Scalable Data Correlation

Vassil Roussev
Department of Computer Science
University of New Orleans
New Orleans, Louisiana 70148, USA

vassil@cs.uno.edu

Chapter 1

SCALABLE DATA CORRELATION

Managing TB-scale investigations with similarity digests

Abstract

The fast capacity growth of cheap storage presents an ever-escalating problem for forensic investigations as currently employed forensic technologies are not designed to scale to the degree necessary to meet the challenge.

In this work, we present an approach which seeks to scale up the process of finding related digital artifacts across large data sets by employing an advanced version of our similarity digest tool `sdhash`. Specifically, we demonstrate that the digest generation process can be performed at rates *exceeding* those of crypto hashes on commodity multi-core systems. Digest comparison rates allow us to query the digest of a 1TB target for the (trace) presence of a small file in less than one second with very high precision and recall rates.

Keywords: Similarity digest, data fingerprinting, hashing, digital forensics, sdhash

1. Introduction

It is typically the case that the overwhelming majority of the data collected before the start of an investigation is, in fact, completely irrelevant to the subject of the inquiry. Thus, the first step in any investigation of non-trivial size is to *automatically* remove from consideration as much irrelevant data as possible. Conversely, an investigation may have a very specific focus where locating known relevant artifacts (images, documents, etc.) can dramatically speedup the process. In both cases, the general approach is to build reference sets of known data and than have the forensic tool automatically search the targets to filter artifacts in/out. Since the common unit of interest is a file, this is often referred to as *known file filtering* (KFF).

The standard approach to accomplish KFF is to apply cryptographic hashes, such as MD5 and SHA1, and build tables of known hashes (such as OS and application installation files). For example, NIST maintains the National Software Reference Library (NSRL), and numerous vendors provide even more extensive collections. One methodological problem is that, to find a match, the two objects need to be identical down to the last bit, which makes the matching process very fragile. By extension, it is becoming increasingly infeasible to keep the reference sets current as popular software updates itself almost daily. Similarly, user artifacts also get modified and tracking versions across multiple targets and network streams can be very important in a complex investigation.

Specifically, our work targets three basic scenarios:

- *Identification of embedded/trace evidence.* Given a data object (file, disk block, network packet) we want to be able to identify (traces of) its presence inside a bigger object of arbitrary size, such as a disk image, RAM snapshot, or network capture.
- *Identification of artifact versions.* Given a large set of artifacts (files), identify those that are versions of a set of reference artifacts. The target objects could be executable code (versions of known software), or user-created data objects (documents).
- *Cross-target correlation.* Identify and correlate different representations–on-disk, in-RAM, network flow–of the same/similar object. This would allow automated annotation RAM and network captures without the deep (and costly) parsing and reconstruction that current approaches require.

In addition to supporting the above scenarios we also place primary importance on accuracy and performance requirements:

- *High precision and recall rates.* Given the volume of information, we need near-certainty with respect to the results obtained by the tool. For example, 1% false positive rate, which under many scenarios is excellent, applied to ten million files on a forensic image would result in

100,000 false positives. Manual examination of that amount of data can easily negate any automation benefits.

- *Scalable throughput.* The only hope of keeping up with the growth in data volume is to employ ever larger computational resources to match it. Therefore, a tool should be able to scale out horizontally and seemlessly take advantage of all hardware resources that are made available to it. In our way of thinking, scalability should be measured in server racks and cabinets, not CPU cores. If a forensic lab has a large and urgent case, it should be possible to dedicate its entire data center, if necessary, to get results on time.

- *Stream-oriented processing.* Performance-wise, commodity hard disks are quickly becoming the new tapes: they have huge and growing capacities, but the only meaningful way to sustain high throughput is by sequential access. On other words, they need to be read end-to-end (or at least in large chunks). The current approach of file-centric data access via the file system's API is convenient for developers but is completely out of sync with the performance nightmare of randomized disk access.

