we *assume* that x is a member, although there is a distinct possibility that we are wrong and

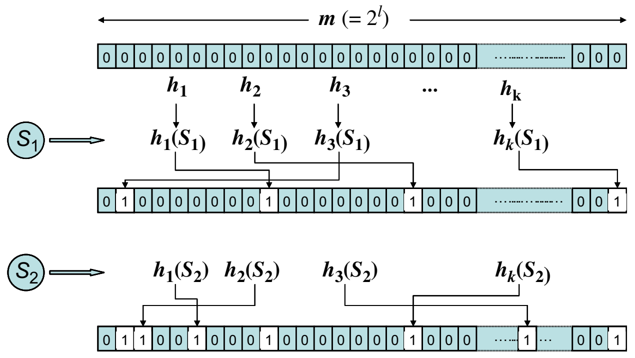

Fig. 2 – Insertion of elements into a *Bloom* filter.

the bits we checked were set by chance. Namely, it can be shown that the probability for a false positive is given by:

$$P_{FP} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k = (1-p)^k,$$

where p is the probability that a specific bit is still 0, after all the elements of S are hashed:

$$p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$

It is clear that the false positive rate depends on three factors: the size of the filter m, the number of elements n, and the number of hash functions k, and that those can be traded off. This optimization is not linear: increasing the number of hash functions increases the chance of finding a 0 bit for a non-element; however, reducing the number of hash functions increases the fraction of 0 bits in the filter. To illustrate this, Table 1 gives some example false positive rates for $m = 256$ and various k, and n.

An interesting property of Bloom filters is that although we cannot predict which bits will be set after n insertions, we can predict with great confidence how many of the bits will be set to 0 (or 1). Specifically, it can be shown (e.g. Brin et al., 1995) that the expected number of zero bits in the filter is strongly concentrated around its expectation $m(1 - 1/m)^{k|S|}$.

A similar result holds for the intersection of two filters (of the same size and hash functions). In other words, given two filters representing disjoint sets, we can rather confidently predict how many bits will match by chance. If the actual observation is outside a relatively narrow interval, we

Table 1 – Example false positive rates for $m = 256$

$m = 256$		k					
		2	4	6	8	12	16
m/n	16	0.0139	0.0024	0.0009	0.0006	0.0005	0.0007
	14	0.0178	0.0038	0.0018	0.0013	0.0013	0.0022
	12	0.0237	0.0065	0.0037	0.0032	0.0041	0.0076
	10	0.0330	0.0119	0.0085	0.0085	0.0137	0.0274
	8	0.0491	0.0241	0.0217	0.0257	0.0488	0.0986
	4	0.1553	0.1604	0.2209	0.3140	0.5438	0.7457

can conclude that it is statistically unlikely for that to happen by chance. Therefore, it is reasonable to assume that the two original sets had a certain degree of overlap.

The above observation is the basis for the work described in Roussev et al. (2006), which presents a tool (*md5bloom*) which allows general-purpose filter manipulation, as well as the described direct filter comparison with associated statistical interpretation. Thus, the filter becomes an aggregate hash function that enables efficient similarity comparison. In the specific uses presented by the authors the individual hashed components are derived from objects with well-known structure, such as a code library or a directory of files. For example, by using individual functions within a code library as the unit of hashing, the authors were able to easily identify versions with only 35% overlap despite the fact that the filters used had a high false positive rate of about 0.15 at 4.75 bits per element and four hash functions. The main shortcoming of the *md5bloom* tool is that, although it can work well for objects with known structure, it does not address the issue of determining suitable decomposition for an arbitrary object.

3. Design analysis

In this section, we present a generalized view of the context-based hashing approach presented in Section 2.2 and explore the design space.

Designing a context-based hashing scheme can be broken into several steps, each of which has more than one possible choice. The first step is determining the context, which brings up two questions:

- What is a suitable length for the context?
- What is a suitable context hash function?

ssdeep adopts most of its choices based on earlier work on spam filtering after which it is modeled Tridgell (2002). Specifically, it picks 7 as the length of the window over which the context function is evaluated and uses a version of the Adler32 checksum for the rolling hash function.

Are these good choices and how do we determine that?

