Automated Evaluation Of Approximate Matching Algorithms On Real Data

By

Frank Breitinger and Vassil Roussev

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2014 EU

Amsterdam, NL (May 7th - 9th)

Digital Investigation 11 (2014) S10–S17

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Automated evaluation of approximate matching algorithms on real data

Frank Breitinger[a,*], Vassil Roussev[b]

[a] da/sec - Biometrics and Internet Security Research Group, Hochschule Darmstadt, Haardtring 100, 64295 Darmstadt, Germany
[b] Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA

Keywords:
Digital forensics
Approximate matching
Hashing
Similarity hashing
sdhash
mrsh-v2
ssdeep
FRASH

ABSTRACT

Bytewise approximate matching is a relatively new area within digital forensics, but its importance is growing quickly as practitioners are looking for fast methods to screen and analyze the increasing amounts of data in forensic investigations. The essential idea is to complement the use of cryptographic hash functions to detect data objects with *bytewise identical* representation with the capability to find objects with *bytewise similar* representations.

Unlike cryptographic hash functions, which have been studied and tested for a long time, approximate matching ones are still in their early development stages and evaluation methodology is still evolving. Broadly, prior approaches have used either a human in the loop to *manually* evaluate the goodness of similarity matches on real world data, or controlled (pseudo-random) data to perform *automated* evaluation.

This work's contribution is to introduce automated approximate matching evaluation on real data by relating approximate matching results to the longest common substring (LCS). Specifically, we introduce a computationally efficient LCS approximation and use it to obtain ground truth on the *t*5 set. Using the results, we evaluate three existing approximate matching schemes relative to LCS and analyze their performance.

Introduction

One of the biggest challenges facing a digital forensic investigation is coping with the huge number of files that need to be processed. This is a direct result of the exponential growth in our ability to store digital artifacts and the overall trend of digitizing all forms of information, such as text, documents, images, audio and video. Consequently, a critical requirement of modern forensic investigative tools is the ability to perform large-scale automated filtering and correlation of data.

One of the most common processing methods is *known file filtering*, which–in its basic form–consist of computing the crypto hashes for all files on a target device, and comparing them to a reference database. Depending on the underlying database, files are either filtered out (e.g., files of the operating system) or filtered in (e.g., known offensive content). For example, NIST maintains a large public database of known content–the NSRL (NIST Information Technology Laboratory, 2003–2013).

Reference databases based on crypto hashes provide precise and reliable results; however, they can only identify content based on *identity*. This makes them fragile and difficult to maintain as digital artifacts (such as code) get updated on a regular basis, making the reference data obsolete. Therefore, it is useful to have algorithms that provide *approximate* matches that can correlate closely related versions of data objects.

[*] Corresponding author.
E-mail addresses: frank.breitinger@cased.de (F. Breitinger), vassil@roussev.net (V. Roussev).

http://dx.doi.org/10.1016/j.diin.2014.03.002

Generally, an approximate matching algorithm extracts features of an input and produces a similarity digest; the digests can then be compared to determine a measure of similarity. Depending on the level of at which the algorithm operates, one distinguishes between bytewise, syntactic- or semantic approximate matching Breitinger et al. (2014).

Semantic matching operates at the highest level of abstraction and provides results that are closest to human perceptual notions of similarity. Syntactic matching relies on purely syntactic rules to break up the data representation into features, e.g., cutting a header of a fixed byte size. Bytewise matching relies only on the sequence of bytes that make up a digital artifact, without reference to any structures (or their interpretation) within the data stream. In what follows, we focus on bytewise approximate matching.

While crypto hashes are well-known and established in various fields of computer science, approximate matching is a rather new area and missing standardize processes for testing and evaluating these algorithms. Breitinger et al. (2013a) presented a test framework called FRASH which performs efficiency as well as sensitivity & robustness tests. Some time later FRASH was extended by a precision & recall test on synthetic data (Breitinger et al., 2013b).

The main contribution of this work is the development of *automated* precision and recall tests on *real world data*. Compared to synthetic data, real world data yields more realistic results and allows a better characterization of the behavior of approximate matching algorithms. For this test, we first created a ground of truth wherefore we identified the similarity of objects based on an own metric called *approximate longest common substring* (aLCS). We validate our aLCS results by comparing them against the traditional *longest common substring* (LCS). In a second step, we analyze the false positive and false negative rates of the approximate matching algorithms with respect to the ground truth.

The rest of the paper is organized as follows: Sec. 2 introduces the necessary background and related work. Our evaluation methodology as well as some implementation details are provided in Sec. 4. The core of this paper is Sec. 5 where we present our experimental results. Sec. 6 concludes the paper.

Background & related work

Hash functions (e.g., SHA-1 Gallagher and Director (1995)) have a long tradition and are applied in various fields of computer science like cryptography (Menezes et al., 2001), databases (Sumathi & Esakkirajan, 2007, Sec. 9.6) or digital forensics (Altheide and Carvey, 2011, p. 56ff). This is in contrast to bytewise approximate matching which probably had its breakthrough in 2006 with an algorithm called context triggered piecewise hashing (CTPH). In the following we give a brief overview of bytewise approximate matching and explain how these algorithms are tested.