2. Background: similarity digests and `sdhash`

The main motivation behind the concept of similarity digests is the observation that data fingerprinting methods based on random polynomials, a technique pioneered by Rabin [15], do not work all that well on real data. Specifically, it is difficult to guarantee uniform coverage and control the false positives. The Related Work section describes some of the more notable research efforts made to remedy these shortcomings. For the rest of this section, we briefly outline the original similarity digest design and its implementation; a more detailed description is provided in [21].

2.1 Statistically improbable features

In designing `sdhash`, we took an approach that is inspired, in part, by information retrieval concepts, such as *statistically improbable phrases* (SIP). The rationale is to pick a set of features to represent an object that are statistically rare in the general population, and later compare them with those of other object. If we find that a number of these features are in common, then we can use that as an indication of a degree of correlation. The more features in common, the higher the correlation; if all the features match, then the two original objects are likely identical.

The main challenge is to translate this idea from text to binary data and to implement it efficiently. We start by defining a feature as a 64-byte sequence (string) and consider all feature in object (such as a file) as candidates. The first thing to note is that we are not in a position to either collect, store, or query empirical probabilities for all possible 64-byte features. Instead, for each feature, we calculate a normalized Shannon entropy measure H_{norm} as follows:

The result is that we have placed all features in 1,000 classes of equivalence. Then, using an empirical probability distribution from a representative set, we can assign them a *precedence rank* R_{prec}, such that features with the lowest probability have the highest rank.

Next, we define a *popularity rank* R_{pop}, which describes how the R_{prec} of a feature relates to those of its neighbors. To calculate it, for every (sliding) window of W consecutive features, we find the leftmost feature with the lowest precedence rank and increment its R_{pop} by one (evidently, $R_{pop} \leq W$). Intuitively, the rarer the feature, the more likely it is that it will have a higher score than its neighbors.

Finally, we make a sweep across all feature and select all the ones for which $R_{pop} \geq t$, where $0 < t \leq W$ is a threshold parameter. During this process, we also perform an additional filtering step which simply ignores all features for which $H_{norm} \leq 100$. This is based on our empirical studies showing that features with such low entropy trigger the vast majority of false positives.

2.2 Similarity Digest Bloom Filter (`sdbf`)

Once the features have been selected, each one is hashed using SHA1, the result is split into five subhashes (which we treat as independent hash functions) and is placed in a Bloom filter [1], [10] with a size of 256 bytes. (Recall that a Bloom filter is a probabilistic set data structure which offers a compressed representation in exchange for a controllable false positive rate.) As soon as a filter reaches 128 elements (features), we declare it full and create new one to accommodate further features. This process continues until all features are accommodated and the resulting sequence of filters is the similarity digest of the original object. We refer to this digest representation as `sdbf`, and the tool used to generate and compare the digests as `sdhash` (**s**imilarity **d**igest **hash**).

Evidently, there is a large number of tunable parameters that can provide different trade offs between granularity, compression, and accuracy. However, it is also important to have a standard set of paramaters so that independently generated digests are compatible. Based on exhaustive testing, we chose $W = 64$, $t = 16$ and the already mentioned parameters for our reference implementation. In practical terms, the length of each digest is in the order of 3% of the size of the original data. Thus, each filter represents, on average, 7-8KB chunk of the original artifact.

The base operation for comparing digests is the comparison of two constituent filters. Given two arbitrary filters, we can analytically predict their dot product (i.e., how many bits the two have in common) due to chance. Beyond that, the probability that the two filters have common elements rises linearly and allows us to define a similarity measure $D(\cdot)$, which yields a number between 0 and 1.

