FRASH: A Framework To Test Algorithms Of Similarity Hashing

By

Frank Breitinger, Georgios Stivaktakis and Harald Baier

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2013 USA

Monterey, CA (Aug 4th - 7th)

DFRWS is dedicated to the sharing of knowledge and ideas about digital forensics research. Ever since it organized the first open workshop devoted to digital forensics in 2001, DFRWS continues to bring academics and practitioners together in an informal environment.

As a non-profit, volunteer organization, DFRWS sponsors technical working groups, annual conferences and challenges to help drive the direction of research and development.

http:/dfrws.org

Digital Investigation 10 (2013) S50–S58

Contents lists available at SciVerse ScienceDirect

Digital Investigation

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

FRASH: A framework to test algorithms of similarity hashing

Frank Breitinger*,[1], Georgios Stivaktakis [1], Harald Baier [1]

da/sec – Biometrics and Internet Security Research Group, Hochschule Darmstadt, Haardtring 100, 64295 Darmstadt, Germany

Keywords:
Digital forensics
Similarity hashing
Test framework
ssdeep
sdhash

A B S T R A C T

Automated input identification is a very challenging, but also important task. Within computer forensics this reduces the amount of data an investigator has to look at by hand. Besides identifying exact duplicates, which is mostly solved using cryptographic hash functions, it is necessary to cope with similar inputs (e.g., different versions of a file), embedded objects (e.g., a JPG within a Word document), and fragments (e.g., network packets), too. Over the recent years a couple of different similarity hashing algorithms were published. However, due to the absence of a definition and a test framework, it is hardly possible to evaluate and compare these approaches to establish them in the community.

The paper at hand aims at providing an assessment methodology and a sample implementation called *FRASH:* a framework to test algorithms of similarity hashing. First, we describe common use cases of a similarity hashing algorithm to motivate our two test classes *efficiency* and *sensitivity & robustness*. Next, our open and freely available framework is briefly described. Finally, we apply FRASH to the well-known similarity hashing approaches `ssdeep` and `sdhash` to show their strengths and weaknesses.

1. Introduction

The handling of terabytes of data is a major challenge in today's IT forensic investigations. It is important to *automatically* reduce the amount of data that needs to be inspected manually by either removing non-relevant objects like operating system files or marking suspect files like company secrets or child pornography.

Identifying exact duplicates is often solved using cryptographic hash functions. However, it is also helpful to have more flexible and robust algorithms that allow *similarity detection* (e.g., different versions of a file), *embedded object detection* (e.g., JPG in a Word document), *fragment detection* (e.g., analyzing a device on the byte level or network packages) or *clustering files* (e.g., e-mails and Word documents with similar content).

As a consequence the community came up with similarity hashing, which either operates on the *byte level* or on the *semantic level* (e.g., to decide about the similar perception of pictures). Both levels feature their respective strengths and weaknesses. For instance, in the former case an active adversary can circumvent detection by changing the format of a multimedia file or zip it. However, byte level approaches offer fragment and embedded object detection.

In the following we focus on byte level similarity and thus two inputs are equal/similar if they share common byte sequences. This topic has become more and more visible in the community, e.g., Garfinkel (2010) addresses this as one of the candidates to solve the signature metrics abstraction problem.

In general establishing a new algorithm requires a thorough assessment by the community on base of well-known criteria. For instance, the US National Institute of Standards and Technology (NIST) governed the process to standardize

* Corresponding author.
E-mail addresses: frank.breitinger@cased.de (F. Breitinger), georgios.stivaktakis@cased.de (G. Stivaktakis), harald.baier@cased.de (H. Baier).
[1] URL: dasec.h-da.de (Frank Breitinger, Georgios Stivaktakis and Harald Baier).

1742-2876/$ – see front matter
http://dx.doi.org/10.1016/j.diin.2013.06.006

the new symmetric block cipher AES (Nechvatal et al., 2000) or the cryptographic hash function SHA-3 Keccak (Bertoni et al., 2009). Hence, similarity hashing will only be accepted by both the scientific community and practitioners if an assessment methodology and a test framework are available (Garfinkel, 2010; Dewald and Freiling, 2012).

Our contribution within this paper is to provide a test framework, which evaluates existing similarity hashing algorithms. We call it FRASH: a *FR*amework to test *A*lgorithms of *S*imilarity *H*ashing. FRASH is open source and freely available online.[2] On the one hand FRASH is inspired by previous work on 'eligible properties' of similarity hashing algorithms (Breitinger and Baier, 2012d). On the other hand we analyzed multiple papers and how the authors evaluate and compare similarity hashing, e.g., Roussev (2011); Sadowski and Levin (2007); Tridgell (2002–2009). The result of our analysis yields several test cases, which we group in two classes: *efficiency* and *sensitivity & robustness*.

The first class measures the runtime efficiency and the compression rate of the algorithms. Efficiency is important for practical reasons as the computation and storage amount must meet practical needs. The second class addresses sensitivity & robustness issues like random-noise-resistance, alignment robustness, fragment detection, and file correlation. FRASH assesses a similarity hashing algorithm and uncovers its strengths and weaknesses in normal operation and when under attack, respectively.

Currently ssdeep and sdhash are the best-known algorithms. We therefore make use of FRASH to assess them. Our results show that sdhash is superior to ssdeep in all categories except for compression.

