Chapter 2

MANAGING TERABYTE-SCALE INVESTIGATIONS WITH SIMILARITY DIGESTS

Vassil Roussev

Abstract The relentless increase in storage capacity and decrease in storage cost present an escalating challenge for digital forensic investigations – current forensic technologies are not designed to scale to the degree necessary to process the ever increasing volumes of digital evidence. This paper describes a similarity-digest-based approach that scales up the task of finding related digital artifacts in massive data sets. The results show that digests can be generated at rates exceeding those of cryptographic hashes on commodity multi-core computing systems. Also, the querying of the digest of a large (1 TB) target for the (trace) presence of a small file can be completed in less than one second with very high precision and recall rates.

Keywords: Similarity digests, data fingerprinting, hashing

1. Introduction

The overwhelming majority of the data collected in a digital forensic investigation is often irrelevant to the subject of the inquiry. Thus, the first step in most investigations is to automatically eliminate as much irrelevant data as possible. Conversely, many investigations may have a very specific focus on locating known relevant artifacts such as images or documents. Therefore, the general approach is to construct a reference set of known data and then use a forensic tool to automatically search the target to filter irrelevant or relevant artifacts as the case may be. Since the common unit of interest is a file, this technique is often referred to as "known file filtering."

The standard known file filtering approach is to use cryptographic hashes, such as MD5 and SHA1, and construct tables of known hash values (e.g., of operating system and application installation files). NIST maintains the National Software Reference Library (NSRL) [11], and some vendors provide even more extensive collections of known hash values. However, a methodological problem with known file filtering is that two matching objects must be identical 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 new software updates are produced almost daily. Additionally, user artifacts are often modified, and tracking the various versions of the artifacts across multiple targets and network streams can be an overwhelming task.

The similarity digest approach described in this paper targets three basic scenarios:

- **Identification of Embedded/Trace Evidence:** Given a data object (file, disk block, network packet), identify the traces of its presence inside a larger 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 the artifacts that are versions of a set of reference artifacts. The target objects could be executable code (e.g., versions of known software) or user-created data objects (e.g., documents).

- **Cross-Target Correlation:** Identify and correlate different representations of the same or similar object on disk, in RAM or in a network flow. This would allow the automated annotation of disk, RAM and network captures without the deep (and costly) parsing and reconstruction required by current approaches.

In addition to supporting the above scenarios, the approach must meet certain accuracy and performance requirements:

- **High Precision and Recall Rates:** Despite the volume of data, the results must be near certain. A false positive rate of 1% is considered to be excellent in many scenarios; however, this rate would result in 100,000 false positives in a forensic image containing ten million files.

- **Scalable Throughput:** The only hope of keeping up with the growth in data volume is to employ ever larger computational resources. Therefore, a forensic tool should scale horizontally and seamlessly leverage the hardware resources that are made available. Indeed, scalability should be measured in terms of server racks and cabinets, not CPU cores. When a digital forensic laboratory receives a large and urgent case, it should be possible to dedicate the entire data center, if necessary, to obtain results on time.

- **Stream-Oriented Processing:** In terms of performance, commodity hard disks are quickly becoming the new magnetic tapes: their capacities are huge and growing, but the only meaningful way to sustain high throughput is by sequential access. In other words, hard drives need to be read end-to-end, or at least in large chunks. The current approach of file-centric data access via a file system API is convenient for developers, but it does not address the performance nightmare of randomized disk access.

2. Background

The use of similarity digests is motivated by the observation that data fingerprinting methods based on random polynomials, a technique pioneered by Rabin [14], do not work all that well on real data. Specifically, it is difficult to guarantee uniform coverage and control false positives. This section briefly outlines the original similarity digest design and the `sdhash` implementation. Interested readers are referred to [16] for additional details.

2.1 Statistically Improbable Features

The design of `sdhash` was inspired, in part, by information retrieval concepts such as statistically improbable phrases. The underlying notion is to pick a set of features that are statistically rare in the general population to represent an object, and later compare the features with those of another object. If a number of these features are common, then a degree of correlation is indicated. The more features in common, the higher the correlation; if all the features match, then the two objects are likely identical.

The main challenge is to translate this idea from text to binary data and to implement it efficiently. The starting point is to define a feature as a 64-byte sequence (string) and consider all the features in an object (such as a file) as candidates. It is infeasible to collect, store and query empirical probabilities for all possible 64-byte features. Instead, a normalized Shannon entropy measure H_{norm} is computed for each feature.