To compare two digests $F = f_1 f_2 \ldots f_n$ and $G = g_1 g_2 \ldots g_m$, $n \leq m$, we define the *similarity distance* $SD(F, G)$ as:

$$SD(F, G) = \frac{1}{N} \sum_{i=1}^{n} \max_{j=1..m} D(f_i, g_j)$$

Intuitively, for each filter of the shorter digest F we find the best match in G, and we average the maxima to produce the final result. In the special case where F has a single filter, the result would be the best match, which makes perfect sense. In the case where the two digests have comparable, this definition will seek to estimate their highest similarity.

3. Block-aligned similarity digests (`sdbf-dd`)

The main goal of this work is to take the basic algorithm, which has been shown to be robust [19], and make it more scalable by parallelizing it. As a reference point, the optimized serial version of `sdhash`(1.3) is capable of generating a digest at the rate of 27MB/s on a 3GHz Intel Xeon processor. It takes 3ms to query (the digest of) a 100MB target for the presense of a small file (which has a single filter as a digest). Thus, the small-file query rate is approximately 33MB per millisecond.

It is clear that the original generation algorithm is sequential in nature and its design was primarily targeted at file-to-file comparisons. To parallelize it, we need to move away from the chain dependencies among `sdbf` component filters in order to allow concurrent generation.

As it turns out, for large targets, we can move to a block-aligned version of the `sdbf`, which we refer to as `sdbf-dd`. The idea is relatively simple–split the target into blocks of fixed size and run the signature generation in block-parallel fashion. To make this efficient, we also fix the output to one filter such that each block in the original data maps to exactly one filter in the digest. This design has the added benefit that we can quickly map any query matches to disk blocks and do followup processing.

Given that in the original `sdhash` design a filter typically represents 7-8KB of data, 8KiB is de facto a minimum block size. There are two additional considerations that factor into the decision process: larger block size leads to smaller signatures and faster comparisons, however, they also make it harder to maintain compatibility with the sequential implementation.

To balance these conflicting requirements, the block size was set to 16KiB and the maximum number of features per filter was increased to 192 (from 128). Under average conditions, we would expect that a 16KiB block would produce upwards of 256 features that clear the standard threshold of $t = 16$. Therefore, we select the features starting with the most popular and go down successively until either fill up the filter , or there are no more features above the threshold left.

As our subsequent experiments show, it was also necessary to increase the maximum number of features per filter for the `sdbf` version of the code from 128 to 160. This brings two improvements: a) it increases the average data chunk covered by a single filter from the 7-8KB range to the 9-10KB, thereby shortening the digest and reducing comparison times; b) allowes for more features to be accumulated thereby enabling better compatibility with the `sdbf-dd` version.

Under the above parameters, the in-memory `sdhash-dd` representation requires a 1.6% of the size of original data (260 bytes for every 16,384 bytes of data). Thus, 1TB of data can be represented by a 16GB digest, which easily fits in RAM. The new `sdhash` representation with variable chunk size and 160 elements per Bloom filter requires, on average, 2.7% of the size of the original file (258 bytes per 9,560 bytes of data). The on-disk representations of both versions are base64-encoded, which incurs a 33% overhead, but this is readily recoverable with standard compression (e.g., for archival purposes).

4. Evaluation

In this section, we measure the throughput and accuracy of our new approach. All measurements are based on `sdhash` version 2.0, which is in an active development stage and will be available at the time of publication at `http://roussev.net/sdhash`.

4.1 Throughput

All performance tests were run on a commodity Dell PowerEdge R710 server with two six-core Intel Xeon CPUs @2.93GHz with hyper-threading, for a total of 24 hardware-supported threads. The host is equipped with 72GiB of RAM @800MHz and has hardware-supported RAID 10 with benchmarked sequential throughput of 300MB/s. All presented numbers are stable averages over multiple runs (unless otherwise noted).

4.1.1 Block-aligned digest generation (`sdhash-dd`).

In this experiment we quantify the scalability of the parallelized version as a function of the number of threads employed. First, we hash a memory-resident 10G target to eliminate the effects of I/O; the results are shown in Table 1.