Bytewise approximate matching algorithms

The introduction of approximate matching for forensic purposes dates back to 2006 when Kornblum (2006) presented an approach called *context triggered piecewise hashing* and Roussev et al. (2006) introduced *similarity hashing*. Subsequently a small community came up which follows the challenges of approximate matching (a.k.a. 'similarity hashing'). To date, several different approaches have been published, all with different strength and weaknesses.

Besides the two most prominent implementations `ssdeep` and `sdhash`, a couple of further algorithms raised. However, most of them are very limited. For instance, `MinHash` (Broder, 1997) and `SimHash` (Sadowski and Levin, 2007) allow to detect small changes (up to several bytes) only, `bbHash` is too slow (2 min for 10 MiB), `mvHash-B` is not file type independent. Hence, in what follows we briefly describe the three most promising approaches with respect to digital forensics (a detailed description is beyond the scope of this paper as we treat them black boxes for testing purposes).

ssdeep

The `ssdeep` tool ([1]) was introduced as a proof of concept implementation of *context triggered piecewise hashing* (CTPH) and has gained widespread acceptance. It was presented by Kornblum (2006) and is based on the spam detection algorithm from Tridgell (2002–2009). The basic idea is behind it is simple: split an input into chunks, hash each chunk independently and concatenate the chunk hashes to a final similarity digest (a.k.a. fingerprint).

In order to split an input into chunks, the algorithm identifies trigger points using a rolling hash (a variation of the Adler-32 function) which considers the current context of seven bytes. Each chunk is then given to the non-cryptographic hash function FNV Noll (1994–2012). Instead of using the complete FNV hash, CTPH only takes the least significant 6 bits which is equal to one Base64 character. Thus, two files are similar if the have common chunks.

Follow up efforts (Chen and Wang, 2008; Seo et al., 2009; Baier and Breitinger, 2011) have targeted incremental improvement of the algorithm, however, none of these implementations have been made available for public testing and evaluation.

sdhash

The `sdhash` tool[2] was introduced four years later Roussev (2010) in an effort to address some of the shortcomings of `ssdeep`. Instead of dividing an input into chunks, the `sdhash` algorithm picks statistically improbable features to represent each object. A feature in this context is a byte sequence of 64 bytes, which is hashed using SHA-1 and inserted into a Bloom filter (Bloom, 1970). The similarity digest of the data object is a sequence of 256-byte Bloom filters, each of which represents approximately 10 KB of the original data. The tool also supports *block* mode (Roussev, 2012) in which the input is split into fixed-size chunks (by default 16 KiB) and the best features are selected from each block.

[1] http://ssdeep.sourceforge.net (last accessed 5 Dec. 2013).

[2] http://sdhash.org (last accessed 5 Dec. 2013).

mrsh-v2

mrsh-v2 was proposed by Breitinger and Baier (2012) and is based on MRS hash (Roussev et al., 2007) and combines design ideas from both `ssdeep` and `sdhash`. The overall approach is to divide an input into chunks and hash each chunk based on a rolling hash (`ssdeep`) and combine it with `sdhash`-like use of Bloom filters for the similarity digest.

Approximate matching evaluation

In order to facilitate the *systematic* and *reproducible* testing of bytewise approximate matching, (Breitinger et al., 2013a) introduced an open source, extensible framework called FRASH. It is implemented in Ruby 2.0, and the current version provides facilities for evaluating three different aspects of an approximate matching algorithm's performance (some of these build on ideas from (Roussev, 2011)).

- Efficiency:
  - *generation efficiency* measures the execution time taken by the algorithm to process an input of a given size and generate the similarity digest;
  - *comparison efficiency* measures the time taken to perform the similarity digest comparisons;
  - *space efficiency* (*compression*) measures the size of the digest as function of the input length.
- Sensitivity and robustness:
  - *fragment detection* is a basic *sensitivity* measure, which gives the smallest fragment for which the similarity tool *reliably* correlates the fragment and the original file;
  - *single-common-block correlation* provides a related *sensitivity* measure, which calculates the smallest object that two files need to have in common for the algorithm to reliably correlate them;
  - *alignment robustness* analyzes the resilience of the algorithm by measuring the impact of inserting random byte sequences at the beginning of an input on its correlation results;
  - *random noise resistance* analyzes the impact of random input edits on the correlation results of the algorithm.
- *Precision and recall* tests quantify the *detection error trade-off* between false positive and false negative rates (a.k.a. *ROC*) for the algorithms.

Problem description

Critical in any evaluation process is the establishment of ground truth; for our subject area, this means establishing whether or not two digital objects (files) are similar. Prior work, such as (Roussev, 2011), has approached this problem from two perspectives: automated controlled tests based on pseudo-random data, and manual user evaluation of positive results.