Intuitively, a good hash function will produce relatively evenly sized chunks. The length of the context determines the number of evaluations of the context hash function – if the context has a length of c, this is equivalent to calculating the hash of the entire object c times. At the same time, a shorter context should produce shorter chunks, which would lengthen the composite hash.

First, let us try to determine the significance of the choice of hash function. Generally, there are two kinds of hash functions in common use today – polynomial (such as Adler32) and cryptographic (MD5 and the likes). The former is optimized for speed whereas the latter for collision-resistance. For our experiments, we picked MD5 and compared its performance to that of a simple polynomial hash usually referred to as *djb2* and defined as follows:

$$h_0 = 5381, \quad h_k = 33h_k + s_k, \ \text{for } k > 0.$$

where s_k denotes the kth character of the string being hashed.

Table 2 – Observed chunk sizes: *random* input, $t = 8$

Context	5		6		7		8		9	
Hash	*djb2*	*md5*	*djb2*	*md5*	*djb2*	*md5*	*djb2*	*md5*	*djb2*	*md5*
Mean	39	39	71	71	135	135	263	262	519	521
Median	29	30	52	52	97	96	184	185	363	362
Max	334	400	791	740	1715	1345	3341	2948	7155	4950
Std dev	31	31	63	64	128	127	257	255	511	515

We used a corpus of documents of various types obtained at random from a search engine (10 per topic). The documents were then manually opened to verify their validity. All duplicates were removed as were all documents less than 64 kB in size. We also used files containing pseudo-random numbers for reference.

First, we consider the choice of hash function over random input, in particular, we examine the length of the chunks. Table 2 presents a representative sample of the results for the *MD5* and *djb2* functions with a trigger value $t = 8$ and various lengths for context.

It is not difficult to observe that the behavior of the two functions is quite similar and that using a collision-resistant function is unnecessary in this case. This result is to be expected as the input provides all the randomness. The above results are quite close to our observations for some high-entropy data types, such as *JPEG* and *PDF*.

Let us now consider the case of other types of documents with lower entropy, such as MS Word documents. As Table 3 suggests, there are some differences in performance, however, those remain relatively small with the notable exception of extreme behavior, such as exceptionally long chunks. Therefore, sticking with the computationally affordable *djb2* function appears to be a reasonable choice.

Recall that, based on a trigger value of 8, our expectations are that a context would be discovered every $2^8 = 256$ bytes, on average. Therefore, the logical choice of context should be in the 7–8 range. We chose 7 as it shows less extreme behavior but also set a minimum chunk size equal to a quarter of the expected one, or 64 bytes. Thus, we expect a mean of 234 bytes for the chunk size for the tested *doc* files.

Next up is the choice of hash function for the chunk hashes. *ssdeep* uses the least significant 6 bits from a 32-bit *FNV* hash and this is referred to as *LSB6* hash. *FNV* is not a collision-resistant function and has some known collision issues, which are common among multiplicative functions. Therefore, the probabilistic rationale presented in the paper, which assumes a perfect hash, should be taken with a grain of salt. Generally, given that the performance benchmark is block-level crypto hashing, using *MD5* would seem an appropriate choice, especially for inputs with lower entropy which would present a serious problem for simple hashes.

Next, we need to decide on the composition and comparison semantics of the object hashes. In *ssdeep*, the composite hash is a string sequence with each character representing a chunk of the object. The comparison is done based on edit distance with different string operations – insert, delete, change, and swap – having different weights in the determination of the final score (1, 1, 3, and 5, respectively). This

Table 3 – Observed chunk sizes: *.doc* input, $t = 8$

Context	5		6		7		8		9	
Hash	*djb2*	*md5*	*djb2*	*md5*	*djb2*	*md5*	*djb2*	*md5*	*djb2*	*md5*
Mean	42	41	83	75	170	140	328	298	632	588
Median	30	29	53	49	93	81	173	154	329	290
Max	14,208	4102	14,208	11,608	65,536	29,901	486,950	360,555	504,890	361,128
Std dev	99	51	175	127	547	320	2201	1839	2609	2168

design is modeled on the algorithm used by the *trn* newsreader Tridgell (2002) and does not appear to be based on the specifics of the context-based hashing. Recall that contexts will pick randomly sized chunks so the preference of swaps and modifications over inserts and deletes does not have the same significance as in text editing.