Table 1. `sdhash` generation performance on 10GB target

Threads	**Time** (sec)	**Throughput** (MB/s)	**Speedup**
1	374.0	26.74	1.00
4	93.0	107.53	4.02
8	53.0	188.68	7.06
12	44.5	224.72	8.40
24	27.0	370.37	13.85

It is evident that the process shows excellent scalability, with the 24-threaded version achieving 370MB/s of throughput. For reference, a SHA1 job on the same target produced throughput of

333MB/s. It is also clear that, given enough hardware parallelism, the `sdhash-dd` computation is I/O-bound.

We also applied `sdhash-dd` with 24 threads to a 100GB target; the computation took 475 seconds (with cold cache) for an effective throughput of **210**MB/s.

4.1.2 File-parallel digest generation (`sdhash`).

To scale up the performance for the `sdhash` version of the code, which is still the preferred digest algorithm for most typical files, we implemented file-parallel hashing. To quantify the performance gain, we used 40,000 file sample from the Govdocs1 corpus `http://digitalcorpora.org/corpora/files` [7], with a total size of 26,750M. Table 1.4.1.2 summarizes the results (300 files were not hashed as their size was below 512 bytes). It is notable that all files were cached, so the performance of the I/O system is not a factor.

Table 2. File-parallel `sdhash` generation performance on 39,700 files (26.75GB)

Threads	**Time** (sec)	**Throughput** (MB/s)	**Speedup**
1	920	29.08	1.00
4	277	96.57	3.32
8	187	143.05	4.92
12	144	185.76	6.39
24	129	207.36	7.13

The task distribution in the current implementation is not optimal–the files provided on the command line are deterministically split among the threads with no regard for their size. Thus, an "unlucky" thread could be tasked to digest a well-above average amount of data and, therefore, slowdown the overall completion time. Another impediment to further speedup is that many of the files are small, which increases overhead and reduces concurrency due to extra I/O and memory-management operations.

To better understand these effects, we created a synthetic set of 10,000 files of 500KB each and reran the experiment; the results are shown in Table 4.

Table 3. File-parallel `sdhash` generation performance on 10,000 files (5GB)

Threads	**Time** (sec)	**Throughput** (MB/s)	**Speedup**
1	177	28.25	1.00
4	50	100.00	3.54
8	30	166.67	5.90
12	24	208.33	7.38
24	19	263.16	9.32

We also applied `sdhash` with 24 threads to each of the 147,430 file on the mounted 100GB target from the previous experiment. With cold cash, the computation took 1,621 seconds for an effective throughput of **57**MB/s.

Figure 1 summarizes the parallelization speedup of digest generation achieved in each of the three cases discussed–`sdhash-dd`, `sdhash` on real files, and `sdhash` on optimally balanced workload.

Recall that although the experimental server supports 24 hardware threads, there are only 12 physical CPU cores. Running two threads on the same core only yiels benefits if the threads have complimentary workloads, or a thread is stalled. In our case, the threads are competing for the same CPU units so we can expect speedup to go up only modestly when going from 12 to 24 threads. The graph clearly illustrates this behavior, as well as the fact that `sdhash` scales to a lesser degree. The latter is a likely a function of higher memory management overhead–multiple allocations/deallocations per file vs. on large allocation for `sdhash-dd`.

4.1.3 Digest comparison rates.

Table 4 shows the digest comparisons rates at various levels of concurrency. The main measure we use is the number of Bloom filter (BF) comparisons per millisecond; based on this measure we can calculate the expected execution time different configurations. As a benchmark, we use the comparison rate for small files ($\leq$ 16KiB) which have digests of one BF. Based on the 12-/24-thread rate, we expect to be able to search (the digest of) a target for a small file at the rate of $58,000 \times 16,384$ bytes = 950MB per millisecond, or **950GB/sec**.

