Evaluating detection error trade-offs for bytewise approximate matching algorithms

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

[a] *da/sec – Biometrics and Internet Security Research Group, Hochschule Darmstadt, Haardtring 100, 64295 Darmstadt, Germany*[1]
[b] *Department of Computer Science University of New Orleans, Lakefront Campus, 70148 New Orleans, LA, USA*[2]

Introduction

One of the critical requirements of modern (digital) forensic investigations is the ability to perform large-scale automated filtering and correlation of data. The most common method deployed for this purpose is *known file filtering*, which consists of computing the crypto hashes of files on a forensic target and comparing them to a reference database. A match can be used to either filter out (eliminate from consideration) a known good file (perhaps from the NSRL (NIST Information Technology Laboratory, 2003–2013) data set maintained by NIST), or to focus attention on a known bad file (e.g., from a contraband/malware database).

Positive results from above process are inherently trustworth due to the collision resistant properties of crypto hashes. However, the main problem is that, for the process to work robustly, we need to maintain an up-to-date database of all versions of all files of interest. That is increasingly impractical as both data and code are becoming ever more dynamic. Therefore, it is useful to have algorithms that provide *approximate* matches that can correlate closely related versions of data objects.

Generally, an approximate matching scheme works by extracting features from data objects and storing them as digests. The digests are then compared for commonality and the algorithm produces an estimate of the level of correlation between the original objects. The method by which features are picked and interpreted defines the type of approximate matching in use, which generally falls into

* Corresponding author. Tel.: +49 6151 70 471.
E-mail addresses: frank.breitinger@cased.de (F. Breitinger), georgios.stivaktakis@cased.de (G. Stivaktakis), vassil@roussev.net (V. Roussev).

[1] https://www.dasec.h-da.de.
[2] http://roussev.net.

one of three categories: bytewise–, syntactic– or semantic approximate matching.

Semantic matching operates at the highest level of abstraction and provides results that are closest to human perceptual notions of similarity. For example, facial recognition methods could correlate images of the same person. Semantic matching methods tend to be the most specialized and computationally expensive, and work across different representations (file formats) of the object.

Syntactic matching relies on purely syntactic rules to break up the data representation into features. Among the simplest examples are splitting the data fixed-sized (e.g., disk blocks), or variable-sized (network packet payloads) pieces and treating the pieces as features. Syntactic matching requires basic knowledge of the syntactic structure of the data but it does not actually interpret the data.

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. It is the most general in that it can compare *any* two blobs of binary data and relies on the assumption that similarities between objects are reflected by similarities in their byte level representation. As we discuss later, there are a number of very common scenarios where this is, indeed, the case.

In this work, we are concerned only with bytewise approximate matching algorithms and their evaluation. Currently the best-known algorithms are `ssdeep` (Kornblum, 2006) and `sdhash` (Roussev, 2010) and we will use them as a case study. Prior work from Roussev (2011) has argued that `sdhash` outperforms `ssdeep` in terms of a variety of metrics. This work is an effort to standardize such evaluation and make it possible for researchers and practitioners to easily reproduce the test results and compare tools, both current and future. Indeed, there are already other efforts, such as `mrsh-v2` (Breitinger and Baier, 2012), to develop approximate matching tools, which reinforces the need for having an open and extensible platform for testing and evaluation.

The specific aim of this work is to extend FRASH (Breitinger et al., 2013) by developing precision and recall tests that enable us to characterize the behavior of approximate matching algorithms using a detection error trade-off (DET) curve (a.k.a. ROC). For these tests, we use controlled (pseudo-)random data that allows us to precisely generate and manipulate the test targets and know the exact ground truth. The rest of the paper is organized as follows: Section 2 introduces the necessary background and terminology, and discusses related work; Section 3 details the design and implementation of the precision and recall testbed; and Section 4 presents the experimental results of applying our testing methodology to existing approximate matching algorithms. Section 5 concludes the paper.

Background & related work

Hash functions are popular across 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). Mostly hashing is associated with cryptographic hash functions (e.g., SHA-1 from Gallagher and Director (1995)) which fulfill some security requirements and thus imply the avalanche effect: no matter how similar two inputs are–the hash values of non-identical inputs differ by approximately 50% of their bits. As a consequence the community came up with new approaches called *approximate matching* which gets more and more important (Garfinkel, 2010) especially in digital forensics.

Bytewise approximate matching algorithms

At present, there are two algorithms with mature implementations that are in use in the forensic field–`ssdeep` and `sdhash`. Below, we provide a brief sketch of their operation; a detailed description is beyond the scope of this paper as we treat them black boxes for testing purposes.