A bigger issue here is the fact the hash produced by *ssdeep* is a *sequence* where order is considered significant. On the other hand, *md5bloom* uses an approach where order information is lost once the hashes are inserted into the filter. Maintaining sequence information carries non-trivial computational and storage costs. To illustrate, consider a 256-byte Bloom filter vs. a 256-byte LSB6 hash. The filter can accommodate 256 elements at 8 bits per element with four hash functions (Table 1) and will have a false positive rate of 0.024. The LSB6 hash will have at most $256/6 = 42$ elements (in reality, 32 with byte alignment) with a false positive rate of $1/64 = 0.015$. Further, the filter would be compared against another filter, which will improve its performance as shown in Roussev et al. (2006) and in other applications, such as hierarchical Bloom filters Shanmugasundaram et al. (2004). No such effect will be present in the LSB6 case. From a performance perspective, computing edit distance requires dynamic programming (quadratic complexity) whereas comparing filters is a straightforward linear scan with bitwise *AND* operations and table lookups.

The next step in the design process is to determine whether the composite hash will be of fixed or variable size. Fixed-size hashes have an obvious appeal – minimum storage requirements and simple management. However, they also have some scalability issues as they limit the ability of the hashing scheme to compare files of varying sizes.

This is illustrated by *ssdeep*, which sticks to a fixed-size final hash. Thus, the width of the trigger k in bits depends on the size of the file. So, if a file f_1 is at least twice as big as f_2 ($|f_1| > 2|f_2|$) then $k_1 > k_2$, which renders the hashes produced for the two file incomparable since they are effectively taken at different granularities. To somewhat alleviate the problem, *ssdeep* produces two hashes with trigger widths of k and $k + 1$. This postpones the day of reckoning but if $|f_1| > 4|f_2|$, hashes are again incompatible. Examples where this would be a serious limitation is looking for a file in an (uncompressed) archive or a disk image.

md5bloom has a very similar problem if we attempt to produce a composite hash which consists of a single filter. Recall that, in order to compare two filters, they must be of the same size and use the same hash functions. While it is possible shrink a filter in half at the expense of doubling the density, this trick is only practical if applied once or twice, otherwise it would either produce filters with too low bit per element count, or would require that filters be created very sparse, initially.

Fixed-size composite hashes produce another problem – they may force the reevaluation of the hash over the entire object if it turns out to be too short or too long. For example, running *ssdeep* on hundreds of files of different file types, reveals that the problem does not exist in high-entropy (compressed/encrypted) formats, such as *JPEG*, or *PDF* but for text/html files the hash is computed 1.3 times per file, on average, and almost two times for MS Word and Excel documents. Bloom filters have more flexibility and could be budgeted ahead of time for a certain level of variation but, in general, are not immune to the problem. Finally, fixed-size hashes have an obvious problem in dealing with stream data where object size is not known ahead of time and recalculation is not an option.

This analysis points to the need to devise a variable-sized hashing scheme that scales with the object size but also maintains a low relative overhead. One generic rule of thumb could be that, ideally, we should be able to compare the hashes of two large drives (500 GB) worth of data in main memory. On a modern machine with 2–4 GB of RAM, this points to an acceptable overhead in the 0.2–0.4% range.

Finally, we need to strike a proper balance between flexibility and standardization. By flexibility here we mean both the ability of the tool to adapt to the target and the option of allowing the investigator to fine-tune the tool to a specific need. At the same time, the hashes produced must adhere to some standardized scheme to ensure that hashes produced independently are guaranteed to be comparable and to yield a meaningful result. Existing cryptographic hashing tools fit that description, whereas newer tools like *ssdeep* and *md5bloom* do not.

4. *MRS* hashes

A multi-resolution similarity hash is, essentially, a collection of similarity hashes so we first focus on the basic construction of a variable-length similarity hash.

4.1. *Flat similarity hash*

Based on the analysis in the previous section and a lot of experimentation, we made the following design choices.

Context hash function: djb2. Since simple hashes work as well as crypto hashes in this case, we picked the simplest hash that is reputed to work well.

Context length: 7. While our tool retains the flexibility to use context of any length, standardization here is important as most users will have no basis for choosing a suitable number.

Chunk hash function: MD5. The reasoning here is twofold: chunks could be relatively long so having a collision-resistant function is worth the extra processing; also, it works well with our choice for hash composition, which is based on Bloom filters.