The rest of the paper is organized as follows: In Section 2 we discuss the state of the art and relevant literature. In addition, we explain multiple similarity hashing functions and their usage in digital forensics. The scope of FRASH is explained in Section 3. Section 4 provides details about the implementation itself, which is followed by some experimental results in Section 5. Finally, we conclude the paper and point to future work in Section 6.

2. Background

Nowadays a popular use case of cryptographic hash functions within computer forensics is detecting known inputs. The proceeding is quite simple: hash all files on a storage medium and compare the hashes to a reference database. In case of a match, the investigator is convinced that the referred file actually is on the device. The most famous database is the *National Software Reference Library* (NSRL, NIST Information Technology Laboratory (2003–2013)) with its Reference Data Set (RDS).[3] However, due to their security requirements crypto hashes only allow yes-or-no decisions, whereas similarity hashing comparison outputs a match score between 0 and 100.

Identifying similarity has a long history and may start with the Jaccard index suggested by Jaccard (1901) that calculates the similarity of two finite sets A and B by $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$. A common application of the Jaccard index is plagiarism detection. Two strings are decomposed (e.g., by spaces or by n-grams) into tokens, which are the elements of the respective sets A and B. Then $J(A,B)$ is used to identify the similarity of the two input strings. However, the sets have to be kept in memory to compute $J(A,B)$, which may be very space consuming. For instance, assume that we split the long byte sequences into 4-g. In the worst case we then have to keep $2^{4 \cdot 8} = 2^{32}$ different 4-g in memory, i.e., 16 GiB.

Manber (1994) presented the sif tool to quantify similarities among text files. "Files are considered similar if they have a significant number of common pieces, even if they are very different otherwise." Manber uses a set of *anchors*, which are short character sequences. In order to test for similarity sif searches for anchors and considers the neighborhood, e.g., the next 50 characters. As comparing strings directly is time consuming Manber integrated Rabin fingerprinting (Rabin, 1981) to hash the substrings. Then it is possible to compare numeric values. The main problem is that training data is needed in order to identify reasonable anchors. As a consequence text files of different languages may not be comparable as they do not contain anchors.

In recent years similarity hashing has become more and more popular and thus new approaches were published. All approaches share two commonalities as they consist of

- a generation function that outputs a fingerprint/hash value/digest and
- a comparison function that measures the similarity of two fingerprints.

For the remainder of this paper we use the terms *similarity hashing* and *comparison function*, respectively.

Although FRASH will only be applied to ssdeep and sdhash, the following subsections briefly describe published similarity hashing algorithms and explain, where the algorithms succeed and where they fail in normal operation and when under attack. We decided to mention all algorithms for two reasons. First, this paper should give a rough overview of existing algorithms and how they proceed. Second, the papers describing the algorithms contain valuable test information and point to the authors' concerns. Readers familiar with the existing approaches may skip the remainder of Section 2.

2.1. Context triggered piecewise hashing

Similar to sif, Kornblum (2006) presented an algorithm known as context triggered piecewise hashing (abbreviated CTPH) that is based on the spam detection algorithm of Tridgell (2002–2009). The implementation is freely available and currently in version ssdeep 2.9[4].

The overall idea of ssdeep is quite simple. CTPH identifies trigger points to divide a given byte sequence into chunks. In order to generate a final fingerprint all chunks are hashed using FNV (Noll, 1994–2012) and concatenated.

[2] https://www.dasec.h-da.de/staff/breitinger-frank/#downloads; (last accessed 2013–04–11).

[3] http://www.nsrl.nist.gov; (last accessed 2013–04–11).

[4] http://ssdeep.sourceforge.net; (last accessed 2013–04–11).

To represent the fingerprint of a chunk CTPH only takes the least significant 6 bits of the FNV hash resulting in a Base64 character. To determine the distance of two fingerprints, they are treated as text strings and compared by using the weighted edit distance. The match score is scaled between 0 and 100.

A trigger point is identified as follows: Kornblum calculates a modulus called block size b, which correlates to the file size. Then a window of a fixed size 7 slides through the whole input, byte for byte, and generates a pseudo random number r at each step. If $r \equiv -1 \bmod b$, the byte sequence in the window is a trigger point and thus the end of the chunk.

Kornblum aims at having $S = 64$ chunks, but also needs to have the same value of b for similar sized files. He therefore defines the block size as a saltus function

$$b = b_{min} \cdot 2^{\left\lfloor log_2\left(\frac{N}{S \cdot b_{min}}\right)\right\rfloor}, \quad (1)$$

where $b_{min} = 3$ and N is the input length in bytes. Since the block size b is used for determining the chunks and depends on the length of the input, only ssdeep hash values with the same value of b can be compared. To be more flexible ssdeep outputs two hash values using b and $b/2$. As a consequence ssdeep cannot find similarity between inputs exceeding a factor of four by design (in the worst case only factor two).

Due to its uniqueness CTPH was improved in the upcoming years by several researchers (e.g., Chen and Wang (2008); Seo et al. (2009); Breitinger and Baier (2012b)) with respect to runtime efficiency. However, the paper at hand focuses on the original version of ssdeep as the improvements mainly address runtime efficiency. In addition no implementation of an improved version is available.