The main advantage of controlled experiments is that ground truth is constructed and, therefore, precisely known. This allows randomized tests to be run completely automatically and the results to be interpreted with standard statistical measures. The obvious downside is that much of real data is far from random so the applicability of the result to the general case remains uncertain. Nevertheless, running controlled tests in this manner is quite useful in characterizing algorithms' baseline capabilities.

The main advantage of user evaluation is that it provides results on real data as it would be experienced by an investigator. The downside is that the process is manual and, therefore, not suitable for large-scale testing. Also, the results include a degree of subjective judgment on whether two objects are, in fact, similar. Finally, there is also the problem of how to treat objects that exhibit non-trivial commonality that is not normally observable by the user (the significance of such findings is inherently case-specific).

In this work, we seek to bridge the gap between the two approaches by providing the means to perform *fully automated testing on real data*. In order to solve this challenge, we need a practical algorithm that can establish whether two (arbitrary) data objects are similar, or not.

Recall that we are interested in byte-level similarity, which means that we do not consider any syntactic, or semantic features in our analysis. In other words, the artifacts are being compared as strings and similarity is defined as the presence of common substrings.

Given this framework, it is natural to consider the *longest common substring* (LCS) as starting point for defining the similarity of two objects. For instance, consider the strings ABABBCADEF and ABAECADEBF–their longest common substring is CADE; since its length is 40% of the length of the strings, we could use that as a baseline measure of similarity. In general, it is clear that there could be additional sources of commonality, so LCS should be considered a lower bound.

One problem with using LCS is that the algorithm has quadratic time complexity–$O(mn)$, where m and n are the string lengths. Given that files could be quite large, and the number of test cases grows quadratically as a function of the number of files in the test set, the use of an exact algorithm quickly becomes infeasible. Therefore, we created a tool which outputs a good approximation of the longest common substring and, by design, provides a lower bound on LCS.

Approximate longest common substring

The basic idea of the approximate longest common substring metric (aLCS) is not to compare files byte by byte but rather in variable sized chunks. To pick the chunks, we use a derivative of the standard approach to data fingerprinting by random polynomials pioneered by Rabin. Specifically, we borrow the rolling hash from `ssdeep` and adjust the parameters such it produces chunks of 40 bytes, on average. Each chunk is hashed with the FNV-1a hash (Noll, 1994–2012) and the sequence of all hash values form the basis for the aLCS signature. Besides the hash values, we also store the entropy and length for each chunk and the resulting sequence forms the *alcs-digest*.

Given two digests, it is straight forward to construct an estimate of the LCS; a reference implementation is publicly available at http://www.dasec.h-da.de/staff/breitinger-frank/.

Table 1
Empirical *pdf* & *cdf* for d_r

X	0	1	2	3	4	5	10	15	20
$Pr\{d_r = X\}$	0.8869	0.0449	0.0155	0.0040	0.0047	0.0116	0.0062	0.0001	0.0000
$Pr\{d_r \leq X\}$	0.8869	0.9318	0.9473	0.9513	0.9561	0.9677	0.9834	0.9992	0.9999

Implementation details

The tool is implemented in *C* and separated into three steps: reading, hashing and comparison, which are declared in the main function. It has a command line interface and is run against all files in a target directory: `./aLCS <dir>`.

First, all files in `dir` are read. Out of the file names, we create 'hash-tasks' which are added to a thread pool. A hash-task contains the path to a file and denotes 'hash file *x*'. The tasks are run in parallel on the available level of parallelism (CPU cores). Once all *alcs-digests* are created, we perform an all-against-all comparison, which launches parallel compare-tasks (compare *file*$_1$ against *file*$_2$). The results are serialized to standard output.

The reference implementation has three main settings configurable in `header/config.h`. `MIN_LCS` is the minimum L_a length which is printed to stdio and is by default 0 (all comparison are printed). The `THREAD_POOL_QUEUE_SIZE` is the length of the queue and should be *fileamount*$\cdot$(*fileamount* $-$ 1)/2. `NUMTHREADS` is the amount of threads which should be equal to the amount of cores.

Verification of ground truth

To verify the correctness of our approximate longest common substring, we compared the results against LCS for some real world files. In order to solve this challenge, we implemented a parallelized LCS tool written in *C*. The output is a summary file similar structured then our aLCS output: `file1 | file2 | LCS`. A small ruby script is used to compare LCS-summary and aLCS-summary.

Our subset consists of 201 randomly selected files. We compared these files using aLCS as well as LCS and finally compared both summaries. All 20,100 comparisons yield a true positive, i.e., $0 \leq alcs \leq lcs$. We also consider the distribution of the differences between the LCS and aLCS scores. Specifically, we define d_r for files f_1 and f_2 as follows:

$$d_r = \lceil 100 \times \frac{lcs(f_1,f_2) - alcs(f_1,f_2)}{\min(|f_1|,|f_2|)} \rceil, d_r \in 0,1,\ldots,100.$$

In other words, we consider the score difference relative to the size of the smaller of the two files, and build the empirical distribution in Table 1. As we can see, upwards of 95% of the observed differences do not exceed 3% of the size of the smaller files – we consider this a reasonable starting point for our purposes (further research may refine this). If anything, this should give tools a slight boost as the available commonality would be underestimated.