`ssdeep`

`ssdeep`–also known as *context triggered piecewise hashing* (CTPH)–is the first and probably best-known approximate matching algorithm which was presented by Kornblum in 2006. It is based on the spam detection algorithm from Tridgell (2002–2009). The implementation is freely available and currently in version `ssdeep` 2.9.[3]

The overall idea of `ssdeep` is a version of Rabin's (1981) seminal work on data fingerprinting by random polynomials. CTPH identifies trigger points to divide a given byte sequence into chunks. Each chunk is hashed using FNV (Noll, 1994–2012). Instead of using the complete FNV hash, CTPH only takes the least significant 6 bits which is equal to a Base64 character. In order to generate a final fingerprint, all Base64 characters are concatenated. To determine the distance of two fingerprints, they are treated as text strings and compared using the weighted edit distance. The match score is scaled between 0 and 100.

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 implementation have been made available for public testing and evaluation.

`sdhash`

`sdhash`,[4] proposed by Roussev (2010), uses a completely different algorithm that attempts to pick characteristic features for each object that are unlikely to be appear by chance in other objects which is the result from an empirical study. In the baseline implementation, each feature is hashed with SHA-1 and inserted into a Bloom filter (Bloom, 1970) where a feature is a sequence of 64 bytes. The signature of the data object, called a *similarity digest*, is a sequence of 256-byte filters, each of which represents approximately 10 KB of the original data, on average.

Subsequently, a block-aligned version was developed (Roussev, 2012), in which fixed-size blocks (16 KiB by default) are mapped to each 256-byte filter. Although the

[3] http://ssdeep.sourceforge.net (last accessed 11 Feb. 2014).
[4] http://sdhash.org (last accessed 11 Feb. 2014).

two versions are compatible (the two version of the digests can be meaningfully compared) we did not consider the block version in our study as it requires additional parameters.

FRASH

FRASH is an open source, extensible framework for testing bytewise approximate matching algorithms; it is implemented in Ruby and is freely available at https://www.dasec.h-da.de/staff/breitinger-frank/. The existing implementation includes test cases for *efficiency* and *sensitivity and robustness*.

The efficiency tests evaluate tools with respect to three criteria:

Generation efficiency measures the execution time taken by the algorithm to process input of a given size and generate the similarity digest.
Comparison efficiency measures the execution time for performing a given set of digest comparisons.
Space efficiency (compression) measures the ratio between the input length and the size of the similarity digest.

The sensitivity and robustness tests use the following evaluation scenarios:

Single-common-block correlation (sensitivity) calculates the smallest object that two files need to have in common for which the algorithm reliably correlates two targets.
Fragment detection (sensitivity) quantifies the smallest fragment for which the similarity tool reliably correlates the fragment and the original file.
Alignment robustness analyzes the impact of inserting byte sequences at the beginning of an input by correlating the size of the change to changes in the comparison output.
Random noise resistance analyzes the impact of random input edits on the correlation capabilities of the algorithm.

FRASH performes the above input manupulations in a randomized fashion over numerous runs in order to obtain statistically useful results; a detailed description of these tests is given by Breitinger et al. (2013).

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{\{\text{retrieved documents}\}}$$

Terminology

Our terminology closely follows standard information retrieval treatment of precision and recall.

Definitions

Genuines Two, or more files, defined to be similar.
Impostors Two, or more files, defined to be non-similar.
Score (*s*) is the output of comparing two similarity digests and is a number between 0 and 100.
Threshold (*t*) of significance A score paremeter, used to separate matches from non-matches.
Match A score s, with $s \geq t$.
Non-Match A score s, with $s < t$.
True positive (TP) A match of two genuines.
True negative (TN) A non-match of two impostors.
False positive (FP), (false match) A match of two impostors.
False negative (FN), (false non-match) A non-match of two genuines.
False positive rate (FPR) False match count divided by file count.
False negative rate (FNR) False non-match count divided by file count.

Error types

Approximate matching algorithms can make false decisions, i.e., output a similarity scores also the input files are not similar and vice versa. Table 1 provides a summary of the possible classification outcomes from approximate matching comparisons.

Whether a comparison result is erroneous or depends, in part, on the chosen threshold t. For instance, if $t = 20$ and the score of two similar files is 19, the input files are falsely classified as non-similar (a false negative); however, for $t = 15$, the classification would be a true positive.