Baier and Breitinger (2011) did a detailed security analysis of CTPH where the authors demonstrated peculiarities and exploits to overcome this algorithm. The most obvious attack is to change one bit in each chunk, i.e., an expected amount of 64 manipulations per file. Due to the fact that two fingerprints need a minimum of 7 characters in common to be considered as similar, this attack can be optimized: manipulate one bit in each 7th chunk. The authors presented a second attack called *adding trigger points*, where they compute trigger points in advance and insert them at the beginning of the file, e.g., into the header of a JPG. Thus it is possible to fake every fingerprint.

2.2. sdhash

Four years later Roussev (2010) suggested a completely different algorithm named *similarity digest hashing*, which resulted in the tool sdhash.[5] Instead of dividing an input into chunks the algorithm extracts statistically improbable features (Roussev, 2009) by using the Shannon entropy; a feature is a byte sequence of 64 bytes. Hence files are similar if they share identical improbable features.

[5] http://roussev.net/sdhash/sdhash.html; (last accessed 2013–04–11).

sdhash computes the SHA-1 hash value of all extracted features. Each SHA-1 hash is split into five sub-hashes, however, only the least significant 11 bits of each sub-hash are used for sdhash, the remaining bits are dropped. In a last step the sub-hashes are inserted into a $2^{11} = 2048$ bit Bloom filter: if h_f denotes the 11-bit integer value of the sub-hash, the bit at position h_f is set to 1. Only a fixed number of features are inserted into a Bloom filter. If this number is exceeded, a new Bloom filter is created. Thus the final sdhash digest/fingerprint is a sequence of Bloom filters. In comparison to ssdeep we have a variable fingerprint length, which is approximately 2.5% of the input length.

A comparison of digests is a comparison of all Bloom filters of digest one against all Bloom filters of digest two with respect to the Hamming distance as metric. Due to the constant length of the Bloom filter an insertion of an appropriate number of bytes in the beginning shifts the features and reduces the similarity score.

The tool was improved over the last years with respect to accuracy and runtime efficiency, e.g., the newest version is parallelized. The current effectiveness of sdhash is demonstrated by Roussev and Quates (2012), where it solves the M57 triage problem in a very short time period.

Roussev (2011) provides a comparison of ssdeep and sdhash and shows that the latter "approach significantly outperforms in terms of recall and precision in all tested scenarios and demonstrates robust and scalable behavior".

Breitinger and Baier (2012c) did a detailed analysis of sdhash. In general only 80% of all input bytes influence the final hash value and thus it is possible to make uncovered changes. With respect to security the authors couldn't find an attack to fake fingerprints. The only weakness is called *Bloom filter shifts*: if a file allows inserting content in the beginning, it is possible to reduce the match score down to ≈ 25. However, this is still an acceptable score as Roussev (2011) shows that scores above 21 are reliable.

2.3. bbhash

The approach bbhash (Breitinger and Baier, 2012a) uses a fixed set of 16 random byte sequences called building blocks (bb) each with a length of $l = 128$ bytes (shall be 'short' compared to the file size). These building blocks are used to rebuild an input as accurately as possible. To find the optimal representation of a given file by the set of bb, bbHash moves through the input file byte-by-byte, reads out the current context of length l and computes the Hamming distance of all bb against the current input byte sequence. If the Hamming distance is smaller than a certain threshold, its index contributes to the file's fingerprint.

Due to the fact that processing a 10 MB needs approximately 2 min, we neglect this approach.

2.4. mrsh-v2

mrsh-v2 was proposed by Breitinger and Baier (2012d) and is based on MRS hash (Roussev et al., 2007) and CTPH. Similar to CTPH the algorithm divides an input into chunks and hashes each chunk. Instead of having a Base64 encoded fingerprint with a maximum length of 64 characters, mrsh-v2 makes use of a sequence

of Bloom filters. Like sdhash this leads to a variable length fingerprint, where the implementation aims at having 0.5% of the input length. As a specialty mrsh-v2 has two modes, one for fragment detection and one for file similarity.

The current implementation does not support an all-against-all comparison of files. Therefore we cannot consider this approach.

2.5. mvHash-B

As described by Breitinger et al. (2013), mvhash-B has three phases to create the fingerprint. First, majority voting is used to map every byte of the input data to either 0x00 or 0xFF. Majority voting in this case means counting the amount of 0s/1s in the *n*-neighborhood of the currently processed input byte. If the neighborhood is crowded by 1s, the majority vote yields an output 0xFF and vice versa. Next, run length encoding compresses these sequences of 0x00s or 0xFFs bytes. For instance, a run length encoded sequence can look like this 22|51|6|19|…|45. Finally, the run length encoded sequence is inserted into Bloom filters to represent the actual fingerprint. By design mvHash-B aims at having a fingerprint length of 0.5% of the input length.

A drawback of this implementation is the dependence on the file type: each file type requires its own configuration – no standard configuration works for all file types. In other words: although mvhash-B works on the byte level, it needs different configurations and is therefore not included into FRASH. However, mvhash-B is not a perceptual hashing algorithm.

3. Towards a test framework

Currently it is very hard to compare different algorithms. Most tools were briefly compared to other existing algorithms with respect to compression and runtime efficiency. However, Roussev (2011) pointed to additional features of similarity hashing. He enumerates the following challenges:

1. Document similarity detection: identify related documents, e.g., different versions of a Word document.
2. Embedded object detection: identify a given object inside a container, e.g., a JPG within a Word document.
3. Fragment detection: identify an original input based on a fragment, e.g., analyzing a device on the byte level.
4. Clustering files: group files that share similar content, e.g., a Word document and an e-mail.