First, the Shannon entropy is estimated as:

$$H = -\sum_{i=0}^{255} P(X_i) \log P(X_i)$$

where $P(X_i)$ is the empirical probability of encountering ASCII code i. Next, the normalized Shannon entropy measure is computed as:

$$H_{norm} = \lfloor 1000 \times H / \log_2 W \rfloor$$

where all the features are placed in 1,000 equivalence classes.

Using an empirical probability distribution from a representative set, a precedence rank R_{prec} is assigned to each feature, such that a feature with a lower probability has a higher rank.

Next, a popularity rank R_{pop} is defined to express how the precedence rank R_{prec} of a feature relates to those of its neighbors. To calculate the popularity rank R_{pop}, for every sliding window of W consecutive features, the leftmost feature with the lowest R_{prec} is identified and its R_{pop} is incremented by one (evidently, $R_{pop} \leq W$). Intuitively, the rarer the feature, the more likely that the feature has a higher score than its neighbors.

Finally, a sweep is made across all the features, and the features satisfying the property $R_{pop} \geq t$ are selected ($0 < t \leq W$ is a threshold parameter). During this process, an additional filtering step is performed, which simply ignores all the features for which $H_{norm} \leq 100$. This is based on our empirical studies, which indicate that features with low entropy values trigger the vast majority of false positives.

2.2 Similarity Digest Bloom Filter

After the features have been selected, each feature is hashed using SHA1. The result is split into five subhashes (which are treated as independent hash functions) and placed in a Bloom filter [1, 10] of size 256 bytes. Note that a Bloom filter is a probabilistic set data structure, which offers a compressed representation in exchange for a controllable false positive rate.

In the original implementation [16], a Bloom filter is declared to be full when it reaches 128 elements (features) and a new filter is created to accommodate additional features. This process continues until all the features are accommodated, at which time, the sequence of filters corresponds to the similarity digest of the original object. This digest representation is referred to as `sdbf` and the tool that generates and compares the digests is called `sdhash` (or similarity digest hash).

Several parameters can be tuned to provide trade-offs between granularity, compression and accuracy. However, it is also important to have a standard set of parameters so that independently-generated digests are compatible. Based on exhaustive testing, we selected $W = 64$, $t = 16$ along with the parameters mentioned above for the 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 the average, a 7-8 KB chunk of the original artifact.

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

To compare two digests $F = f_1 f_2 \dots f_n$ and $G = g_1 g_2 \dots g_m$ $(n \leq m)$, a similarity distance $SD(F, G)$ is defined as:

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

In other words, for each filter of the shorter digest F, the best match in G is found and the maxima are averaged to produce the final result. In the special case where F has a single filter, the result is the best match, which is the expected behavior. In the case where the two digests have a comparable number of filters, the similarity distance estimates their highest similarity.

3. Block-Aligned Similarity Digests

The main goal of this work is to make the basic algorithm, which has been shown to be robust [17], more scalable by parallelizing it. As a reference point, the optimized serial version of `sdhash` (version 1.3) is capable of generating digests at the rate of 27 MB/s on a 3 GHz Intel Xeon processor. It takes approximately 3 ms to query the digest of a 100 MB target for the presence of a small file, which has a single filter as a digest. Thus, the small-file query rate is approximately 33 MB/ms.

The original digest generation algorithm is sequential; it was primarily targeted toward file-to-file comparisons. To parallelize the algorithm, it is necessary to move away from the chain dependencies among `sdbf` component filters; this allows the concurrent generation of digests.

As it turns out, for larger targets such as disk images, it is possible to use a block-aligned version of `sdbf`, which we call `sdbf-dd`. The idea is relatively simple – the target is split into blocks of fixed size and signature generation is performed in a block-parallel fashion. To make this efficient, the output to a filter is fixed such that each block in the original data maps to exactly one filter in the digest. This design has the added benefit that query matches can be quickly mapped to disk blocks and follow-up processing can be performed.

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

To balance these conflicting requirements, the block size was set to 16 KiB and the maximum number of features per filter was increased from 128 to 192. Under average conditions, we would expect that a 16 KiB block would produce upwards of 256 features that clear the standard threshold of $t = 16$. Therefore, the features are selected starting with the most popular features until the filter is filled or none of the remaining features are above the threshold.