Ideally, an approximate matching algorithm would produce results such that there is no overlap between true positives and false positives. In other words, it would be possible to identify a threshold such that for all scores $s \geq t$ the results are true positives, and for $s<t$ all results are true negatives. In other words, there would be no overlap between the scores.

In practice, the choice of t is typically a trade-off between true and false positives as a function of t—a lower value for t can increase the TP rate at the expense of higher FP rate and vice versa.

Precision and recall

Precision and recall are standard measures in information retrieval to evaluate the effectiveness of query results. Precision is defined as the ratio of the number of relevant records retrieved to the total number of records retrieved:

Table 1
Possible results of a comparison.

Match	Similar files	Non-similar files
Yes	true positive (tp)	*false positive* (fp)
No	*false negative* (fn)	true negative (tn)

Recall is defined as the ratio of the number of relevant documents retrieved to the total number of relevant documents:

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{\{\text{relevant documents}\}}$$

We could also express these term using true and false positives:

$$\text{precision} = \frac{tp}{tp + fp}, \quad \text{recall} = \frac{tp}{tp + fn}.$$

Methodology and implementation

Like FRASH, our precision & recall extension is implemented in Ruby. The new class is called `SyntheticDataPrecisionRecallTest` and is responsible for the test-set creation, test execution, and plotting the performance graphs.

Test methodology

Generally, the tool needs to first create a set of unique files (using random bytes), then create versions of them using one of the four modification methods, run the actual comparisons and summarize the results.

Depending on the desired test scope, the tool has two main configuration parameters called *file-count* and *runs*. The former parameter is the number of all input files; the latter specifies the number of independent test runs to be executed. A single run comprises the following steps:

1. Create a file set, denoted by *test-set*, of *file-count* number of files created with `SecureRandom.random_bytes`.
2. Apply a file manipulation method and option to each of the generated files and create versions with known level of similarity. The single-common-block correlation test requires a bit more elaborate setup so it uses custom test generation code, the rest is done in a generic fashion.
3. Run the approximate matching algorithms comparing all pairs of files across the two sets—originals and modified copies—and return a match-string comprising of `file name 1`, `file name 2` and a `match score`, e.g., `readme report 000`.
4. The match scores are aggregated and the results are used to compute the overall false positive and false negative rate per test and option.

To enable the framework to distinguish true positives from false positives, corresponding files have equal file names but different folders. For instance, all originals are in the `left`-folder and numbered consecutively. The modified files are located in the `right`-folder, which is overwritten by every manipulation. Thus, a possible comparison is `left/0 | right/0 | 100` which is a true positive.

The size of the created files is fix but adjustable. Currently, we use the following 6 file sizes: 1, 4, 16, 64, 256 and 1024 KiB.

In terms of execution time, having a set of *file-count* files results in *file-count*^{2} comparisons. Hence, the total number of comparisons per algorithm is calculated by *file-count*$^2 \cdot runs \cdot o$ where o is the number of all options (see Section 3.2).

Test set manipulations

Our tool utilizes the four generic data manipulation techniques that are already integrated in FRASH where each has different options:

Fragment detection: f_2 is a fragment of f_1 where the size of f_2 is one out of X={50%,40%,30%,20%,10%,5%,4%,3%,2%,1%}.
Single-common-block correlation: f_1 and f_2 have equal size and share a common byte string (block) of size X={50%,40%,30%,20%,10%,5%,4%,3%,2%,1%}. The position of the block is chosen randomly for each file.
Alignment robustness: f_2 is a copy of f_1, prefixed with a random byte string of length X={1%,5%,10%,20%}.
Random-noise resistance: f_2 is an obfuscated version of f_1, i.e., X% of f_2's bytes are edited, where X={0.5%,1.0%,1.5%,2.0%,2.5%} of the file size. The percentage boundaries were determined experimentally, as `ssdeep` and `sdhash` both did not detect two files as similar, when ≈2.5% of the bytes were manipulated in one file.

The term option is the combination of a test and a specific setting, e.g., alignment 1% or fragment 10% are valid options. Hence, there are 29 different options.

Implementation

In addition to the standard installation of Ruby, FRASH needs the following packages (gems), which can be installed using the `gem install <GEMNAME>` command: actionpack, activesupport, i18n, activemodel, rack, erubis, parallel, colored and terminal-table. The graphs are plotted using Gnuplot.[5]

`SyntheticDataPrecisionRecallTest` inherits from `BaseTest` and overwrites the `@gen_compare_command` variable, so that the tools also return zero-comparison matches. The histogram method performs the comparisons and stores the match strings in an array. Then, it calls the `separate_genuines_from_impostors` method, which traverses every match string and assigns the scores to the genuines or impostors. Recall, a match string contains original file name, modified copy file name, and match score. Hence, identical file names reveal genuines and varying file names show impostors. At the end we compute the frequency of all scores and store it in `@histogram`. The histogram is the foundation for calculating the error probability distribution and detection error trade-off curve.