Based on all the papers presented in Section 2 we designed and implemented the test framework FRASH, which contains tests for these different challenges. FRASH is split into two main categories *efficiency* and *sensitivity & robustness*, which are described in the following.

Each test description is divided into two parts. First, we give a reason why this test is important, and second, we explain the test itself. All details about their realization/implementation are given in Section 4.

3.1. Efficiency

The efficiency test is composed of three sub-tests called runtime efficiency, fingerprint comparison and compression to evaluate the basic properties of the algorithms.

3.1.1. Runtime efficiency

Runtime efficiency also known as *ease of computation* is one of the fundamental properties of hash functions. Due the large amount of data it is obvious that similarity hashing has to be fast.

Runtime efficiency measures the time, which the algorithm needs to process the input. Processing in this case means that we measure the time for reading the file from the device and generating the fingerprint. We include SHA-1 (Gallagher and Director, 1995) as a benchmark.

3.1.2. Fingerprint comparison

A fast fingerprint comparison is part of the runtime efficiency as an approach is only useful if it has a fast comparison function. The comparison time may vary due to different fingerprint length and comparison algorithms like Hamming distance or Levenshtein. One could rate this test as an extension of the aforementioned one.

Fingerprint comparison measures the time of the comparison proceeding. As said in Section 2 each approach has a comparison function to measure the similarity of fingerprints. Therefore this test does an all-against-all comparison of fingerprints in the corpus and measures the time. This time excludes the fingerprint generation mentioned from the previous paragraph.

3.1.3. Compression

Similar to runtime efficiency compression is the second fundamental property of hash functions. Traditional hash functions output a fixed length fingerprint, which is in contrast to similarity hashing, where we often have a variable length. As fingerprints are typically stored within a database a preferably short fingerprint is desirable.

Compression measures the ratio between input and output of an algorithm and returns a percentage value. To be more precise:

$$compression = \frac{output\ length}{input\ length} \cdot 100 \qquad (2)$$

3.2. Sensitivity & robustness

The sensitivity & robustness test is composed of four sub-tests called single-common-block correlation, fragment detection, alignment robustness and random-noise-resistance.

Sensitivity measures what amount of commonality is detectable by the algorithm. The tool that detects smaller levels of commonality is more sensitive. Higher sensitivity is generally better (up to a point). Fragment detection and common-block are representatives of the sensitivity part.

3.2.1. Single-common-block correlation

This test was proposed by Roussev (2011) and "simulates a situation where two files have a single common

object". Considering two files f_1 and f_2 that are completely different, but share a common object O, "what is the smallest O for which the similarity tool reliably correlates the two targets?" (Roussev, 2011).

First, two random files f_1 and f_2 of size $X \in$ {512 KB, 2048 KB,8192 KB} are created followed by the common block O of size $X/2$. Next, O overwrites f_1 and f_2 at different and randomly chosen offsets – the size of f_1 and f_2 remains X all over the time. Within the last step we perform a comparison of f_1 and f_2. If we obtain a match score >0, we reduce O by 16 KB and restart. The test stops when the match score $= 0$. Due to the fact that we choose the offset randomly we perform 5 runs for each file size and average the values.

The output shows the match scores in fives steps. For each match score it shows the minimum averaged size of O.

3.2.2. Fragment detection

Considering a file, what is the smallest piece/fragment, for which the similarity tool reliably correlates the fragment and the original file? Classical use cases are HDD block level analysis or network packets. For instance, analyzing an HDD on the sector level (e.g., 512 bytes), is it possible to find the original file; does this sector belong to this 800 MB movie?

Fragment detection identifies the minimum correlation between an input and a fragment. It sequentially cuts X% of the original input length and generates the match score whereby $X = 5$ by default. This means the maximum amount of cuts is $\lceil \frac{100}{X} \rceil - 1$. In case that the algorithm still identifies similarity we continue with a further reduction in 1% steps until only 1% of the input is left.

For instance, assuming the default value of $X = 5$ and an input length of 100,000 bytes. This results in 19 cuts of 5000 bytes each. As the next cut would result in *NULL*, the algorithm continues in cutting 1% pieces, i.e., 1000 bytes, until only 1% = 1000 bytes are left.

We decided to have two different modes:

1. *Random cutting* is the first mode. The framework randomly decides whether to start cutting at the beginning or the end of an input and then continues randomly.
2. *End side cutting* is the second mode and only cuts blocks at the end of an input.

The reason why we do not cut in the beginning is that this is similar to the alignment test, which is described in the next paragraph.

3.2.3. Alignment robustness

As written in Section 2 most approaches are vulnerable to inserting content at the beginning of the file, especially algorithms using Bloom filters. A typical real live scenario are logfiles. Hence, this test addresses challenge 1.

This test analyzes the impact of inserting byte sequences at the beginning of an input whereby we add fixed and percentage blocks. In general both tests consist of two parameters, the maximum size M and the size of a step s. To generate the final result the test inserts sequentially a block of size s at the beginning and stops after n steps when $n \cdot s \geq M$.

In both cases the result is an averaged overview, i.e., for each step i where $0 < i \leq n$ we average the match score.