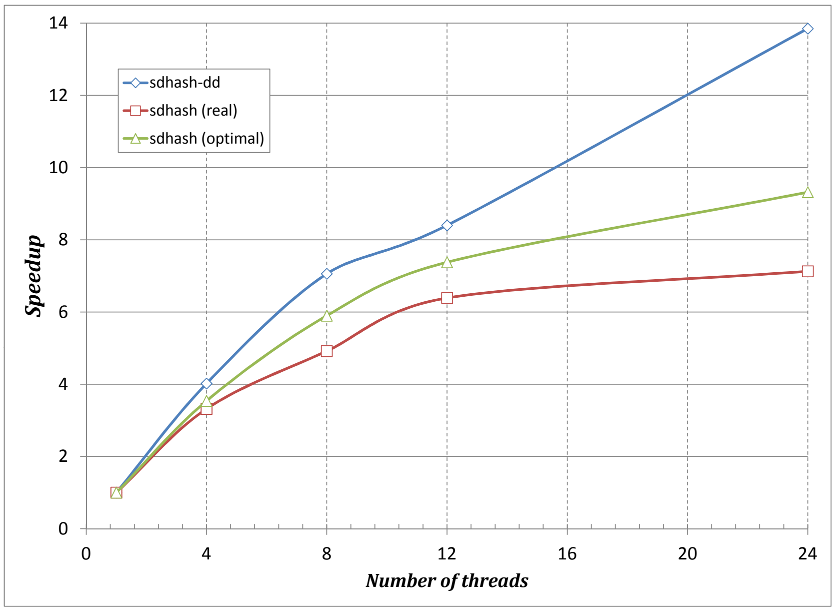

Figure 1. `sdhash` speedup summary

Table 4. `sdhash` digest comparison performance: 372.9 mln Bloom filter (BF) comparisons

Threads	**Time** (ms)	**Throughput** (BF/ms)	**Speedup**
1	19,600	19,027	1.00
4	9,800	38,054	2.00
8	8,900	41,902	2.20
12	6,500	57,373	3.02
24	6,400	58,270	3.06

Relative to the original rate of 33MB/ms, even our new single-threaded version offers speedup of 9.73 times (321MB/ms) by almost completely eliminating floating point operations, and optimizing the dominant case of BF comparisons yielding zero. The 12-/24-threaded versions are 28.78 times faster (950MB/ms). The comparison operation offers an extreme example of how the execution of two threads on a single core offers no advantage as the threads compete for the exact same functional units.

Note that the above rates are *inclusive* of the time it takes to load the digests. To gain a better estimate, we compared the `sdbf-dd` digest of a 100GB target (6.1 million BF) with a set of 300 small files extracted from the image with a total BF count of 908. The total execution time using the 24-threaded version completed in 79 seconds, of which 15 were spend loading and setting up the in-memory representation of the digests. Thus, the "pure" comparison rate is 86,600 BF/ms, or **1.4 TB/sec** for the small-file case.

4.2 Accuracy

We have recently performed a fairly detailed study [19] of the base `sdhash` algorithm and its performance on controlled and real-world data. Therefore, we limit our discussion to validating the new parameters chosen for it, as well quantifying the accuracy of `sdbf` vs. `sdbf-dd` comparisons.

For this experiment we use synthetic targets generated from random data to precisely control the content of the query and the target. The experiment was performed as follows:

For each query size, we independently generate a 100MB target and 10,000 control files of the given query size. Then, we extract 10,000 samples of the query size from the target. The target is hashed using `sdhash-dd` while the remaining 20,000 files are hashed with `sdhash`and then compared with the target; a result of zero is a negative, whereas a non-zero result is counted as a positive. Ideally, we expect all samples to match, while none of the controls to do so.