Case study: `ssdeep` and `sdhash`

In this section we present the results from applying our test methodology to analyze the performance of `ssdeep`

[5] http://www.gnuplot.info (last accessed 11 Feb. 2014).

and sdhash. As the results are very comprehensive, we only present a selection. However, all details are available online.[6]

Our test examines the performance of the algorithms as a function of file size. For that purpose we consider the behavior at six fixed file sizes—1, 4, 16, 64, 256 and 1024 KiB. We decided for these size boundaries after analyzing the sizes of almost 1,000,000 files in the govdoc-corpus.[7] As shown in Table 2, nearly 91% of all files are smaller than 1MiB.

To avoid library-level integration with the tools (which would facilitate efficiency but introduce tight code dependencies) we found it necessary to use the command line interface provided by the tool. Thus, even though each tool invocation completes in a fraction of a second, there is considerable overhead that adds up on a large scale.

One challenge that may not be immediately obvious is the amount of computation needed to complete the tests. For this test we did 10 different runs on 100 files (vs 100 modified versions) results in 100,000 comparisons per option which ends up in 2,900,000 comparisons per file size and algorithm. Due to 6 different file sizes, we had 17.4 million comparisons per algorithm, which is in total 34.8 million comparisons.

To make the code run in a reasonable amount of time, we have parallelized the implementation so it can take advantage of multi-core architectures. In our tests, we utilized two 2.6 GHz AMD Opteron, 48-core Ubuntu server, which ran the above test suite in 15 h, or about 320 comparisons/second.

The approximate matching comparisons produced by these tools yield a (match) score, which is a number between 0 and 100. Despite its range, this value is *not* an estimate of percentage commonality between the compared objects but a level of confidence. It is meant to serve as a means to sort and filter the results.

The rest of this chapter is divided into three parts. First, we explain the representation of our results in Section 4.1. Section 4.2 presents the average results over all file sizes and tests. As this is very general, we decided to present some more details about specific options in Section 4.3.

Result presentation

We use three types of graphs to visually summarize the results of our study—score histograms, error probability distributions, and detection error trade-off curve (DET curve).

Score histograms

Show the frequency for each score value, i.e., how often a score occurs, for both true and false positives. Ideally, scores for known impostors would be all zero, while genuine scores would be positive. More precisely, we need impostor scores to be generally lower than genuine score and we could use the threshold parameter as means to clearly separate them. The score histogram is a convenient means to identify suitable threshold values.

Error probability distributions

Show the *false positive rate (FPR)* and *false negative rate (FNR)* as a function of the chosen threshold value. More formally, let t be the threshold where $0 \leq t \leq 100$ and s denote the comparison score. Then:

- $FPR(t)$ is the number of impostor comparisons with $s \geq t$ divided by the total number of impostor comparisons.
- $FNR(t)$ is the number of genuine comparisons with $s<t$ divided by the total number of genuine comparisons.

Detection error trade-off curve

Correlates the FPR (x-axis) and the FNR (y-axis). Thus, we can answer the question *if at most x% false matches shall be tolerated, how many false non-matches must be expected?* Obviously, the more false matches are tolerated, the less false non-matches can be expected and vice versa.

Table 2
File sizes distribution in the govdoc-corpus (min size is 1 KiB).

File size range (KiB)	≤4	≤16	≤64	≤256	≤1024
Amount (%)	5.40	20.71	52.54	75.82	90.60

Averaged results per file size

Recall, our test considers the behavior for the file sizes 1, 4, 16, 64, 256 and 1024 KiB. In the following we decided to show the details for 256KiB which is a very common size for files (the numbers are averaged results over all options).

Score histogram

The score histograms for ssdeep and sdhash are given in Figs. 1 and 2, respectively. In case of ssdeep we do not have any impostors with a score above 0 (which is perfect). With respect to sdhash, there are only a few impostors with a score above 0 but all below 8. With regards to the false negatives, the absolute value of ssdeep (19,645) is higher than for sdhash (3849).