1. *Fixed blocks* test defines a maximum block size of $M = 64$ KB and a step size of $s = 4$ KB. Additionally, it includes the results for inserting a block of 1 KB, 2 KB and 3 KB, respectively. We decided for a step size of 4 KB as this is the typical sector size. The additional smaller blocks show the behavior of small changes.
2. *Percentage blocks* test defines a maximum block size of $M = 100$% and a step size of $s = 10$% with respect to the original file length. Additionally, it includes the results for blocks of 200%,300%,400% and 500%. We decided for a step size of 10% in order to analyze the impact of large changes. Especially logfiles may grow very rapidly.

3.2.4. Random-noise-resistance

The analysis of ssdeep showed that a few changes all over the input are sufficient to obtain a non-match. The intention of the random-noise-resistance check is to have a randomly driven test trying to produce false negatives. This allows an estimation of how many bytes need to be changed all over the input to receive a non-match. Furthermore, this test aims at challenge 1.

The test analyzes the impact of random changes for an input. A random change is one of the typical edit operations deletion, insertion, and substitution, where each edit operation is chosen with a probability of 1/3. Additionally each byte in the input is equiprobable to be changed. To reduce the runtime of this test, we perform ten changes at a time instead of a single one.

Random-noise-resistance tries to answer the following question: What is the maximum number of changes if the match score s is equal or above X, i.e., $s \geq X$? In order to show a good overview we set $X = \{90,80,\ldots,0\}$. The output shows a fixed value, but also a percentage value. For instance, if $X = 90$ is given the algorithm responds with 20 changes for a 58,000 byte input. Thus the percentage value would be $(20/58{,}000) \cdot 100\% = 0.03448\%$. As the test framework is working on multiple inputs, it calculates an average value.

4. Implementation details of the test framework

The framework is implemented in Ruby 1.9.3 and currently supports sdhash and ssdeep. Due to the usage of the bash command find, a Unix environment is necessary to run the framework. Additionally, we included several Ruby packages called *gems*: actionpack, activesupport, i18n, activemodel, rack, erubis, colored and terminal-table. We recommend installing the *Ruby Version Manager (RVM)* as this allows easy handling of the Ruby environment including its package manager *gem* that allows the installation of gems like: gem install <GEMNAME>.

Further information about RVM[6] and gems[7] are available online.

[6] https://rvm.io/rvm/install; (last accessed 2013–04–11).
[7] http://docs.rubygems.org; (last accessed 2013–04–11).

4.1. Command line parameters and options

All following parameters are optional. The only mandatory part is PATH, which needs to be a file, a directory, or a combination separated by a space character. If the file or directory name contains spaces, they must be escaped with a backslash, e.g., file\ with\ spaces.txt.

FRASH has the following options:

```
$ frash [-v] [-r] [-t] PATH
```

`-h` prints usage instructions on the screen.
`-v` is the verbose mode and prints details for all files. This is only possible for the efficiency test. More details are given in Section 4.2.1.
`-t` allows to set the test scope, i.e., which tests will be performed. By default these are all tests. Parameters for -t are:

efficiency, single_common_block, fragment, alignment, random-noise.

`-r` reads the directory recursively, i.e., -r without any number reads all sub-directories. Optionally -r can be extended by a number, which sets the maximum number of recursive steps. If -r is not set, it is 1 by default.

The following is an example how to run the *efficiency* test for a specific directory:

```
$ frash -t efficiency /var/fileset/
```

Currently only fixed paths are possible, which means that the path needs to start from the root directory.

4.2. Proceeding

The minimum input is a file or directory otherwise the framework exits with an error message. Next, depending on the configuration, the framework runs the different tests in silent mode. After finishing all tests FRASH outputs a summary of the processed input consisting of: file count, total duration, average file size, total file size, and largest file followed by the test results. By default FRASH has the following test order: (1) efficiency test and (2) sensitivity & robustness test.

4.2.1. Efficiency test

This test is composed of 3 sub-tests called runtime efficiency, compression and fingerprint comparison, and are performed in this order. This is necessary as tests are based on the results of previous ones, e.g., 'compression' is based on 'runtime efficiency' as it needs the fingerprint-file. To derive the exact compression rate we neglect unnecessary information like the file name and focus on the fingerprint itself.

As said in Section 4.1 the efficiency test has two different modes:

Default mode simply passes PATH to the different hashing algorithms and pipes the output in a file.

Table 1
Statistics of the t5-corpus.

	jpg	gif	doc	xls	ppt	html	pdf	txt
t5-corpus	362	67	533	250	368	1093	1073	711

Verbose mode hashes all files individually. In other words, FRASH calls the find command to retrieve all *n* files. Next, each algorithm is called *n* times; once per file. Thus it is possible to receive the compression rate and runtime efficiency for all files separately. For instance, this might be helpful to analyze a couple of large files.

The runtime efficiency is based on the real time for each algorithm call. In default mode we start timing right before we hand the directory to the algorithm. To measure times we make use of Ruby's Benchmark[8] module, particularly its realtime method, which is similar to bash's time command, except that it returns only the elapsed real time of the executed code block. In order to have a benchmark for runtime we included SHA-1.

4.2.2. Sensitivity & robustness

This test is composed of four sub-tests called single-common-block correlation, fragment detection, alignment robustness and random-noise-resistance. Although by default they are performed in this order they are independent. All tests need the find command to identify the files within the directory. This is necessary as the tests process all files successively.