Table 5 shows the results for query sizes of 1,000 to 3,800 bytes. Our complete experiment investigated query sizes of up to 64KB with the observed behavior essentially identical to the one described by query sizes 2,000 and up in the table. For the sake of brevity, we omit them. The results clearly demonstrate that for queries of 2KB and up, which are relevant for block storage device, the tool produces near-perfect results. On the low end, which is relevent to investigating network traffic, the FP rate starts out high (due to the small number of features selected) but quickly drops below 1% for queries of 1,400 byte and up. This is not a coincidence but the result

Table 5. False (FP) and true (TP) positive rates for `sdhash` queries (1,000-3,800 bytes) vs. `sdhash-dd` target

Query size	**FP rate**	**TP rate**	**Query size**	**FP rate**	**TP rate**
1,000	0.1906	1.000	2,000	0.0006	0.997
1,100	0.0964	1.000	2,200	0.0005	1.000
1,200	0.0465	1.000	2,400	0.0001	1.000
1,300	0.0190	1.000	2,600	0.0001	0.997
1,400	0.0098	1.000	2,800	0.0000	1.000
1,500	0.0058	1.000	3,000	0.0000	0.999
1,600	0.0029	0.999	3,200	0.0000	0.998
1,700	0.0023	0.999	3,400	0.0000	0.998
1,800	0.0013	0.999	3,600	0.0000	1.000
1,900	0.0010	0.998	3,800	0.0000	0.998

of performance tuning of the parameters referenced earlier. It is based on the observation that more than 85% of Internet traffic by volume is based on packets in the 1,400 to 1,500 byte range [5]; that is, long data transfers dominate. In a network investigative scenario, the FP rate can be easily managed by searching for multiple packets of the same source; for example if `sdhash` flags three independent packets as belonging to a file, the empirical probability that all three are wrong is only $0.19^3 = 0.0069$.

5. Related Work

The basic idea of generating a more flexible data fingerprint dates back to 1981 and was put forward by Michael Rabin in his seminal work [15]. Since then, considerable research has gone into producing more complex versions; however, the basic idea has carried over with relatively small variations. We limit the discussion to a brief summary of the essential ideas–a more detailed survey of hashing and fingerprinting techniques can be found in [22].

5.1 Rabin Fingerprinting

Rabin's scheme is based on random polynomials and its original purpose was "to produce a very simple real-time string matching algorithm and a procedure for securing files against unauthorized changes." [15]. A Rabin fingerprint can be viewed as a checksum with low, quantifiable collision probabilities that can be used to efficiently detect identical objects. In the 1990s, there was a renewed interest in Rabin's work in the context of finding similar objects, with an emphasis on text. To name a few, Manber [9] created the `sif` tool for Unix to quantify similarities among texts files; Sergey Brin, in his pre-Google years, used it in a copy-detection scheme [2]; Broder applied it to find syntactic similarities among web pages [3].

The basic idea, alternatively called anchoring, chunking, or shingling, is to use a sliding Rabin fingerprint, over a fixed-size window, as the means to split up the data into pieces. For every window of size w, we compute the hash h, divide it by a chosen constant c, and compare the remainder to another constant m. If the two are equal (i.e., $m \equiv h \bmod c$), we declare the beginning of a chunk (an anchor) and we slide the window by one position and continue the process until we reach the end of the data. For convenience, the value of c is typically a power of two ($c = 2^k$) and m can be any fixed number between zero and $c - 1$. Once we have determined our baseline anchoring, we can use it a number of ways to select characteristic features: a) choose the chunks in between anchors as our features; b) start at the anchor position and pick the following l number of bytes; c) use multiple, nested features.

Note that, while shingling schemes pick a randomized sample of features, they are deterministic and, given the same input, produce the exact same features. Further, they are locally sensitive in that the determination of an anchor point depends only on the previous w bytes of input, where w could be as small as a few bytes. We can use this property to solve our fragility problem in traditional file and block-based hashing. Consider two versions of the same document: we can view one of them as derived from the other by means of inserting and deleting characters. For example, converting an HTML page to plain text will remove all the HTML tags. Clearly, this would modify a number of features but we would expect chunks of unformatted text to remain intact and to produce some of the original features, allowing us to automatically correlate the versions. For the actual feature comparison, we store the hashes of the selected features and use them as a space-efficient representation of the object's fingerprint.