Hash composition: a set of Bloom filters. To enable universal comparison of filters we standardized them to be 256 bytes, 8 bits per element, and we use four hash functions. To obtain the four hashes, we take the MD5 chunk hash and split it into four 32-bit numbers and take the least significant 11 bits from each part.

The combined hash is generated as follows:

- A 32-bit *djb2* hash is computed on a sliding window of size 7. At each step, the least significant t bits of the hash (the trigger) are examined, and if they are all set to 1, a context discovery is declared; t is the essential parameter that distinguishes the different levels of resolution, as explained later in the section. For the lowest level 0, the default value is 8.
- Context discovery triggers the computation of the MD5 chunk hash between the previous context and the current one. (The beginning/end of an object is considered a context, mostly for convenience.)
- The chunk hash is split into four pieces and four corresponding 11-bit hashes are obtained and inserted into the current Bloom filter.
- If the number of elements in the current filter reaches the maximum allowed (256), a new filter is added at the end of the list and becomes the current one.

The hash consists of the concatenation of all the Bloom filters, preceded by their total count.

Comparison semantics: set-based (no ordering). Intuitively, our comparison semantics can be described as follows: given two files: A and B, where $|A| < |B|$ and their corresponding similarity hashes $h = h_1, h_2, \ldots, h_n$ and $g = g_1, g_2, \ldots, g_m$, how many among the filters $h_1, h_2, \ldots, h_n$ of A have a similar filter among $g_1, g_2, \ldots, g_m$?

More formally, let $z(f_1, f_2)$ be the similarity score of filters f_1 and f_2 $(0 \le z \le 1)$, and

$$\begin{aligned} z_1 &= \max\{z(h_1,g_1), z(h_1,g_2), \ldots, z(h_1,g_m)\}, \\ z_2 &= \max\{z(h_2,g_1), z(h_2,g_2), \ldots, z(h_2,g_m)\}, \\ &\cdots \\ z_n &= \max\{z(h_n,g_1), z(h_n,g_2), \ldots, z(h_n,g_m)\}, \end{aligned}$$

then $z(h, g) \equiv (z_1 + z_2 + \cdots + z_n)/n$.

We refer to the similarity between two filters as a *Z-score*, since it is derived from counting the number of zero bits in the filters and their inner product. It can be shown (e.g. Brin et al., 1995, Section 2.5) that the magnitude of the intersection of the original sets from which the filters are derived is proportional to the logarithm of $Z = (Z_1 + Z_2 - Z_{12})/(Z_1 Z_2)$, where Z_1 is the number of zero bits in the first filter, Z_2 – in the second filter, and Z_{12} in their inner product (bitwise AND).

The expected minimum is reached for $Z_1 = Z_2$ and $Z_{12} = 0$ (no common elements) so $Z_{\min} = 1/Z_2 = 1/2048$, for our specific case of 256 bytes.

Similarly, maximum is achieved whenever the two filters are identical so $Z_1 = Z_2 = Z_{12} \to Z_{\max} = 1/Z_1$. Thus, we can map the interval $[\log(1/2048), \log(1/Z_1)]$ interval to $[0, 1]$ to obtain a score. Interpretation of the score is discussed in Section 5.

4.2. Multi-resolution similarity hash

Let us now consider the expected behavior of the similarity hash. Generally, we would hope that the trigger value t can predict the average size of a chunk. In an ideal world, where data is uniformly random and the context hash function is perfect, we would expect that chunk sizes would *average* 2^k. Indeed, empirical tests show that, over a long random file the average chunk size does indeed tend to stay close to our expectations for the *djb2* function. The only caveat is that the observed standard deviation is fairly large – close to the value of the average itself. While it is difficult to make sweeping generalizations, we have found that, for our corpus of files, the actual average can vary within a factor of two (on either side) for things like Word & Excel documents, and gets closer to our expectations for compressed formats, such as PDF and JPEG. We should emphasize that our scheme does not depend on things being distributed in any particular way, however, for *very* sparse data the results may not be very helpful – indeed, it would be difficult to define any meaningful notion of similarity for such data.