To compare algorithms we need to have equal inputs. Hence, we first modify the input and then hand it to *all* hashing algorithms. In other words, there are four working steps for all files: copy original input, manipulate the copy depending on the test, compare original against copy, save the result.

4.3. Integrating a new algorithm

In order to add a new algorithm it must fulfill the following requirements:

- Accept a directory and a file as input.
- Print fingerprint to standard output, e.g., Base64 encoded.
- The fingerprint must also contain the file name as it is used to determine the file producing the largest fingerprint.
- The algorithm needs to support an all-against-all comparison.

The new algorithm needs to be installed on the system and set in the environmental variable PATH.[9] Then it can be added to the framework performing the following steps:

[8] http://ruby-doc.org/stdlib-1.9.3/libdoc/benchmark/rdoc/Benchmark.html; (last accessed 2013–04–11).

[9] Here we talk about the environmental variable and not the console parameter. http://www.linfo.org/path_env_var.html; (last accessed 2013–04–11).

Table 2
Runtime efficiency and fingerprint comparison.

	Average	Total	Fingerprint comparison	$\frac{algorithm}{SHA-1}$
sha1sum	0.0013s	5.632s	–	1.00
ssdeep -s	0.0089s	39.789s	18.217s	7.06
sdhash	0.0167s	74.278s	346.730s	13.19
sdhash -p4	0.0066s	29.382s	346.902s	5.22

1. Create a wrapper. Copy the file new_algorithm_template.rb in the lib/hash_functions folder and decide on a name, e.g., my_alorithm.rb. The lines, which need to be implemented/changed, are marked with the comment #CHANGE ME:
   - PROGRAM_NAME contains the algorithm name, which will be called.
   - The variables @all_pair_comparison_command and @gen_compare_command contain the commands for an all-against-all and for a two-fingerprint comparison, respectively.
   - If the fingerprint file has a header set @digest_header to it as the framework needs to remove the header from the fingerprint file.
   - self.file_stats(digest) Processes the fingerprint and implements two extractor functions to divide the standard output into file name and hash value.
   - self.digest_length(hash) Computes the hash value length in bytes. For instance, if the hash value is represented by a Base64 string it should be multiplied by 0.75 as the hex presentation would be shorter.
2. The last step is the activation of the new algorithm. This is done by creating an object of the new algorithm as an instance variable in the initialize method of lib/testers/base_test.rb and adding it to the @hash_functions array. As we use inheritance it is also possible to add an algorithm to a specific class, e.g., fragment_detection_test.rb.

5. Experimental results & assessment

The following subsections describe the test results for ssdeep 2.9 and sdhash 3.2. In the latter case we only execute the default mode, i.e., 'block mode' or 'sampling' representing trade offs between storage/speed and precision were neglected. These modes allow to improve the test results for one domain but will worsen them in another. Hence, we focused on the default mode. However, in order to identify the best settings for a specific approach, the user needs to be familiar with the modes of an algorithm.

All tests expect random-noise-resistance are based on the t5-corpus[10] Roussev (2011, section 4.1) containing 4457 files of the file types given in Table 1 with a total size of 1.78 GB. This corresponds to an average of 418.91 KB per file. As random-noise-resistance is extremely time consuming, we randomly extracted a subset[11] containing 85 files which has a total file size of 14.4 MB and an average file size of 165.32 KB.

[10] http://roussev.net/t5/t5-corpus.zip; (last accessed 2013–04–11).
[11] https://www.dasec.h-da.de/staff/breitinger-frank/#downloads; (last accessed 2013–04–11).

Table 3
Compression test overview.

	Avg. hash length	Avg. ratio	Digest file size
sha1sum	20 B	0.00466%	311 KB
ssdeep -s	57 B	0.01329%	483 KB
sdhash	10.6 KB	2.52033%	61.2 MB

Most of the test results are very comprehensive wherefore we reduced them in the upcoming sections and only provide an overview.

5.1. Efficiency test results

The test environment for the efficiency was a private laptop having the following benchmark data:

CPU : 4x Intel(R) Core(TM) i7-2620M CPU @ 2.70 GHz (2 Cores, 4 Threads)
HDD : Mushkin Chronos SSD 120 GB (SATA3)
RAM : 2x4GB DDR3 SODIMM 1333 MHz.
Kernel : Linux 3.2.0-31-generic x86_64.

The -s parameter of ssdeep is the silent mode that suppresses all error messages. In case of sdhash we did two runs in which the second run had the -p4 parameter, which parallelized it, using 4 threads.

5.1.1. Runtime efficiency and fingerprint comparison

The results for both tests are given in Table 2. *Average* is the average time per file. *Total* and *fingerprint comparison* measure the time for hashing all files and do an all-against-all fingerprint comparison, respectively. The last column shows the relationship of all algorithms compared to SHA-1.

To sum it up, by default sdhash is slower than ssdeep but outperforms it when it is parallelized. Since the fingerprint comparison time of both sdhash runs are nearly equal, it looks like the comparison is not performed in parallel or at least without an improvement. Using SHA-1 as a benchmark all algorithms are significantly slower.

5.1.2. Compression

Table 3 shows the results for the compression test. The framework outputs the average hash value length, average compression ratio in percent, the maximum hash value including the corresponding file (not included in the table) and the size of all hashes.

To conclude, with respect to compression ssdeep outperforms sdhash as it produces much smaller fingerprints.