Error probability distributions

Figs. 3 and 4 show the error probability distribution for ssdeep and sdhash. The false positive rate of ssdeep is 0. sdhash has a FPR of 10^{-4} which is very close to 0. Hence, both algorithms are nearly working perfect. With respect to false negative, ssdeep has an initial rate of 65% for t=1 whereas sdhash performs pretty well with a rate of approximately 3%. The reason why the sdhash FNR rate does not go up to 100 due to the fact that we still have a lot of 100 scores after all modifications, e.g., fragment detection or alignment.

Detection error trade-off curve

Figs. 5 and 6 show the DET curves for sdhash and ssdeep for 256 KiB and 1024KiB files, respectively, where

[6] https://www.dasec.h-da.de/staff/breitinger-frank/#downloads (last accessed 11 Feb. 2014).

[7] "These documents were obtained by performing searches for words randomly chosen from the Unix dictionary, numbers randomly chosen between 1 and 1 million, and randomized combinations of the two, for documents of specified file types that resided on web servers in the .gov domain using the Yahoo and Google search engines" (http://digitalcorpora.org/corpora/files).

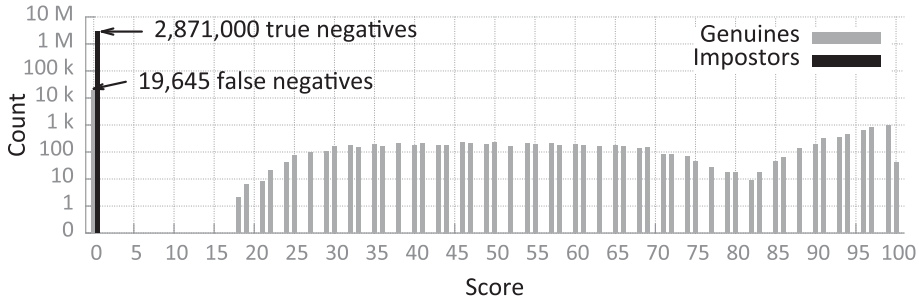

Fig. 1. Histogram for `ssdeep` (256 KiB files).

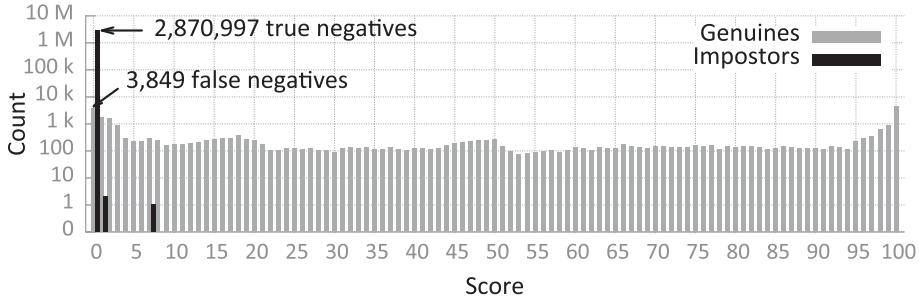

Fig. 2. Histogram for `sdhash` (256 KiB files).

the detection rates are nearly the same. Note, the scale on the *x*-axis is optimized for a better view. In case of `ssdeep` there are plenty false negatives but no false positives. Thus, the graphs drops straight to the *x*-axis (we manually set the point 0.000034). `sdhash` has less false negatives but therefore a few false positives: 7 for 256 KiB and 3 for 1024 KiB (indicated by the dots on the line).

Due to the significant amount of less false negatives, we rate `sdhash` as better than `ssdeep`. However, both algorithms perform very well on random data.

Detailed test results

In this section, we present the results for each of the four types of data manipulation we use to evaluate the tools. In all cases, we use test data of 1, 4, 16, 64, 256, and 1024 KiB in size to obtain a consistent sample of the tools' behavior. In addition to the numerical scores, we discuss the relationship between the observed behavior and the design of the algorithms.

Fragment detection

In this test, we evaluate the ability of the algorithm to find a pieces of target. Specifically, we compare a random test file f_1 with a randomly sampled fragment f_2; the size of f_2 taken to be X% of that of the original, where X= {1%,2%,3%,4%,5%,10%,20%,30%,40%,50%}. Table 3 shows the results.

Observations.

- The behavior of `ssdeep` is consistent across all file sizes: above 0.96 for the 50% case, 0.68–0.70 at 40%, and (near) zero in all other cases.

Considering the algorithm, these observations make sense—the algorithm produces a fixed-size signature, which means that, in *relative* terms, it maintains the same resolution, so the minimum detectable fragment should be defined in relative terms; the tests clearly capture this feature.