Returning to our specific choices of parameters, we would expect that a single filter of 256 elements and a trigger $t = 8$ to cover approximately $256 \times 2^8 = 65{,}536$ bytes, or 64K. We have also observed that, with the exception of high-entropy data objects, most other objects tend to produce a large number of very small chunks (in the order of the context length). For the most part, this is due to the presence of blocks of zeros, or ones. From the point of view of comparison, that is not a problem as those will be effectively collapsed as they are mapped to the same bits in the Bloom filter. However, for performance reasons, we would like to avoid the computation of an MD5 hash on just a few bytes. Hence, we introduced a minimum chunk size, which is currently set to ¼ of the expected average chunk size, or 2^{k-2}. Thus, whenever a context is discovered, the next 2^{k-2} bytes are skipped as far as context hashing is concerned so the chunk is guaranteed a minimum size. Arguably, some small details may be lost, however, we have not observed any noticeable changes in effectiveness.

The overall design, as well as the chosen parameters implicitly set a lower threshold on the size of the objects for which the similarity hashing will be effective. Our goal was to support objects of 32K and higher, and our tests have shown that this scheme works well at the lower end. Although it is possible to increase the granularity by lowering t, we quickly reach the point of diminishing returns. Generally speaking, small files (under 32K) would likely be better served by string matching algorithms at a computational cost that should be similar to (if not lower then) our hashing scheme.

So far, the similarity hashing allows two objects of arbitrary sizes (subject to the 32K minimum) to be compared. For example, a 64K file could be compared to a 64 MB image of a flash card (or a 64 GB hard drive) to determine if remnants of it are still present. In the former case, we would expect to go through about 1024 filter comparisons, or 256 kB of trivial bit operations, which would be instantaneous on a modern machine. However, if we tried to compare two 64 MB images (256 MB of comparisons), the computation would not be so trivial, and with two 64 GB images (1 TB), it would be infeasible.

Fortunately, we can trade a lower granularity of the hashing for improved performance by simply varying the t parameter. For example, if we set t to 12, we would expect average chunk size around 2^{12} bytes and, therefore a single filter would cover $256 \times 2^{12} = 2^{20}$ bytes = 1 MB. Thus, comparing two 64 MB files at this granularity would need 64×64 filter comparison, or going over about 256 kB. Similarly, we could scale up t to 16, 20, 24, … to accommodate as a large a target as necessary.

So, what is the best value for t? The answer is that, unless the sizes of all possible targets are within a relatively narrow range, no single value of t will satisfy all our needs. Therefore, we generalize our similarity hashing to include multiple levels (or *resolutions*) of similarity hashing that vary according to the trigger parameter t. Thus, for larger files we store several similarity hashes for standardized value of t – currently, 8, 12, 16, 20, 24, 28, and 32. The latter is the upper limit for our context hash function, however, each filter will cover a full 1 TB of data.

By fixing the values of t ahead of time, as well as fixing the rest of the parameters, we guarantee that independently generated hashes will always be comparable and will yield meaningful results. Notice that the additional cost of maintaining multiple hashes drops by a factor of 16 for each subsequent level so storage overhead is not an issue.

5. Experimental results

We are finally ready to present some experiments with our prototype tool, called *mrshash*, that validate our work. In its current version, the tool takes four parameters: RAM allocation, comparison level, mode of comparison, and a list of files. Memory is allocated in several large chunks (one for each level) to minimize memory management overhead. There are two comparison modes: one-to-many and all-pairs. In the former, the first file in the list is compared to each one of the rest, whereas in the latter mode, all possible file pairs are considered.

The test data was obtained through a search engine by retrieving the top 10 documents on a random generic topic (e.g. ‘document’). We removed all duplicates, files below 64 kB, and verified the validity of each one of the remaining files by opening them. As explained below, this procedure allowed for a great variety of files but also netted some truly similar documents that we hope to capture automatically.

5.1. *Pair-wise file similarity*

In the trivial case of comparing a file with itself, the result will be bit-for-bit identical hashes provably yielding a Z-score of 1.