Table 4
An extract of the single-common-block correlation with a file size of 2048 KB.

	Score	≥40	≥30	≥25	≥20	≥5
ssdeep	Avg. block size (KB)	605	384	368	–	–
	Avg. block size (%)	29.53	18.75	17.97	–	–
	Matches	5	5	4	–	–
sdhash	Avg. block size (KB)	912	720	604	480	170
	Avg. block size (%)	44.53	35.16	29.49	23.44	8.28
	Matches	3	5	4	4	5

Table 5
An extract of the fragmentation detection test using random cutting.

	Fragment size	50%	30%	25%	20%	5%
ssdeep	Avg. score	65.86	50.90	47.62	44.98	26.00
	Matches (%)	94.64	38.64	20.75	8.86	0.04
	Std. deviation	10.09	10.29	11.34	13.08	1.00
sdhash	Avg. score	69.49	70.63	71.18	71.91	76.16
	Matches (%)	100	99.46	98.86	97.33	75.59
	Std. deviation	22.45	23.17	23.27	23.22	22.72

Table 7
An extract of the alignment robustness test with percentage blocks.

	Added block	10%	50%	100%	200%	400%
ssdeep	Avg. score	91.09	71.73	60.22	45.78	29.00
	Matches (%)	99.62	95.03	72.85	24.07	0.06
	Std. deviation	6.39	11.12	10.53	10.35	2.94
sdhash	Avg. score	67.65	69.58	68.46	68.41	67.52
	Matches (%)	100	100	100	100	100
	Std. deviation	21.30	21.58	21.66	21.58	21.98

5.2. Sensitivity & robustness test results

This section is divided into four paragraphs and focuses on the different tests single-common-block correlation, fragment detection, alignment robustness, and random-noise-resistance. Compared to efficiency we used a different workstation as runtime is unimportant and these tests are very time consuming.

5.2.1. Single-common-block correlation

Although FRASH outputs three tables showing the results for files of 512,2048 and 8196 KB this paper only includes a summary for 2048 KB as the results are similar for the others. Row 1 shows the match score. Row 2 and 3 are the average block size in KB and percentage, respectively. The last row is the amount of comparisons that yield the results, e.g., in column one we had 3 matches for sdhash having a score $\geq$40 out of 5 runs.

The main conclusion of Table 4 is that ssdeep outputs higher match scores for smaller pieces compared to sdhash. For instance, to output a match score of 30, sdhash needs a 720 KB piece whereas ssdeep is fine with a 384 KB piece. However, sdhash is able to detect smaller blocks like a 170 KB piece with a score of 5.

5.2.2. Fragment detection

Tables 5 and 6 show the results for *random cutting* and *end side cutting*, respectively. Actually each table comprises 24 columns containing the results for 95%,90%,85%,…,10%,5%,4%,3%,2%,1% fragments. We reduced this down to 5 for this paper in order to provide a suitable overview. The first row shows the average match score. 'Matches' is the amount of valid scores, i.e., how many percent of all files were matched. The last row is the standard deviation of the score.

As mentioned in Section 2.1 in many cases ssdeep can only detect file fragments between 50% and 25% which is also the conclusion considering both tables. The algorithm works with a high precision until pieces of 45%-fragments then the 'matches' reduces rapidly. Table 6 shows that ssdeep also uncovers 0.49% of 5%-fragments which is one of the mentioned special cases. On the other hand sdhash outputs a very high rate also for 5%-fragments. In case of 1% fragments sdhash has 43.53% matches.

Comparing both cutting modes the results for ssdeep are nearly the same. However, sdhash shows a different behavior which is due to the fingerprint representation using Bloom filters. End side cutting only influences the last Bloom filters, thus, the beginning is equal which results in a high score. On the other hand random cutting also cuts the beginning which shifts features to different Bloom filters and reduces the score.

5.2.3. Alignment robustness

The results are Tables 7 and 8 showing the impact of *percentage blocks* and *fixed blocks*, respectively, in which the rows are equal to the ones from fragment detection.

Similar to fragment detection, ssdeep can hardly find a similarity if the modification is too large. For instance, ssdeep can only detect similarity for 24% if we add 200%. But therefore it is very robust against small changes. This is in contrast to sdhash where we have 100% matches but with lower scores.

In the authors' opinion sdhash shows a better alignment robustness as it identified all.

5.2.4. Random-noise-resistance

Random-noise-resistance is presented by Table 9. Row 1 and 2 are the average changes in value and percentage, respectively. Matches is the amount of files yielding the result.

Recall, this test presents the average amount of manipulations so that the match score is above or equal 90,80,…,0. Again we selected the most important columns which are 6 out of 10.

Doing random changes all over the file is a weakness of ssdeep. For instance, there are only 29 matches for 85 manipulations whereas sdhash identifies 78 matches with 729 manipulations (column $\geq$50). Having a look at the last column, sdhash still outputs a similarity score $\geq$10 when 1.10% of the input file is manipulated.

Table 6
An extract of the fragmentation detection test using end side cutting.

	Fragment size	50%	30%	25%	20%	5%
ssdeep	Avg. score	71.73	58.26	56.76	55.35	55.68
	Matches (%)	93.07	44.56	26.72	14.18	0.49
	Std. deviation	11.84	14.49	15.07	15.37	19.92
sdhash	Avg. score	99.51	99.21	99.09	98.93	98.26
	Matches (%)	100	99.84	99.55	97.98	76.73
	Std. deviation	1.36	1.79	2.04	2.25	2.96

Table 8
An extract of the alignment robustness test with fixed blocks.