- The behavior of `sdhash` is more dynamic; it starts with complete failure at 1 KiB but quickly improves as the file size grows: at 4KiB, it needs a 30% sample for near-

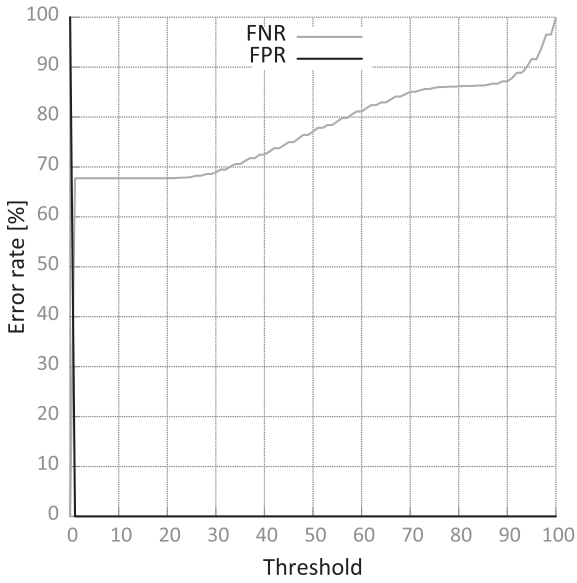

Fig. 3. FPR and FNR for `ssdeep` (256 KiB files).

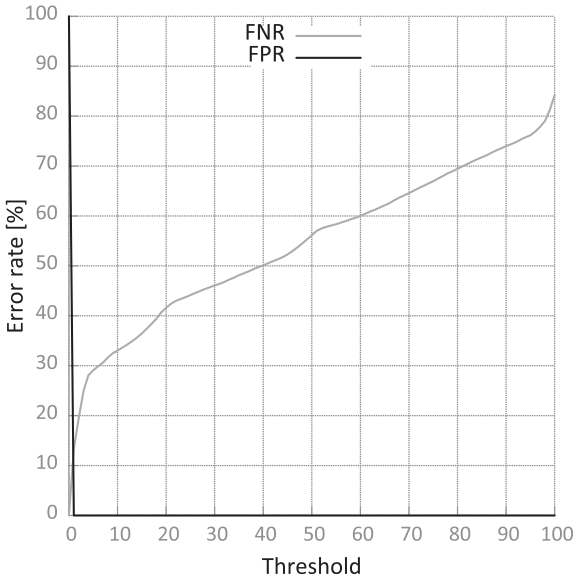

Fig. 4. FPR and FNR for `sdhash` (256 KiB files).

perfect detection, whereas at 256 KiB and up, even a 1% sample is detected perfectly.

This behavior should be expected—the tool uses a variable-sized digest so a fragment of any size above the design minimum should be detectable, regardless of the size of the source file. Considering—in *absolute* terms—the size of the fragment at which it becomes perfectly detectable, we can see that it is approximately the same in all cases: 30% × 4 KiB ≈ 10% × 16 KiB ≈ 2% × 64 KiB.

- Finally, both tools exhibit bi-modal behavior—if the parameters fall within their design space, the results are (near) perfectly correct, otherwise, they are perfectly wrong; there is not much in between. The good news is that FRASH clearly delineates the two cases, giving analysts an important guide on how to use them.

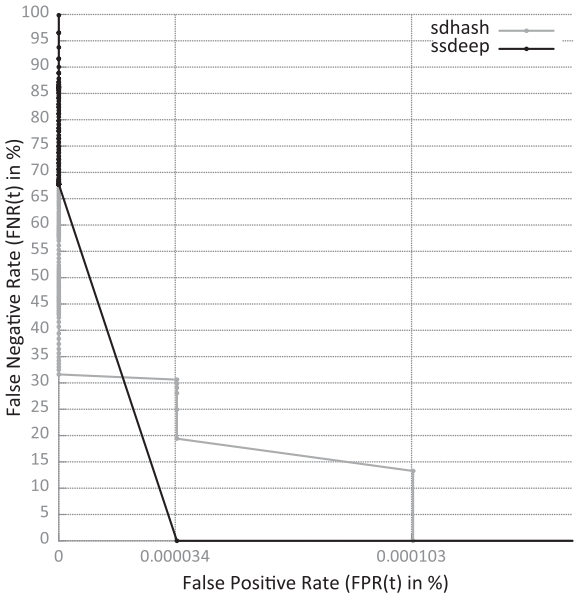

Fig. 5. DET curves for `sdhash` and `ssdeep` (256 KiB files).