It was also necessary to increase the maximum number of features per filter for the `sdbf` version from 128 to 160. This introduces two improvements: (i) the average data chunk covered by a single filter is increased from 7-8 KB to 9-10 KB, thereby shortening the digest and reducing comparison times; and (ii) more features can be accumulated, providing for better compatibility with the `sdbf-dd` version.

Note that the specific values of the parameters – (i) number of features per `sdhash` filter (128 to 192); (ii) number of features per `sdbf-dd` filter (128 to 256); and (iii) `sdbf-dd` block size (8 KB to 16 KB) – were closen using a grid search to strike the right balance between true positive and false positive rates.

With these parameter values, the in-memory `sdhash-dd` representation requires approximately 1.6% of the size of original data (260 bytes for every 16,384 bytes of data). Thus, 1 TB data can be represented by a 16 GB digest, which easily fits in RAM. The new `sdhash` representation with variable chunk size and 160 elements per Bloom filter requires, on the 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 easily recoverable with standard lossless compression for archival purposes.

4. Experimental Evaluation

This section assesses the throughput and accuracy of the proposed approach. All the measurements are based on `sdhash` version 1.7, which is available at `roussev.net/sdhash/sdhash.html`.

4.1 Throughput

All the performance tests were run on a Dell PowerEdge R710 server with two six-core Intel Xeon 2.93 GHz CPUs with hyper-threading, for a

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

total of 24 hardware-supported threads. The host was equipped with a 800 MHz processor, 72 GB RAM and hardware-supported RAID 10 with a benchmarked sequential throughput of 300 MB/s. Unless otherwise noted, the results are presented as stable averages over multiple runs.

4.1.1 Block-Aligned Digest Generation. This experiment quantified the scalability of the parallelized version as a function of the number of threads employed. The experiment used a 10 GB memory-resident target, which was hashed to eliminate I/O effects.

The results are shown in Table 1. It is evident that excellent scalability is achieved, with the 24-threaded runs achieving 370 MB/s of throughput. For reference, a SHA1 job on the same target produced a throughput of 333 MB/s. It is also clear that, given enough hardware parallelism, the `sdhash-dd` computation is I/O-bound.

Additionally, `sdhash-dd` was applied with 24 threads to a 100 GB target. The computation took 475 seconds with a cold cache, yielding an effective throughput of 210 MB/s.

4.1.2 File-Parallel Digest Generation. File-parallel hashing was implemented to scale up the performance of the `sdhash` version of the code, which is still the preferred digest algorithm for most files. To quantify the performance gain, a 39,700 file sample from the Govdocs1 corpus [6, 7] was employed; the total size of the sample was 26.75 GB.

Table 2 summarizes the results. Note that 300 files were not hashed because their size was under 512 bytes. Also, all the files were cached, so the performance of the I/O system was not a factor in the experiment.

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 to their size. Thus, an “unlucky” thread could be tasked to digest a well above average amount of data and, therefore, slow down the overall completion time. Another impediment to further speedup is that many of the files in the sample were small,

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

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

which increased the overhead and reduced concurrency due to extra I/O and memory-management operations.

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

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

To better understand these effects, a synthetic set of 10,000 files (500 KB per file) was created, and the experiment was performed again. The results are shown in Table 3.

Additionally, `sdhash` was applied with 24 threads to each of the 147,430 files on the mounted 100 GB target from the previous experiment. With a cold cache, 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 above: (i) `sdhash-dd`; (ii) `sdhash` on real files; and (iii) `sdhash` on an optimally-balanced workload.

Note that, although the experimental server supported 24 hardware threads, there were only twelve CPU cores. Running two threads on the same core only yields benefits if the threads have complimentary workloads or a thread is stalled. In the experiments, the threads competed for the same CPU units, so the speedup can be expected to increase moderately when going from 12 to 24 threads. The graph in Figure 1 clearly demonstrates this behavior, which results in the `sdhash` scalability flattening out. Another contributing factor to the difference between the `sdhash` and `sdhash-dd` scalabilities is that the former has higher