5.2 Similarity Hashing

Kornblum [8] was among the first to propose the use of a generic fuzzy hash scheme for forensic purposes and developed the `ssdeep` tool. It generates string hashes of up to 80 bytes that are the concatenation of 6-bit piece-wise hashes. The comparison is then performed using edit distance. While `ssdeep` has gained some popularity, the fixed-size hash it produces quickly loses granularity, and can only be expected to work for relatively small files of similar sizes.

Around the same time Roussev et al. [17] proposed a scheme which uses partial knowledge of the internal object structure and Bloom filters to derive a similarity scheme. This was followed by a Rabin-style multi-resolution scheme [18] that attempts to balance performance and accuracy requirements by keeping hashes at several resolutions.

Outside forensics, Pucha et al. [14] proposed an interesting scheme for the identification of similar files on a peer-to-peer network. It is targeted at identifying large-scale similarity (e.g. same movie in different languages) that can be used to offer alternatives to user to download.

5.3 Summary

The randomized model of Rabin fingerprinting works well *on average* but on real data suffers from problems in coverage and false positive rates. Both of these can be traced to the fact that underlying data can have significant variation in information content. As a result, feature size/distribution can vary widely making the fingerprint coverage highly skewed. Similarly, low-entropy features produce abnormally high false positive rates making the fingerprint an unreliable basis for comparison.

Active research in payload attribution systems has produced ever more complicated versions of the Rabin fingerprints–[24], [25], [6], [13]–with the goal of ensuring even coverage. These techniques manage the feature selection process so that big gaps or clusters are avoided. Yet, none of these methods consider false positives due to weak (non-identifying) features. It is important to recognize that coverage and false positives are inherently connected–selecting weak features to improve coverage directly increases the risk of false positive results.

6. Conclusions

In this work, we presented presented a practical approach to correlating large forensic data sets at the byte-stream level. This allows for content filtering to be applied on a wide scale and be incorporated early in the forensic process. For example, current best practices dictate that creating a copy of the original media, such as a hard drive, is the first step in any formal inquiry. Typically, no useful processing is done during this multi-hour process and, at at the end of it, the investigator known nothing more than they did in the beginning. Our work demonstrate that we can generate a similarity digest at line speed so that, at the end of the cloning process we can immediately perform content queries. In fact, it is entirely feasible to take this a step further and to start matching hashed data against reference sets *while* the hashing/cloning process does on.

Our results show that, on an $8,500 commodity rack server, we can sustain a hash generation rate of up to 370MB/s on a 24-threaded system. We can search for the content of a small query file (16KiB) in a reference set at the rate of 1.4TB/s. For query objects above 2KB we obtain near-perfect true and false positive rates.

In our view, the capabilities that we present here can qualitatively change the scope and efficiency of digital forensic investigations. We also view them as just the opening bid to massively scale data correlation and, hopefully, spur similar efforts to dramatically speed up other types of automated forensic processing to meet the challanges of big data.

References

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

[2] S. Brin, J. Davis, and H. Garcia-Molina. ”Copy Detection Mechanisms for Digital Documents”, Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, May 1995, San Jose, CA.

[3] A. Broder, S. Glassman, M. Manasse, and G. Zweig, ”Syntactic Clustering of the Web”. In Proceedings of the 6th International World Wide Web Conference, pp. 393-404, 1997.

[4] A. Broder, M. Mitzenmatcher. ”Network applications of Bloom filters: a survey”. In Annual Allerton Conference on Communication, Control, and Computing, Urbana-Champaign, Illinois, USA, October 2002.

[5] The Cooperative Association for Internet Data Analysis, ”Packet size distribution comparison between Internet links in 1998 and 2008”, `http://www.caida.org/research/traffic-analysis/pkt_size_distribution/graphs.xml`.