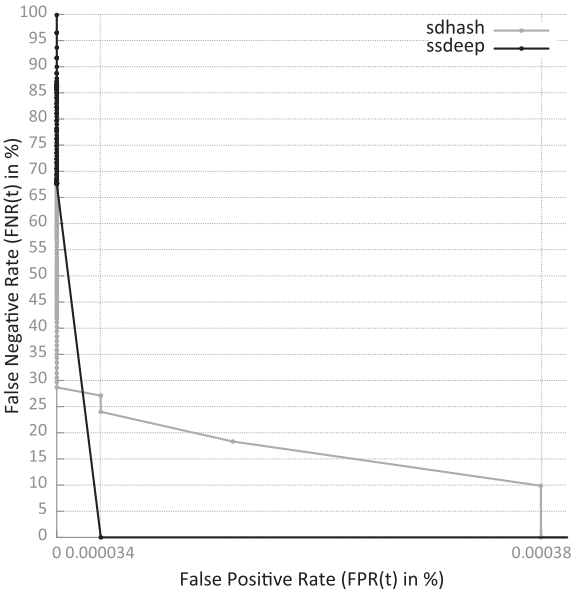

Fig. 6. DET curves for `sdhash` and `ssdeep` (1024 KiB files).

Single-common-block correlation

An extension of the fragment test, the single-common-block test evaluates the ability of an approximate matching algorithm to correlate two files, f_1 and f_2, that are known to have fragment in common. To simplify the analysis and presentation, we choose the files of equal size and vary the amount of commonality as a fraction X of the file size; X= {1%,2%,3%,4%,5%,10%,20%,30%,40%,50%}. Table 4 shows the results.

Observations.

- The behavior of the tools on this test is clearly correlated with their performance on the prior test but the relationships are a bit more complex.
- For `ssdeep`, we need at least 20% commonality to achieve a better-than-50% TPR, and at least 30% to achieve reliable (95%+) detection. As with the fragment detection, the results are consistent across file sizes and depend on the *relative* size of the common fragment.
- For `sdhash`, the usual problems at 1 KiB persist and, just like in the fragment case, the performance improves as the size of the source file grows. Considering the *absolute* size of the common fragment is a good guide but we find that the threshold that gives perfect classification varies between 13 and 20 KiB. This is likely due to alignment issues as the `sdhash` signature consist of a sequence of Bloom filters, such one representing (on average) 9—10 KiB of the source data. Depending on the location of the sample, the data could map to two, or three, separate filters and match could be numerically diluted during comparison.

Table 3
True positive rates for ssdeep and sdhash as a function of file and fragment size.

Fragment size (%)		1%	2%	3%	4%	5%	10%	20%	30%	40%	50%
ssdeep	1 KiB	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.67	0.97
	4 KiB	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.04	0.70	0.98
	16 KiB	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.04	0.68	0.97
	64 KiB	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.05	0.69	0.98
	256 KiB	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.03	0.69	0.96
	1024 KiB	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.03	0.70	0.97
sdhash	1 KiB	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	4 KiB	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.99	1.00	1.00
	16 KiB	0.00	0.00	0.00	0.00	0.05	1.00	1.00	1.00	1.00	1.00
	64 KiB	0.00	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	256 KiB	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	1024 KiB	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Alignment

Recall that the alignment test is designed to evaluate the robustness of the algorithm with respect to data alignment. To perform the test, we compare an original file f_1 to a file f_2, which consists of the entirety of f_1 prefixed by a random byte string of length X={1%,5%,10%,20%}.

As it turns out, both tools deal well with this case and give perfect true positive (100%) and perfect true negative (0%) rates. The only exception is the sdhash case with 1 KiB of data (which we refer to as the *sdhash-1k* case), which yields TP rate in the 0.66–0.69 range. This is an edge case for the tool, which is optimized for targets with minimum size of about 1400 bytes (full network packet).

Random-noise resistance

In the random-noise resistance test, we attempt to correlate a file f_1 and a randomly disturbed version of it–f_2. In addition to file size, we vary the fraction X of bytes modified, where X={0.5%,1.0%,1.5%,2.0%,2.5%} of f_1's size. Table 5 presents the results.

Observations.

- In prior work by Baier and Breitinger (2011), we have shown that ssdeep's noise resistance is a function of the absolute number of changes made to the source data. The current results confirm this—ssdeep performs well for small files and small percentage values. For instance, 0.5–1% of 1024 Bytes leads to 5–10 changes; ssdeep can clearly tolerate the disturbance as evidenced by the true positive rates of 99.5% and 93.3%. Somewhere around 20–22 modifications the TPR drops below 50%, and around 80 it becomes effectively zero.
- Apart from the always-problematic 1 KiB case, sdhash deals well with random noise of up 1.0% for all file sizes—the true positive rate is 100%. The 1.5% case triggers a fluctuating detection rate between 73% (4 KiB) and 98% (256 KiB) and shows that the tool is starting to fail. At 2.0% and up, the TPR crashes to zero indicating that the tool's ability to tolerate noise has been exhausted.