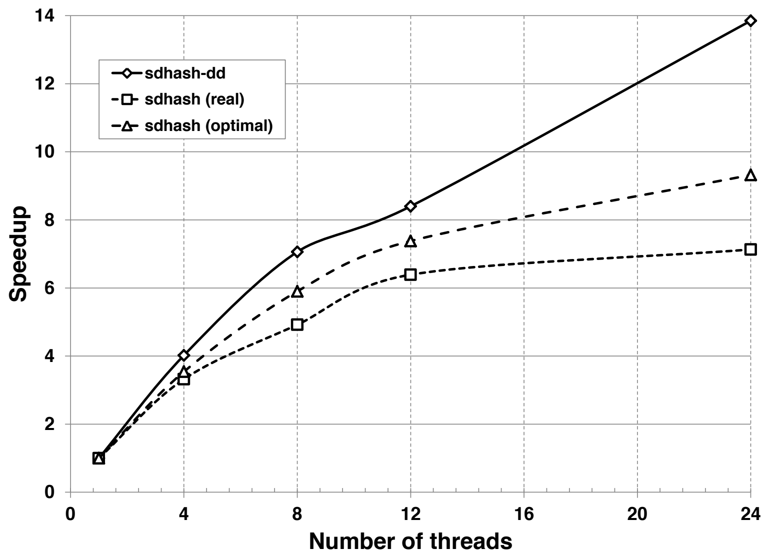

Figure 1. `sdhash` speedup summary.

memory management overhead – multiple allocations/deallocations per file versus one large allocation for `sdhash-dd`.

Table 4. `sdhash` digest comparison performance with Bloom filters.

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

4.1.3 Digest Comparison Rates. Table 4 shows the digest comparison rates at various levels of concurrency. The experiment involved 372.9 million Bloom filter (BF) comparisons. The main measure is the number of Bloom filter comparisons per millisecond (BF/ms). Based on this measure, it is possible to calculate the expected execution time for different workloads.

One of the main benchmarks is the comparison rate for small files ($\leq$ 16 KiB) that have digests of one Bloom filter. Based on the 24-thread rate, the search of (the digest of) a target for a small file should proceed at the rate of $58,000 \times 16,384$ bytes = 950 MB/ms or 950 GB/s.

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

The above rates are inclusive of the time taken to load the digests. To gain a better estimate of comparison rates, we compared the `sdbf-dd` digest of a 100 GB target (6.1 million Bloom filters) with a set of 300 small files extracted from the image with a total Bloom filter count of 908. The total execution time using the 24 threads completed in 79 seconds, of which 15 seconds were spent 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/s for the small file case. This figure is representative of a server-side deployment in which the reference set is loaded once and queried many times.

4.2 Accuracy

We recently performed a detailed study [17] of the base `sdhash` algorithm and its performance on controlled and real-world data. Therefore, we limit our discussion here to validating the new parameters chosen for the algorithm by quantifying the accuracy of the `sdbf` versus `sdbf-dd` comparisons.

The experiment for measuring accuracy used synthetic targets generated from random data to precisely control the contents of a query and target. For each query size, a 100 MB target and 10,000 control files of the given query size were generated. Then, 10,000 samples of the given size were extracted from the target. The target was hashed using `sdhash-dd` while the remaining 20,000 files were hashed with `sdhash` and then compared with the target; a zero result was assumed to be a negative, whereas a non-zero result was counted as a positive. Ideally, all the samples were expected to match, while none of the controls were expected to match.

Table 5 shows the results for query sizes of 1,000 to 3,800 bytes. The experiment investigated query sizes up to 64 KB; however, for reasons

Table 5. Error rates for `sdhash` queries versus `sdhash-dd` target.

Query Size (KB)	FP Rate	TP Rate	Query Size (KB)	FP Rate	TP Rate
1.0	0.1906	1.000	2.0	0.0006	0.997
1.1	0.0964	1.000	2.2	0.0005	1.000
1.2	0.0465	1.000	2.4	0.0001	1.000
1.3	0.0190	1.000	2.6	0.0001	0.997
1.4	0.0098	1.000	2.8	0.0000	1.000
1.5	0.0058	1.000	3.0	0.0000	0.999
1.6	0.0029	0.999	3.2	0.0000	0.998
1.7	0.0023	0.999	3.4	0.0000	0.998
1.8	0.0013	0.999	3.6	0.0000	1.000
1.9	0.0010	0.998	3.8	0.0000	0.998

of brevity, the results obtained for higher query sizes are omitted. Note, however, that the observed behavior for higher query sizes was essentially identical to that observed for query sizes of 2 KB and higher.