In our file-based experiments, we run two separate cases: “all-pairs”, and “each vs. half-directory”. In the former experiment, we simply compare all possible (unordered) pairs. In the latter case, we perform the following: in a directory containing a set of test files, we place half of the files in an uncompressed archive. Then, for each of the original files in the directory we generate the similarity score between the file and the archive. We expect the files that are in the archive to yield distinctly higher scores than the ones that are not. Due to the way the scaling of Z is performed, results close to zero should be considered noise. We performed the experiment for files of different types. We also varied the archive format (tar, zip, cat), but this had no observable effect so all presented results are based on the zip archive format.

doc (MS Word). The sample contained 355 files varying in size from 64 kB to 10 MB for a total of 298 MB of data. The reference archive included a random sample of 178 files (150 MB) and all similarity scores are computed for level 0.

For the all-pairs experiment, out of 62,835 pairs, only 57 yielded a score of 0.1 and above (<0.1%) and those were manually inspected for similarities. Out of the 18 pairs with score of 0.2 and above, there were two false positives (there was no obvious similarity among the files involves). Out of the 29 pairs with score below 0.2, there was one true positive and the rest were true negatives. In other words, if we were to choose 0.2 as a threshold value for classifying the pairs, we would end up with two false positives and one false negative.

The results for the second experiment are depicted in Fig. 3, where the top line represents the Z-scores of the files that are known to be in the archive (true positives, TPs), whereas the lower line represents the scores of the ones that are not part of the archive (true negatives, TNs).

Ideally, we would like to see no overlap in the above graph. In our test, we have two TNs and three TPs with score in the 0.1–0.13 range. Depending on how the cutoff line is drawn, we would end up with two false positives or three false negatives, respectively. If we pick a more conservative 0.2 threshold, we end up with no false positives and four false negatives. Note that we have removed from the data any TN files that have a similarity score with a TP file of 0.2 and above.

To illustrate the kind of similarity we found, we discovered that two of our *doc* files were also two versions of the XBRL 2.1 draft written four weeks apart (92 and 99 pages, respectively), two other files were closely related versions of a manual (53 and 54 pages), and so on.

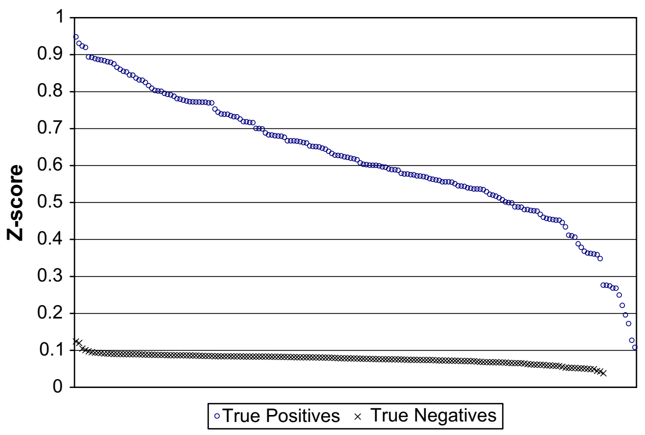

Fig. 3 – Z-Scores for true positives/negatives (doc).

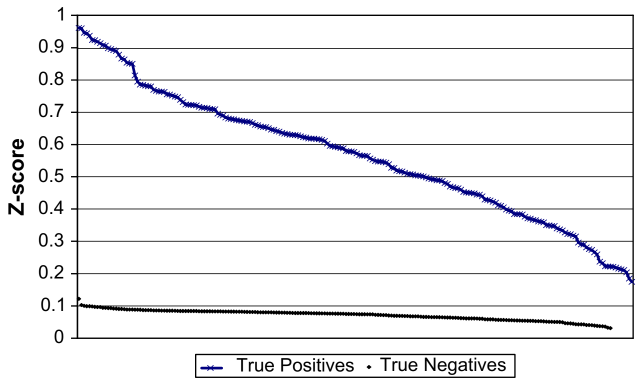

Fig. 4 – Z-Scores for true positives/negatives (xls).

xls (MS Excel). Our pool included 415 files, between 64 kB and 7 MB, total size 257 MB. For the all-pairs experiment, out of 85,905 pairs only 26 yielded a result of 0.1 and above and those were manually inspected. Using a threshold value of 0.2 (as before) yields one false positive and one false negative.

As examples of true positives, we found different drafts of the same environmental form, different work order of the same format, and files with common spreadsheets.

As Fig. 4 demonstrates there is a clear separation between true positives (above 0.17) and true negatives (below 0.12) for the second experiment.