Conclusion

In this paper, we considered the problem of characterizing approximate matching algorithm behavior with respect to precision and recall rates. We argued that the digital forensics community needs an open set of standardized tests that can be used for tool evaluation.

We designed a set of precision and recall tests that are algorithm-neutral and implemented them as an extension of the FRASH framework. To validate the approach and implementation, we conducted a case study of the behavior of ssdeep and sdhash and presented the resulting observations.

The results are broadly in agreement with prior efforts, however, our work shows in more detail the tools' performance under different scenarios and allowed us to *quantify* the relationship between tool design and real-world performance. Further, we showed that each of the algorithms

Table 4
True positive rates for ssdeep and sdhash as a function of file and common block size.

com. block size (%)		1%	2%	3%	4%	5%	10%	20%	30%	40%	50%
ssdeep	1 KiB	0.00	0.00	0.00	0.00	0.00	0.06	0.58	0.94	0.99	1.00
	4 KiB	0.00	0.00	0.00	0.00	0.01	0.07	0.62	0.95	0.99	1.00
	16 KiB	0.00	0.00	0.00	0.01	0.04	0.07	0.62	0.94	0.99	1.00
	64 KiB	0.00	0.00	0.00	0.00	0.01	0.07	0.62	0.94	0.99	1.00
	256 KiB	0.00	0.00	0.00	0.01	0.01	0.07	0.64	0.95	0.99	0.99
	1024 KiB	0.00	0.00	0.00	0.00	0.04	0.08	0.64	0.95	0.99	1.00
sdhash	1 KiB	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.37	0.52
	4 KiB	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.16	0.97	0.99
	16 KiB	0.00	0.00	0.00	0.00	0.00	0.00	0.59	0.86	0.97	0.99
	64 KiB	0.00	0.00	0.00	0.02	0.13	0.81	1.00	1.00	1.00	1.00
	256 KiB	0.08	0.42	0.79	0.95	1.00	1.00	1.00	1.00	1.00	1.00
	1024 KiB	0.32	0.95	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 5
True positive rates for `ssdeep` and `sdhash` as a function of file size and the random noise.

com. block size (%)		0.5	1.0	1.5	2.0	2.5
`ssdeep`	1 KiB	0.99	0.93	0.77	0.55	0.38
	4 KiB	0.62	0.16	0.05	0.01	0.00
	16 KiB	0.02	0.00	0.00	0.00	0.00
	64 KiB	0.00	0.00	0.00	0.00	0.00
	256 KiB	0.00	0.00	0.00	0.00	0.00
	1024 KiB	0.00	0.00	0.00	0.00	0.00
`sdhash`	1 KiB	0.61	0.56	0.33	0.13	0.31
	4 KiB	1.00	0.99	0.73	0.09	0.03
	16 KiB	1.00	1.00	0.82	0.04	0.00
	64 KiB	1.00	1.00	0.96	0.02	0.00
	256 KiB	1.00	1.00	0.98	0.01	0.00
	1024 KiB	1.00	1.00	0.87	0.00	0.00

has a distinct operational range and analysts must understand the relationships between input parameters and result significance in order to operate the tools correctly. Therefore, having a rigorous testing framework, such as FRASH, is critical to evaluating and calibrating various approximate matching algorithms.

We expect that further tools in the approximate matching family will be developed, and the presented framework is designed to accommodate them with minimal effort allowing for fast, objective, and repeatable evaluation.

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, etc, vol. 1. Syngress Media; 2011.

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

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.

Garfinkel SL. Digital forensics research: the next 10 years. Digit Investig 2010;7:64–73.

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, vol. 5. CRC Press; 2001.

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

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

Rabin MO. Fingerprinting by random polynomials. Technical Report TR1581 Center for Research in Computing Technology. Massachusetts: Harvard University Cambridge; 1981.

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, vol. 337. Springer Berlin Heidelberg; 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, vol. 383. Springer Berlin Heidelberg; 2012. pp. 19–34.

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, vol. 1. Springer Berlin Heidelberg; 2007.

Tridgell A. spamsum [accessed 10.04.13], http://www.samba.org/ftp/unpacked/junkcode/spamsum/; 2002–2009.