Methodology and implementation

This section first presents our approximate ground truth in Sec. 4.1 followed by the test methodology in Sec. 4.2. The last part of this chapter presents the used terminology, notations and definitions.

Approximate ground truth

Our approximate ground truth is a text file of unordered pairs of files (f_1, f_2) structured like follows:

```
filename1 | size1 | filename2 | size2 | L_a | entropy
| L_r
 1.pdf | 98781 | 2.pdf | 185271 | 2500 | 4.66 | 0.03
 3.pdf | 16661 | 4.pdf | 18530 | 2077 | 1.75 | 0.12
```

where L_a is the absolute result (a lower bound on the length of the longest common substring), entropy is the information content of the substring and L_r is the relative result. More precisely,

$$L_a = alcs_a(f_1,f_2), \text{ where } 0 \leq L_a \leq \min(|f_1|,|f_2|). \quad (1)$$

$$L_r = \lceil 100 \times \frac{L_a}{\min(|f_1|,|f_2|)} \rceil, \text{ where } 0 \leq L_r \leq 100. \quad (2)$$

where $|f|$ denotes the file size in bytes.

For instance, the first line states that file1 and file2 have an absolute length $L_a = 2500$ bytes which corresponds a relative length $L_r = 0.03 = 3\%$ with an entropy of 4.66. The second line shows a special case with a very low entropy which could be an indicator that both files share mostly zeros.

Test methodology

Although, in theory, any two strings sharing a substring are related, we place a more practical lower bound on the minimum amount of commonality to declare two files related. Specifically, we require that the absolute size L_a is at least 100 bytes *and* that the relative result L_r exceeds 0.5% of the size of the smaller file. More formally, the true positive function $TP_{alcs}\,(f_1 f_2)$ is defined as[3]

$$TP_{alcs}(f_1,f_2) \equiv L_a \geq 100 \wedge L_r \geq 1.$$

Clearly, the *true negative* function

$$TN_{alcs}(f_1,f_2) = \neg TP_{alcs}(f_1,f_2).$$

Terminology, notations and definitions

We follow a fairly standard information retrieval framework for evaluating the quality of the results produced by approximate matching tools.

[3] Note: result of L_r is rounded and thus 0.5 is equal to 1.

Approximate matching score: $S_h(f_1,f_2)$ is the result of comparing two files using an approximate matching function h, where $h \in \{ssdeep, mrsh, sdhash\}$ and $0 \le S_h \le 100$.

Threshold (t) of significance: A score parameter, used in approximate matching to separate matches from non-matches.

Match: Two files, f_1 and f_2, are matched using approximate matching algorithm $h \Leftrightarrow S_h(f_1,f_2) > t$.

True positive TP_h:

$$TP_h(f_1,f_2,t) \equiv TP_{alcs}(f_1,f_2) \ = \ true \wedge S_h(f_1,f_2) > t$$

True negative TN_h:

$$TP_h(f_1,f_2,t) \equiv TN_{alcs}(f_1,f_2) \ = \ true \wedge S_h(f_1,f_2) \le t$$

False positive FP_h:

$$TP_h(f_1,f_2,t) \equiv TN_{alcs}(f_1,f_2) \ = \ true \wedge S_h(f_1,f_2) > t$$

False negative FN_h:

$$TP_h(f_1,f_2,t) \equiv TP_{alcs}(f_1,f_2) \ = \ true \wedge S_h(f_1,f_2) \le t$$

Precision P_h:

$$P_h \ = \ \frac{TP_h}{TP_h + FP_h}$$

Recall R_h:

$$R_h \ = \ \frac{TP_h}{TP_h + FN_h}$$

True negative rate TNR_h:

$$TNR_h \ = \ \frac{TN_h}{TN_h + FP_h}$$

Accuracy A_h:

$$A_h \ = \ \frac{TP_h + TN_h}{TP_h + TN_h + FP_h + FN_h}$$

F-score F_β:

$$F_\beta \ = \ \left(1+\beta^2\right) \times \frac{precision \times recall}{\beta^2 \times precision + recall}$$

The F-score is a generic measure combining precision and recall into a single number. We use three different versions based on the β parameter: F_1,F_2,$F_{0.5}$. The first one weighs precision and recall equally, the second favors recall over precision, and the last one favors precision over recall.

Matthews correlation coefficient MCC:

$$MCC \ = \ \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

where $MCC \in [-1,1]$.

The MCC is a correlation coefficient between observed and predicted binary classifications; it is included here as it is considered a balanced measure even for classes of substantially different sizes (as is our case). A result of $+1$ represents perfect prediction, whereas -1 indicates perfect disagreement; 0 indicates that the classifier offers no advantage over a random guess.