JPEG. We tested on 737 files between 64 kB and 5 MB for a total of 121 MB of data. For the all-pairs experiment, we observed a total of 46 scores of 0.1 and above out of 273,370 pairs. We did not expect to find any similarity as all of the images were distinct. However, we discovered four true pairs with scores of 0.214, 0.166, 0.136, and 0.121, respectively, all of them among the top five Z-scores. In these cases, the images were distinct but they had a common border frame with the website's logo/credits, which showed up in the similarity measure.

For the second experiment, we found a clear separation of true positives and true negatives. As illustrated in Fig. 5, true positives had a Z-score above 0.17, whereas true negatives had scores below 0.1.

PDF (Adobe Acrobat). For technical reasons, our PDF sample was relatively small (59 files) so the details are not included here. In general, the files were picked to be different and the actual separation between true positive and true negatives for the second experiment was even clearer than the JPEG case: all true positives were above 0.2 and all true negatives below 0.03.

5.2. *Clustering*

We extended our small PDF set to include clusters of known similar files. Namely, we introduced two groups of files: nine chapters of pre-print version of an edited book and six papers from DFRWS'06. Note that both the book chapters and the papers were from different authors and their text content had virtually nothing in common. However, both groups of papers were formatted in specific ways by their respective publishers.

We ran our all-pairs test to see if any grouping will emerge. We found that the book chapters were closely related to each other forming two tight clusters of pair-wise similarity: 0.88–0.90, 0.66–0.69 and a bit of an outlier: 0.55. It is worth noting that the first cluster was formed around a relatively short file. The entire DFRWS cluster showed up as similarity measures in the 0.12–0.19 range.

All of the remaining of the pair measurements (2003 out of 2046) involving either comparisons with non-cluster files or comparisons across clusters below 0.09.

5.3. *Efficiency and performance*

Estimating the space efficiency is fairly easy – for a level 0 hash, we expect to use 256 bytes for every 64 kB of data or a ration of 1/256 ~ 0.004, which is within the intended range. For level 1 hashing the ratio is 1/4096 and the rest of the levels are in consequential as far as storage overhead is concerned.

We benchmarked the raw hashing performance of the current version of *mrshash* on a 1 GB file of random data on a dual-core Dell with a Pentium D @ 2.8 GHz, 4 GB of RAM, and an SATA II hard drive.

The observed performance is, essentially on par with block-level MD5 hashing (using *md5deep*) with 512 byte blocks – it took them both ~40 s to complete this task. *ssdeep* took ~50 s to perform the same task. We find this performance a pleasant surprise given the development stage of our tool. So far, we have not exploited the natural parallelism of the multi-resolution computation to utilize both processor cores. Furthermore, the Bloom filter comparisons are tailor-made to exploit the massive parallelism that is becoming available on the desktop environment under the guise of a graphics card. Recent developments, such as the NVIDIA CUDA platform, desire by manufacturers to enable general-purpose computation on all of its 128 processing elements.

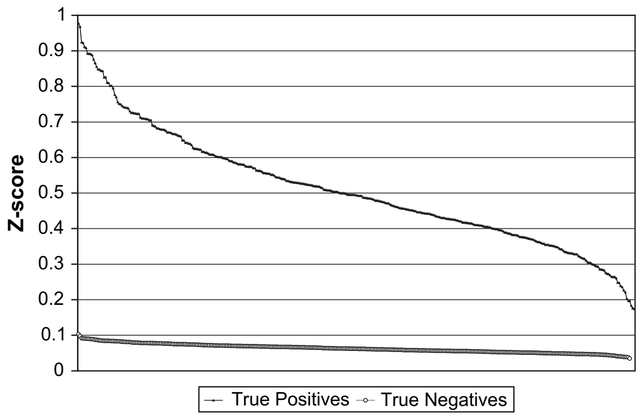

Fig. 5 – Z-Scores for true positives/negatives (jpg).