The results demonstrate that the tool produces near-perfect results for queries of 2 KB and up, which are relevant to a block storage device. For low-end queries, which are relevant to network traffic investigations, the false positive (FP) rate starts out high (due to the small number of features selected), but quickly drops below 1% for queries of size 1.4 KB and up. This is not a coincidence, but the result of tuning the performance parameters, which was discussed above. In general, more than 85% of Internet traffic by volume is in the form of packets in the 1.4 KB to 1.5 KB range [5]; in other words, long data transfers dominate. In a network investigation scenario, the false positive rate for smaller queries can be managed by searching for multiple packets of the same source; e.g., if `sdhash` flags three independent packets of 1 KB as belonging to a file, the empirical probability that all three are wrong is only $0.19^3 = 0.0069$.

5. Related Work

The idea of generating a flexible data fingerprint dates back to Rabin's seminal work that was published in 1981 [14]. Since then, many researchers have developed more complex versions, but Rabin's basic idea has carried over with relatively small variations. In the following, we limit the discussion to the essential ideas. Interested readers are referred to [15] for a detailed survey of hashing and fingerprinting techniques.

5.1 Rabin Fingerprinting

Rabin's scheme is based on random polynomials; its original purpose was "to produce a very simple real-time string matching algorithm and a procedure for securing files against unauthorized changes" [14]. A Rabin fingerprint can be viewed as a checksum with a low, quantifiable collision probability 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 (text) objects. For example, Manber [9] used the concept in his `sif` tool to quantify similarities among texts files; Brin[2], in his pre-Google years, used it in a copy-detection scheme; and Broder [3] applied it to find syntactic similarities among web pages.

The basic idea, alternatively called anchoring, chunking or shingling, is to use a sliding Rabin fingerprint over a fixed-size window to split the data into pieces. For every window of size w, a hash h is computed, divided by a chosen constant c, and the remainder is compared with another constant m. If the two values are equal (i.e., $m \equiv h \bmod c$), then the beginning of a chunk (anchor) is declared, the sliding window is moved one position, and the process is repeated until the end of the data is reached. For convenience, the value of c is typically a power of two ($c = 2^k$) and m is a fixed number between zero and $c - 1$. After the baseline anchoring is determined, the characteristic features can be selected by: (i) choosing the chunks in between the anchors as features; (ii) starting at the anchor position and picking the next l number of bytes; or (iii) using 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. Also, 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. This property can be used to address the fragility problem in traditional file and block-based hashing.

Consider two versions of the same document. One of the documents can be viewed as being derived from the other as a result of inserting and deleting characters. For example, an HTML page could be converted to plain text by removing 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, facilitating an automatic correlation of the versions. For the actual feature comparison, the hashes of the selected features can be stored and used as a space-efficient representation of the fingerprint of an object.

5.2 Similarity Hashing

Kornblum [8] was among the first researchers to propose the use of a generic fuzzy hash scheme for forensic purposes. His `ssdeep` tool generates string hashes of up to 80 bytes that are the concatenations of 6-bit piecewise hashes. The comparison is then performed using the 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.

Roussev, *et al.* [18] have proposed a scheme that 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 [19] that balances performance and accuracy requirements by keeping hashes at several resolutions.

Outside the discipline of digital forensics, Pucha, *et al.* [13] have proposed an interesting scheme for identifying similar files in peer-to-peer networks. The approach identifies large-scale similarity (e.g., the same movie in different languages) that can be used to offer alternative download sources in peer-to-peer networks.

5.3 Summary

The randomized model of Rabin fingerprinting works well on the average, but when applied to real-world data often suffers from coverage problems and high false positive rates. Both these problems can be traced to the fact that the underlying data can have significant variations 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, rendering the fingerprint an unreliable basis for comparison.

Active research on payload attribution systems has produced ever more complicated versions of Rabin fingerprints with the goal of ensuring even coverage (see, e.g., [4, 12, 21, 22]. 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

The similarity-digest-based approach presented in this paper can efficiently correlate large forensic data sets at the byte-stream level. This enables content filtering to be applied on a wide scale and the filtering to be incorporated early in the forensic process. Current best practices dictate that the creation of a cloned copy of the original media, such as a hard drive, is the first step in a formal investigation. However, it is usually the case that no useful processing is done during this lengthy task and, at the end of the task, the investigators know almost nothing more than they did at the beginning of the task. The proposed approach can generate similarity digests at line speed so that content querying can be performed immediately after media cloning. In fact, it is entirely feasible to take this a step further and to start matching hashed data against reference sets even while hashing/cloning is being performed.