Experimental results

This section presents the results from applying our test methodology to analyze the performance `ssdeep`, `mrsh-v2` and `sdhash`.

The assessment is based on the $t5$ corpus first used in (Roussev, 2011), which contains 4457 files with a total size of 1.8 GB (http://roussev.net/t5/). Thus, the average file is ≈400 KB and the file type distribution is given in Table 2.

One challenge that may not be immediately obvious is that a complete, all-pairs comparison run requires a non-trivial number of comparisons–a set of n files results in $n(n-1)/2$ comparisons, which corresponds to 9,930,196 comparisons for the $t5$ set. Although it takes only ≈425 ms per comparison, such work would clearly be impractical without parallel execution. Fortunately, such a workload is readily parallelizable and our implementation takes full advantage of that. In our tests, a 48-core, 2.6 GHz AMD Opteron server needed 1466 min (≈24 hours) to generate and compare all alcs-digests.

The rest of this chapter is divided into the following parts. First, we give a general overview of the detection rates of the different approaches. The next three sections discuss the false positives, false negatives and true positives, respectively. Finally, the last section shows the differences in performance for the containment and resemblance usage scenarios.

Baseline results

First, we present the baseline case where $t = 0$. In other words, we define any approximate matching score greater than zero as a positive result, and any zero score as a negative result. This is the lowest barrier for matching algorithms to jump over as they simply need to match the positive/negative behavior of the baseline aLCS measure, with no additional expectations of the exact value of the score.

Using the definitions from 4.3, the observed statistics from the experiments are shown in Table 3 and lead us to the following initial observations:

- *Precision.* In absolute terms, `sdhash` yields the largest number of true positives, while `ssdeep` is with the lowest number of false positives; `mrsh-v2`'s true positives fall right in the middle but the false positives are much higher than the other two. This results in reasonably high precision for `ssdeep` and `sdhash`, and relatively low one for `mrsh-v2`.
- *Recall.* Due to the high number of false negatives across the board, the recall rates are quite low. In relative terms, `sdhash` and `mrsh-v2` hold a considerable advantage over `ssdeep`.

Table 2
Number of files per file type: $t5$ corpus.

jpg	gif	doc	xls	ppt	html	pdf	txt
362	67	533	250	368	1093	1073	711

Table 3
Baseline approximate matching results for t=0.

	ssdeep	mrsh-v2	sdhash
TP	951	3679	5474
FP	15	23,453	790
TN	9,472,047	9,448,609	9,471,272
FN	457,183	454,455	452,660
Precision	0.98447	0.13560	0.87388
Recall	0.00010	0.00039	0.00058
TNR	1.00000	0.99752	0.99992
Accuracy	0.95396	0.95187	0.95434
F_1	0.00020	0.00078	0.00115
F_2	0.00013	0.00049	0.00072
$F_{0.5}$	0.00050	0.00192	0.00288
MCC	0.04412	0.02232	0.09913

- *TNR & Accuracy*. Due to the very high ratio of negative to positive results, these measures do not provide any meaningful differentiation among the tools.
- *F-scores*. In these combined measures, sdhash holds a consistent $1.47 - 1.50\times$ performance advantage over mrsh-v2 and $5.5 - 5.75\times$ over ssdeep.
- *MCC*. The measure also puts sdhash's performance ahead by a $2.25 - 4.5\times$ margin; interestingly, mrsh-v2 and ssdeep swap places suggesting that mrsh-v2's lower precision is having a bigger effect on *MCC* than on *F*-scores.

Analysis of false positives

Let us now consider the false positive behavior of the tested tools in detail. Fig. 1 shows the empirical probability distribution of the approximate matching score S_h scores for which the respective tool has yielded a false positive. Both mrsh-v2 and sdhash show a highly desirable behavior–the FP scores are heavily concentrated close to zero. Indeed, the cumulative probability for scores in the 1–10 range constitute 99.1% and 96.6% of all FP for mrsh-v2 and sdhash, respectively; ssdeep's result are uniformly distributed throughout the 32-85 range.

Analysis of false negatives

The breakdown of false negative results show virtually identical distribution of L_r scores for all three tools. This is clearly due to the overwhelming number of negatives, which render any differences across tools insignificant. The good news is that, although the false negatives are substantial in number, their L_r scores are heavily clustered around zero. This means that we can put a realtively tight and useful bound on what approximate matching tools might miss.

For example, assume that one of the tools, sdhash, returns a score of zero ($S_{sdhash} = 0$). Given its *negative predictive value* $NPV = TN/(FN + TN) = 0.954$, the result will be TN in 95.4% of the time. Whenever it is not (4.6%), Fig. 2 tells us that ~98% of the time the L_r score would not exceed 15. Put together, the two observations tell us that $S_{sdhash} = 0$ implies99.91%[4] certainty that the L_r score does not exceed 15.