	Added block	1 KB	4 KB	16 KB	32 KB	64 KB
ssdeep	Avg. score	96.56	91.25	82.66	79.33	76.47
	Matches (%)	100	99.69	87.91	74.29	59.28
	Std. deviation	3.79	10.51	16.27	17.84	18.40
sdhash	Avg. score	84.11	51.47	64.37	52.68	78.12
	Matches (%)	100	100	100	100	100
	Std. deviation	10.57	21.04	17.01	21.05	15.90

Table 9
An extract of the random-noise-resistance test.

	Score	≥80	≥60	≥50	≥30	≥20	≥10
ssdeep	Avg. changes	14.65	43.89	85.17	160.00	–	–
	Avg. changes (%)	0.009%	0.026%	0.050%	0.094%	–	–
	Matches	71	54	29	1	–	–
sdhash	Avg. changes	211.67	514.62	729.36	1116.24	1483.54	1860.83
	Avg. changes (%)	0.1216%	0.304%	0.431%	0.660%	0.877%	1.100%
	Matches	78	80	78	85	82	84

6. Conclusion

Manually comparing similarity hashing algorithms is a complex task and require a lot of time. With this paper we took the challenge to create a framework to test similarity hashing algorithms. The result is a tool called FRASH that is open source.

FRASH currently includes two test classes called *efficiency* and *sensitivity & robustness*. The former comprises runtime efficiency, fingerprint comparison and compression. Our tests showed that ssdeep has the better compression and fingerprint comparison whereas sdhash supports parallelism and outperforms ssdeep in runtime efficiency.

Sensitivity & robustnessis composed of four sub-tests named single-common-block correlation, fragment detection, alignment robustness, and random-noise-resistance. As shown in the assessment section, sdhash dominates all of these sub-tests.

In general there are three next steps. Further similarity hashing algorithms need be integrated to identify their strengths and weaknesses. The relevance and correctness of existing tests have to be discussed by the community in order to improve them. Additionally, we are exploring other tests to add to the framework that would produce data that is of interest to, and have an impact on forensic practitioners.

Acknowledgments

The authors would like to thank Simson Garfinkel from the Naval Postgraduate School in Monterey, California and Vassil Roussev from the University of New Orleans for valuable discussions. Additionally, we are thanking the others of the 'approximate matching working group'; John Delaroderie, Barbara Guttman, John Kelsey, Jesse Kornblum, Mary Laamanen, Michael McCarrin, Clay Shields, Douglas White, John Tebbutt, and Joel Young.

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

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

Bertoni G, Daemen J, Peeters M, Assche GV. Keccak specifications 2009.

Breitinger F, Åstebøl KP, Baier H, Busch C. mvhash-b – a new approach for similarity preserving hashing. IT Security Incident Management & IT Forensics (IMF); 2013.

Breitinger F, Baier H. A fuzzy hashing approach based on random sequences and hamming distance. In: 7th annual conference on digital forensics, security and law. ADFSL; 2012a. p. 89–100.

Breitinger F, Baier H. Performance issues about context-triggered piecewise hashing. In: Gladyshev P, Rogers M, editors. Digital forensics and cyber crime. Berlin Heidelberg: Springer; 2012b. p. 141–55. volume 88 of lecture notes of the institute for computer sciences, social informatics and telecommunications engineering.

Breitinger F, Baier H. Properties of a similarity preserving hash function and their realization in sdhash. In: 2012 information security for South Africa (ISSA 2012) 2012.

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.

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. p. 635–8.

Dewald A, Freiling F. Is computer forensics a forensic science? In: M.-P.-I. für ausländisches und internationales Strafrecht, Freiburg U, editors. Proceedings of current issues in IT security 2012. pp. 0–0.

Gallagher P, Director A. Secure hash standard (SHS). Technical report national institute of standards and technologies. Federal Information Processing Standards Publication; 1995. p. 180–1.

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

Jaccard P. Distribution de la flore alpine dans le bassin des drouces et dans quelques regions voisines. Bulletin de la Société Vaudoise des Sciences Naturelles 1901:241–72.

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

Manber U. Finding similar files in a large file system. In: USENIX winter 1994 technical conference 1994. p. 1–10.

Nechvatal J, Bassham EBL, Dworkin M, Foti J, Roback E. Report on the development of the advanced encryption standard (AES) 2000. Technical Report National Institute of Standards and Technology.

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. Cambridge, Massachusetts: Center for Research in Computing Technology, Harvard University; 1981.

Roussev V. Building a better similarity trap with statistically improbable features. In: System sciences, 2009. HICSS '09. 42nd Hawaii international conference on 2009. p. 1–10.

Roussev V. Data fingerprinting with similarity digests. In: Chow K-P, Shenoi S, editors. Advances in digital forensics VI. Berlin Heidelberg: Springer; 2010. p. 207–26. volume 337 of IFIP advances in information and communication technology.

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

Roussev III V, R. GG, Marziale L. Multi-resolution similarity hashing. Digital Investigation 2007;4:105–13.

Roussev V, Quates C. Content triage with similarity digests: the M57 case study. Digital Investigation 2012;9:60–8.

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 on 2009. p. 1–6.

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