Experimental results using an $8,500 commodity rack server demonstrate that a hash generation rate of up to 370 MB/s can be sustained on a 24-threaded system. Content searches of a small (16 KiB) query file in a reference set are executed at the rate of 1.4 TB/s. Moreover, near-perfect true and false positive rates are obtained for query objects of size 2 KB and higher.

The capabilities discussed in this paper can qualitatively change the scope and efficiency of digital forensic investigations. Still, the approach is just the first attempt at massively scaling data correlation. We hope that this work will stimulate efforts to dramatically speed up other automated forensic processing techniques to deal with massive data volumes.

References

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

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

[3] A. Broder, S. Glassman, M. Manasse and G. Zweig, Syntactic clustering of the web, *Computer Networks and ISDN Systems*, vol. 29(8-13), pp. 1157–1166, 1997.

[4] C. Cho, S. Lee, C. Tan and Y. Tan, Network forensics on packet fingerprints, *Proceedings of the Twenty-First IFIP Information Security Conference*, pp. 401–412, 2006.

[5] Cooperative Association for Internet Data Analysis, Packet size distribution comparison between Internet links in 1998 and 2008, San Diego Supercomputer Center, University of California at San Diego, San Diego, California (`www.caida.org/research/traffic-analysis/pkt_size_distribution/graphs.xml`), 2010.

[6] Digital Corpora, NPS Corpus (`digitalcorpora.org/corpora/disk-images`).

[7] S. Garfinkel, P. Farrell, V. Roussev and G. Dinolt, Bringing science to digital forensics with standardized forensic corpora, *Digital Investigation*, vol. 6(S), pp. S2–S11, 2009.

[8] J. Kornblum, Identifying almost identical files using context triggered piecewise hashing, *Digital Investigation*, vol. 3(S1), pp. S91–S97, 2006.

[9] U. Manber, Finding similar files in a large file system, *Proceedings of the USENIX Winter Technical Conference*, pp. 1–10, 1994.

[10] M. Mitzenmacher, Compressed Bloom filters, *IEEE/ACM Transactions on Networks*, vol. 10(5), pp. 604–612, 2002.

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

[12] M. Ponec, P. Giura, H. Bronnimann and J. Wein, Highly efficient techniques for network forensics, *Proceedings of the Fourteenth ACM Conference on Computer and Communications Security*, pp. 150–160, 2007.

[13] H. Pucha, D. Andersen and M. Kaminsky, Exploiting similarity for multi-source downloads using file handprints, *Proceedings of the Fourth USENIX Symposium on Networked Systems Design and Implementation*, pp. 15–28, 2007.

[14] M. Rabin, Fingerprinting by Random Polynomials, Technical Report TR1581, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, 1981.

[15] V. Roussev, Hashing and data fingerprinting in digital forensics, *IEEE Security and Privacy*, vol. 7(2), pp. 49–55, 2009.

[16] V. Roussev, Data fingerprinting with similarity digests, in *Advances in Digital Forensics VI*, K. Chow and S. Shenoi (Eds.), Springer, Heidelberg, Germany, pp. 207–226, 2010.

[17] V. Roussev, An evaluation of forensic similarity hashes, *Digital Investigation*, vol. 8(S), pp. S34–S41, 2011.

[18] V. Roussev, Y. Chen, T. Bourg and G. Richard, `md5bloom`: Forensic filesystem hashing revisited, *Digital Investigation*, vol. 3(S), pp. S82–S90, 2006.

[19] V. Roussev, G. Richard and L. Marziale, Multi-resolution similarity hashing, *Digital Investigation*, vol. 4(S), pp. S105–S113, 2007.

[20] V. Roussev, G. Richard and L. Marziale, Class-aware similarity hashing for data classification, in *Research Advances in Digital Forensics IV*, I. Ray and S. Shenoi (Eds.), Springer, Boston, Massachusetts, pp. 101–113, 2008.

[21] S. Schleimer, D. Wilkerson and A. Aiken, Winnowing: Local algorithms for document fingerprinting, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 76–85, 2003.

[22] K. Shanmugasundaram, H. Bronnimann and N. Memon, Payload attribution via hierarchical Bloom filters, *Proceedings of the Eleventh ACM Conference on Computer and Communications Security*, pp. 31–41, 2004.