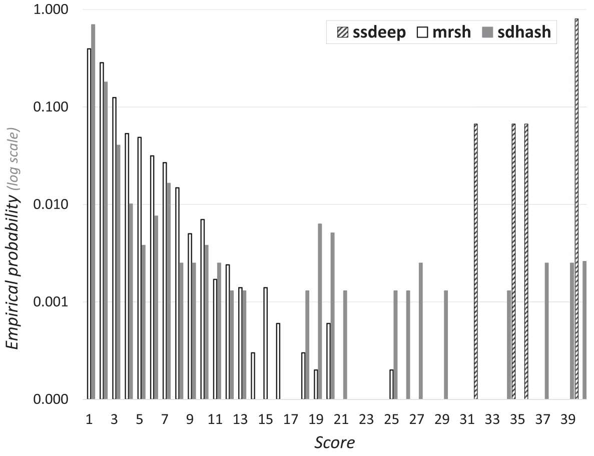

Fig. 1. Empirical probability distribution of L_r scores for approximate matching false positives.

Since the *NPV* for all three tools are similar, we can conclude that negative results from any of the tools are significant in that they allow us to bound the level of commonality that we may be missing with a very high level of certainty.

Analysis of true positives

The next question we would like to explore is–what is the correlation between true positive results (S_h) and the ground truth results (L_r)? To understand this behavior, we build the empirical probability distribution of the *difference* between the true score (as defined by L_r) and the similarity score; that is, $L_r - S_h$, for $h \in \{ssdeep, mrsh, sdhash\}$ (Fig. 3).

We can see that mrsh-v2's comes closest to having a classical Gaussion distribution that is symmetrical and fairly tight around the mean of zero; this implies that mrsh-v2's positive results have about equal chance of

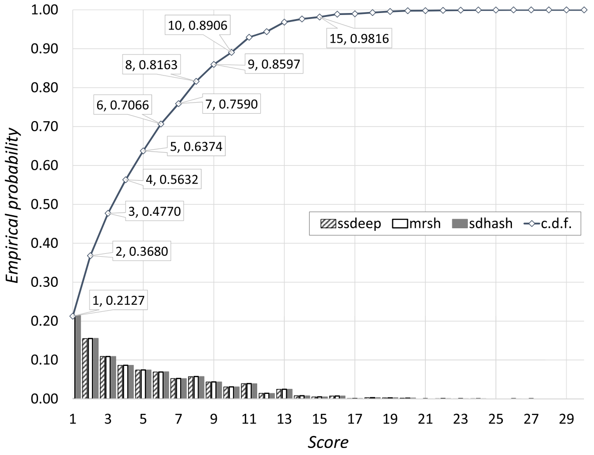

Fig. 2. Empirical probability distribution of L_r scores for approximate matching false negatives.

[4] Out of the 4.6%, there are 98% under 15. Thus, 95.4% + 4.6%· 0.98 = 99.91%.

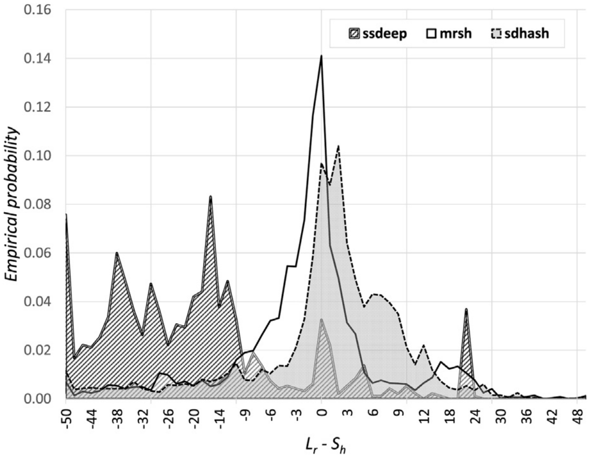

Fig. 3. Empirical probability distributions of $L_r - S_h$, for $h \in$ {*ssdeep*,*mrsh*,*sdhash*}.

being smaller, or larger than ground truth. Next, `sdhash`'s distribution also has a bell-like shape but has a ”bulge” and slight bias to the right of zero, implying that `sdhash` scores are somewhat more likely to be smaller than `alcs`'s score. Finally, `ssdeep`'s distribution is massively skewed to the left of zero (89% of the mass) and shows no particular characteristic shape; still, the graph it tells us that we can view `ssdeep`'s score as an upper bound on the `alcs` result.

Containment vs. resemblance

Having characterized the overall performance of the tools, we consider their behavior under the two basic usage scenarios–*resemblance* and *containment*. Following Broder's ideas (Broder, 1997), we try to approximate the informal notions of ‘roughly the same’ (resemblance) and ‘roughly contained inside’ (containment). For example, comparing two executable files similar in size is likely a resemblance query, whereas comparing a file against a RAM snapshot is clearly a containment query. However, we have no precise guidance as to where to draw the line between the two scenarios.