6. Conclusions

In this paper, we presented a new approach to computing similarity hashes for binary data. The hash uses a context discovery process to split the input into variable-sized chunks and hash them into a sequence of Bloom filters. Further, by varying the context discovery condition through a single parameter, an entire family of such hashes is generated at different levels of granularity. The resulting scheme has the following desirable properties:

- *Genericity* – it works on arbitrary pieces of binary data and is fairly sensitive to discovering similarities in the raw data.
- *Scalability* – it enables the comparison of objects that have several orders of magnitude difference in size. In principle, it could accommodate practically any object that can cross an investigator's desk. For example, it would be practical to search for the remnants of a file the size of 1 MB inside the disk image that is 100 GB.
- *Storage efficiency* – the hashes it produces, require no more than 0.5% in additional storage, thereby enabling the comparison of large drives in main memory.
- *Performance* – its raw hash generation rate is comparable to existing state-of-the-art block-level cryptographic hashing solutions.
- *Standardization* – while the basic approach offers a lot of possibilities, the background work presented here enables virtually all parameters to be standardized, which allows out-of-the-box use, as is the case with standard hashes.
- *Privacy preservation* – MRS hashing opens up the possibility of performing a limited initial inquiry into someone's data without obtaining a copy of it. Specifically, an MRS hash of the target would enable specific queries, such as, does the drive contain remnants of a particular photo, to be answered without opening a full-scale investigation, which can be treated as a fishing expedition by the courts.

At publication time, a beta version of the code will be available at http://roussev.net/mrshash.

7. Future work

We would like to perform much more testing on a bigger scale to explore whole-disk applications of this work and, in particular, applications like automated data classification.

We are very interested in taking advantage of parallel hardware that is becoming increasingly available in commodity computing systems to further improve the performance. In particular, we expect that the GPU can be successfully exploited to dramatically speed up the comparisons of pre-computed MRS hashes.

References

Bloom B. Space/time tradeoffs in hash coding with allowable errors. Commun ACM 1970;13(7):422–6.

Brin S, Davis J, Garcia-Molina H. Copy detection mechanisms for digital documents. In: Proceedings of the ACM SIGMOD annual conference, San Francisco, CA, May 1995.

Broder A, Mitzenmacher M. Network applications of bloom filters: a survey. Internet Math 2005;1(4):485–509.

Broder A, Glassman S, Manasse M, Zweig G. Syntactic clustering of the web. In: Proceedings of the 6th international world wide web conference, 1997. p. 393–404.

Charikar MS. Similarity estimation techniques from rounding algorithms. In: Proceedings of the 34th annual ACM symposium on theory of computing, 2002.

Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. In: Proceedings of the sixth symposium on operating system design and implementation, 2004.

Henziger M. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In: Proceedings of the 29th annual international ACM SIGIR conference on research and development on information retrieval, Seattle, 2006.

Karp R, Rabin M. Efficient randomized pattern-matching algorithms. IBM J Res Dev 1987;31(2):249–60.

Kornblum J. Identifying almost identical files using context triggered piecewise hashing. In: Proceedings of the 6th annual DFRWS, Lafayette, IN, Aug 2006.

Mitzenmacher M. Compressed bloom filters. IEEE/ACM Trans Netw October 2002;10(5):613–20.

Monostori K, Finkel R, Zaslavsky A, Hodasz G, Pataki M. Comparison of overlap detection techniques. In: Proceedings of the 2002 international conference on computational science (I), Amsterdam, The Netherlands, 2002. p. 51–60.

Roussev V, Richard III GG. Breaking the performance wall: the case for distributed digital forensics. In: Proceedings of the 2004 digital forensics research workshop (DFRWS 2004), Baltimore, MD.

Roussev V, Chen Y, Bourg T, Richard III GG. *md5bloom*: forensic filesystem hashing revisited. In: Proceedings of the 6th annual DFRWS, Lafayette, IN, Aug 2006.

Shanmugasundaram K, Bronnimann H, Memon N. Payload attribution via hierarchical bloom filters. In: Proceedings of the ACM symposium on communication and computer security (CCS'04), 2004.

Shivakumar N, Garcia-Molina H. SCAM: a copy detection mechanism for digital documents. In: Proceedings of the international conference on theory and practice of digital libraries, June 1995.

Shivakumar N, Garcia-Molina H. Building a scalable and accurate copy detection mechanism. In: Proceedings of the ACM conference on digital libraries, March 1996. p. 160–8.

Shivakumar N, Garcia-Molina H. Finding near-replicas of documents on the web. In: Proceedings of the workshop on web databases, March 1998. p. 204–12.

Tridgell A. Spamsum README. <http://samba.org/ftp/unpacked/junkcode/spamsum/README>; 2002.