[6] C. Y. Cho, S. Y. Lee, C. P. Tan, and Y. T. Tan. ”Network forensics on packet fingerprints”, 21st IFIP Information Security Conference (SEC 2006), Karlstad, Sweden, 2006.

[7] Garfinkel S., Farrell P., Roussev V., Dinolt G., ”Bringing Science to Digital Forensics with Standardized Forensic Corpora”, Proceedings of the Ninth Annual DFRWS Conference, pp.2-11, Aug 2009, Montreal, Canada.

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

[9] U. Manber. "Finding similar files in a large file system", In Proceedings of the USENIX Winter 1994 Technical Conference, pages 1-10, San Fransisco, CA, USA, 1994.

[10] M. Mitzenmacher, "Compressed Bloom Filters", IEEE/ACM Transactions on Networks, 10:5, pp. 613-620, October 2002.

[11] NPS Corpus, http://digitalcorpora.org/corpora/disk-images/.

[12] NIST. National Software Reference Library, http://www.nsrl.nist.gov.

[13] M. Ponec, P, Giura, H. Brnnimann, J. Wein, "Highly Efficient Techniques for Network Forensics", In Proceedings of the 14th ACM Conference on Computer and Communications Security, 2007, Alexandria, Virginia.

[14] H. Pucha, D. Andersen, M. Kaminsky, "Exploiting Similarity for Multi-Source Downloads using File Handprints". In Proceedings of the Forth USENIX NSDI, Cambridge, MA. Apr, 2007.

[15] M. O. Rabin. "Fingerprinting by random polynomials", Technical report 15-81, Harvard University, 1981.

[16] S. Rhea, K. Liang, and E. Brewer. "Value-based web caching", In Proceedings of the Twelfth International World Wide Web Conference, May 2003.

[17] V. Roussev, Y. Chen, T. Bourg, G.G. Richard III, "md5bloom: Forensic Filesystem Hashing Revisited", In Proceedings of the 6th Annual DFRWS Conference (DFRWS'06). West Lafayette, IN. Aug 2006.

[18] V. Roussev, G. G. Richard III, L. Marziale, "Multi-resolution Similarity Hashing", In Proceedings of the 7th Annual DFRWS Conference (DFRWS'07), Pittsburgh, PA. Aug 2007, doi:10.1016/j.diin.2007.06.011.

[19] V. Roussev, "An Evaluation of Forensics Similarity Hashes", Proceedings of the Eleventh Annual DFRWS Conference, pp.34-41, Aug 2011, New Orleans, LA.

[20] V. Roussev, G.G. Richard III, L. Marziale, "Classprints: Class-aware Similarity Hashes", In Ray, I., Shenoi, S. (eds.), Research Advances in Digital Forensics IV. Springer, 2008. ISBN: 978-0-387-84926-3.

[21] V. Roussev, "Data Fingerprinting with Similarity Digests", In Chow, K.; Shenoi, S. (Eds.), Research Advances in Digital Forensics VI, pp. 207-226, Springer, 2010.

[22] V. Roussev, "Hashing and Data Fingerprinting in Digital Forensics," IEEE Security and Privacy, vol. 7, no. 2, pp. 49-55, Mar./Apr. 2009, doi:10.1109/MSP.2009.40.

[23] V. Roussev, "Building a Better Similarity Trap with Statistically Improbable Features", HICSS, pp.1-10, 42nd Hawaii International Conference on System Sciences, 2009.

[24] S. Schleimer, D. S. Wilkerson, and A. Aiken. "Winnowing: local algorithms for document fingerprinting", In SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on management of data, pages 76-85, New York, NY, USA, 2003. ACM Press.

[25] K. Shanmugasundaram, H. Brnnimann, and N. Memon. "Payload Attribution via Hierarchical Bloom Filters", In Proceedings of the 11th ACM conference on computer and communications security, 2004.