For this work, we chose a criterion based on the ratio of the file sizes. Namely, if the size of the bigger file is at least *two times* the size of the smaller one, we define this as a *containment* query; otherwise, it is a *resemblance* one. In other words, if more than one non-overlapping copy of one file can fit in the other, we assume the main interest to be *containment*. Evidently, if a similarity tool behaves the same way in both cases, we expect the performance metrics to remain stable across the two scenarios.

Table 5
Basic containment/resemblance statistics by approximate matching tool.

	TP	TP_{ratio}	FP	FP_{ratio}	TN	FN
ssdeep-con	74	0.078	5	0.333	7,382,136	354,840
ssdeep-res	877	0.922	10	0.667	2,089,911	102,343
ssdeep	951	1.000	15	1.000	9,472,047	457,183
mrsh-con	2213	0.602	15,285	0.652	7,366,856	352,701
mrsh-res	1466	0.398	8168	0.348	2,081,753	101,754
mrsh	3679	1.000	23,453	1.000	9,448,609	454,455
sdhash-con	3472	0.634	497	0.629	7,381,644	351,442
sdhash-res	2002	0.366	293	0.371	2,089,628	101,218
sdhash	5474	1.000	790	1.000	9,471,272	452,660

To establish a baseline, we use our ground truth (*gt*) results; Table 4 provides a summary. The first row (*gt-con*) provides the statistics for all containment cases (pairs of files); the second row covers all resemblance cases; the last row combines the results. The first column (*TP*) provides the number of true positives, followed by TP_{ratio} which gives the fraction of true positives for the particular case relative to the total number of true positives. The *TN* and TN_{ratio} provide analogous numbers with respect to true negatives; the last two columns provide totals. In simple terms, we see that about 78% of the pairs fall into the containment and 22% into the resemblance cases.

Tables 5and 6 present the statistics for the evaluated similarity tools. From the former, we can see that `ssdeep` has a notably different behavior from both the baseline and the other two tools. Namely, 92% of its matches come from the resemblance case; this is a logical result of its design, which makes the resolution of the similarity digest a function of file size. As file sizes draw apart, `ssdeep` simply loses the ability to compare them. Both `mrsh-v2` and `sdhash` follow a containment/resemblance ratio that is much closer to the baseline but is still tilted in favor of resemblance as the fraction of resemblance TP results is some 66–81% higher than in the ground truth case.

Considering the information retrieval metrics in Table 6, one important observation is that all three tools yield better results for resemblance over containment. For `mrsh-v2` and `sdhash` the improvement is by 50–100%, while for `ssdeep` it is up to 40 times. Among the tools, the relative performance ratios remain comparable to the ones presented earlier in Table 3 with `sdhash` outperforming across the board.

Conclusion

In this paper, we took on the challenge of *automating* the process of characterizing approximate matching algorithm behavior with respect to standard information retrieval metrics, such as precision and recall, using real world data. The main difficulty in this process is establishing the

Table 4
Ground truth statistics for containment/resemblance cases.

	TP	TP_{ratio}	TN	TN_{ratio}	Total	$Total_{ratio}$
gt-con	354,914	0.775	7,382,141	0.779	7,737,055	0.779
gt-res	103,220	0.225	2,089,921	0.221	2,193,141	0.221
gt	458,134	1.000	9,472,062	1.000	9,930,196	1.000

Table 6
Performance measures by scenario and approximate matching tool.

	Precision	Recall	F_1	F_2	$F_{0.5}$	MCC
ssdeep-con	0.93671	0.00001	0.00002	0.00001	0.00005	0.01361
ssdeep-res	0.98873	0.00042	0.00084	0.00052	0.00209	0.08944
ssdeep	0.98447	0.00010	0.00020	0.00013	0.00050	0.04412
mrsh-con	0.12647	0.00030	0.00060	0.00038	0.00149	0.01834
mrsh-res	0.15217	0.00070	0.00140	0.00088	0.00345	0.03297
mrsh	0.13560	0.00039	0.00078	0.00049	0.00192	0.02232
sdhash-con	0.87478	0.00047	0.00094	0.00059	0.00235	0.08976
sdhash-res	0.87233	0.00096	0.00191	0.00120	0.00476	0.12612
sdhash	0.87388	0.00058	0.00115	0.00072	0.00288	0.09913

ground truth by some algorithmic means. We propose the use of *longest common substring* (LCS) as useful measure of commonality between two files. The problem with using LCS, is that its computation is relatively expensive and cannot be easily scaled to the degree necessary–the digital forensics community needs a testing and evaluation framework that can be routinely deployed by practitioners. The main contributions of this work are as follows:

Efficient LCS approximation. Using classical ideas from data fingerprinting with random polynomials, we derive a linear approximate LCS (aLCS) algorithm that places a lower bound on the size of the lowest common substring. The observed performance shows that aLCS is, indeed, a practical approach estimating the size of LCS for real-world files.

Analytical evaluation framework. We propose an analytical evaluation framework, which quantifies the performance of approximate matching algorithms with respect to the ground truth, and provides a statistical interpretation of tool results.

Analysis of existing approximate matching algorithms. We utilized the framework to evaluate three approximate matching algorithms which have public implementations–`ssdeep`, `sdhash` and `mrsh-v2`. Our results show that a) *recall* rates for all the tools are relatively low–in other words, analysts should not take negative results as solid proof for the lack of similarity; b) *precision* rates for `ssdeep` and `sdhash`, are high, which means that a positive result is a strong indication of commonality at the bytestream level; c) on balance, `sdhash` shows the best overall performance.

We should note that further work is required to relate common substrings to human-observable (and forensically relevant) artifacts. For example, we have not controlled for long strings of sparse data (e.g., all zeros) that, more likely than not, are not of forensic interest. As a next step, we would like to integrate this approach into the approximate matching testing framework FRASH.

Acknowledgments

This work was partly funded by the EU (integrated project FIDELITY, grant number 284862) and supported by CASED (Center for Advanced Security Research Darmstadt).

References

Altheide C, Carvey H. Digital forensics with open source tools: using open source platform tools for performing computer forensics on target systems: Windows, Mac, Linux, Unix, etc1; 2011. Syngress Media.

Baier H, Breitinger F. Security aspects of piecewise hashing in computer forensics. In: IT Security Incident Management & IT Forensics (IMF); 2011. pp. 21–36.

Bloom BH. Space/time trade-offs in hash coding with allowable errors. Commun ACM 1970;13:422–6.

Breitinger F, Baier H. Similarity preserving hashing: eligible properties and a new algorithm MRSH-v2. In: 4th ICST Conference on Digital Forensics & Cyber Crime (ICDF2C); 2012.

Breitinger F, Guttman B, McCarrin M, Roussev V. Approximate matching: definition and terminology http://csrc.nist.gov/publications/drafts/800-168/sp800_168_draft.pdf; 2014.

Breitinger F, Stivaktakis G, Baier H. FRASH: A framework to test algorithms of similarity hashing. In: 13th Digital Forensics Research Conference (DFRWS'13); 2013. Monterey.

Breitinger F, Stivaktakis G, Roussev V. Evaluating detection error tradeoffs for Bitwise approximate matching algorithms. In: 5th ICST Conference on Digital Forensics & Cyber Crime (ICDF2C); 2013. To appear.

Broder AZ. On the resemblance and containment of documents. In: Compression and Complexity of Sequences (SEQUENCES'97); 1997. pp. 21–9. IEEE Computer Society.

Chen L, Wang G. An efficient piecewise hashing method for computer forensics. In: Knowledge Discovery and Data Mining, 2008. WKDD 2008. First International Workshop on; 2008. pp. 635–8.

Gallagher P, Director A. Secure hash Standard (SHS). Technical Report National Institute of Standards and Technologies, Federal Information Processing Standards Publication; 1995. pp. 180–1.

Kornblum J. Identifying almost identical files using context triggered piecewise hashing. Digit Investig 2006;3:91–7.

Menezes AJ, van Oorschot PC, Vanstone SA. Handbook of applied cryptography, 5. CRC Press; 2001.

NIST Information Technology Laboratory. National Software Reference Library http://www.nsrl.nist.gov; 2003–2013.

Noll, LC. (1994–2012). Fnv hash. http://www.isthe.com/chongo/tech/comp/fnv/index.html.

Roussev V. Data fingerprinting with similarity digests. In: Chow K-P, Shenoi S, editors. Advances in digital forensics VI. IFIP Advances in Information and Communication Technology, 337. Berlin Heidelberg: Springer; 2010. pp. 207–26.

Roussev V. An evaluation of forensic similarity hashes. Digit Investig 2011;8:34–41.

Roussev V. Managing terabyte-scale investigations with similarity digests. In: Peterson G, Shenoi S, editors. Advances in digital forensics VIII. IFIP Advances in Information and Communication Technology, 383. Berlin Heidelberg: Springer; 2012. pp. 19–34.

Roussev V, Chen Y, Bourg T, Richard III GG. md5bloom: forensic filesystem hashing revisited. Digit Investig 2006;3:82–90.

Roussev V, Richard III GG, Marziale L. Multi-resolution similarity hashing. Digit Investig 2007;4:105–13.

Sadowski C, Levin G. Simhash: hash-based similarity detection. http://simhash.googlecode.com/svn/trunk/paper/SimHashWithBib.pdf; 2007.

Seo K, Lim K, Choi J, Chang K, Lee S. Detecting similar files based on hash and statistical analysis for digital forensic investigation. In: Computer Science and its Applications, 2009. CSA'09. 2nd International Conference; 2009. pp. 1–6.

Sumathi S, Esakkirajan S. Fundamentals of relational database management systems, 1. Berlin Heidelberg: Springer; 2007.

Tridgell, A. (2002–2009). spamsum. http://www.samba.org/ftp/unpacked/junkcode/spamsum/. Accessed 04.10.13.