Forensic Science International: Digital Investigation 42 (2022) 301407

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2022 USA - Proceedings of the Twenty-Second Annual DFRWS USA

FRASHER – A framework for automated evaluation of similarity hashing

Thomas Göbel [a, *], Frieder Uhlig [b], Harald Baier [a], Frank Breitinger [c]

[a] Research Institute CODE, Universität der Bundeswehr München, Munich, Germany
[b] Technical University Darmstadt, Darmstadt, Germany
[c] School of Criminal Justice, University of Lausanne, 1015, Lausanne, Switzerland

ARTICLE INFO

Article history:

Keywords:
Test framework
Test cases
Evaluation framework
Approximate matching
Similarity hashing
Fuzzy hashing
Similarity digest algorithm

ABSTRACT

A challenge for digital forensic investigations is dealing with large amounts of data that need to be processed. Approximate matching (AM), a.k.a. similarity hashing or fuzzy hashing, plays a pivotal role in solving this challenge. Many algorithms have been proposed over the years such as `ssdeep`, `sdhash`, `MRSH-v2`, or `TLSH`, which can be used for similarity assessment, clustering of different artifacts, or finding fragments and embedded objects. To assess the differences between these implementations (e.g., in terms of runtime efficiency, fragment detection, or resistance against obfuscation attacks), a testing framework is indispensable and the core of this article. The proposed framework is called `FRASHER` (referring to a predecessor `FRASH` from 2013) and provides an up-to-date view on the problem of evaluating AM algorithms with respect to both the conceptual and the practical aspects. Consequently, we present and discuss relevant test case scenarios as well as release and demonstrate our framework allowing a comprehensive evaluation of AM algorithms. Compared to its predecessor, we adapt it to a modern environment providing better modularity and usability as well as more thorough testing cases.

1. Introduction

A survey conducted by Singh (2021) states that 65% of practitioners utilize approximate matching (AM) during their investigations to filter out relevant artifacts. Hence, AM has arrived in the daily routine of the digital forensic process. According to the authors, AM is mainly used to identify related (i.e., similar) documents. However, besides similarity detection, AM can be used for embedded object detection (e.g., a malware embedded in a file), fragment detection (e.g., detecting file fragments in network traffic), and clustering of files (e.g., revealing artifacts with similar content) (Breitinger et al., 2014).

1.1. Problem

Since the presentation of `ssdeep` and hence the introduction of AM, a variety of different algorithms have been published, such as `sdhash`, `MRSH`, and `TLSH`. However, there is currently no consensus in the community, which AM algorithm suits best for a particular use case (Singh, 2021). It lacks a comprehensive testing framework that provides reliable metrics, which algorithm fits best for a given use case, rather than basing the outcome of an investigation on default tool settings or personal experiences. Such a framework also allows to automatically compare AM algorithms to uncover their strengths and weaknesses, respectively; so far comparisons have mostly been done manually.

1.2. Contribution

According to Breitinger et al. (2014), AM may operate on the byte, semantic, or syntactic level. This work aims at solving the raised issue with respect to bytewise AM algorithms. We provide an up-to-date view on the problem of evaluating AM algorithms with respect to both the conceptual and the practical aspects. With respect to the practical aspects, we present our open-source test

* Corresponding author.
E-mail addresses: thomas.goebel@unibw.de (T. Göbel), frieder.uhlig@stud.tu-darmstadt.de (F. Uhlig), harald.baier@unibw.de (H. Baier), frank.breitinger@unil.ch (F. Breitinger).
URL: https://www.tu-darmstadt.de/, https://www.unibw.de/digfor, https://www.FBreitinger.de

https://doi.org/10.1016/j.fsidi.2022.301407

framework called FRASHER allowing comprehensive evaluations of AM algorithms. FRASHER follows a modern design and is platform independent as it is implemented in Python and follows a modular architecture with well-defined interfaces. The modular concept allows integrating new algorithms and new test cases. Hence, automation of the evaluation process and usability as additional requirements are satisfied, too. With respect to the conceptual aspects, we present and discuss meaningful test cases with a special focus on the use case class containment as a benefit of bytewise AM (e.g., fragment detection, embedded object detection). As a demonstration, we assess several algorithms and summarise the results.

1.3. Comparison with existing work

Automating the testing for AM algorithms is not new and has been discussed about a decade ago by Breitinger et al. (2013), who presented FRASH as the main inspiration of our work. Consequently, several tests and strategies of FRASH have been integrated into our framework FRASHER. In particular, three tests with respect to efficiency (generation efficiency, comparison efficiency and compression efficiency) as well as three sensitivity and robustness tests (single common block, fragment detection and alignment robustness) are inherited from FRASH. Novel innovations are: a new implementation in the well-known scripting language Python and a more modular design (we argue that this is an increase of usability), more supported and recent AM algorithms (at the time of writing this paper: 7 instead of 2), broader test cases which are based on Baier and Breitinger (2011); Breitinger et al. (2012); Breitinger and Baier (2012a); Oliver et al. (2014); Chang et al. (2015), such as reduction of similarity or emulation of similarity as well as providing a visualisation component (test results of FRASH were exclusively text based where FRASHER supports plotted graphs).

1.4. Article structure

In Section 2 we discuss the current research and typical use cases of similarity hashing algorithms to motivate our framework and our test classes. In Section 3 we introduce the concept and design of FRASHER followed by Section 4, where we discuss our conceptual evaluation methodology for the AM algorithms. Next, Section 5 presents the implementation of FRASHER, while Section 6 highlights the results of our sample evaluation. Lastly, Section 7 outlines some limitations of our current framework and opportunities for future work.

2. Related work

Kornblum (2006) introduced a context triggered piecewise hashing (CTPH) algorithm known as ssdeep and evaluated his algorithm with respect to runtime performance (compared to some cryptographic hash functions like MD5 or SHA256) and detection performance (with respect to the use cases altered document matching and partial file matching). A more detailed analysis of ssdeep with respect to the runtime efficiency was done by Breitinger and Baier (2011) and resistance against obfuscation attacks by Baier and Breitinger (2011). However, the respective analysis was done manually and for one algorithm.

Roussev et al. (2007) presented a variation of ssdeep called multi-resolution similarity hashing, also known as mrshash. Roussev continued his work and later presented the similarity digest algorithm sdhash (Roussev, 2010) which was evaluated with respect to fragment detection. In a subsequent study he manually compared ssdeep and sdhash, where his three test cases were embedded object detection, single-common-block file correlation, and multiple-common-blocks file correlation (Roussev, 2011). Breitinger et al. (2012) performed an in-depth manual assessment of the implementation of sdhash and evaluated its obfuscation resistance.

Breitinger and Baier (2012b) presented an updated version of mrshash known as MRSH-v2 and introduced eligible properties of similarity preserving hashing. The authors first enumerated general properties for similarity preserving hashing (compression, ease of computation, similarity score) and then introduced two security properties (coverage and obfuscation resistance). As use cases the authors proposed file identification and fragment detection and evaluated MRSH-v2 against these properties and use cases, again without any automation. The same applies to the mrsh-cf algorithm presented by Gupta and Breitinger (2015), which replaces the Bloom filter with a cuckoo filter.

As categorization of AM supports the evaluation methodology, some work was done in this area, e.g., Martínez et al. (2014); Lee and Atkison (2017); Ribeiro et al. (2017). A categorization based on behavioral characteristics was recently proposed by Martín-Pérez et al. (2021) which consists of three categories:

- **Feature Sequence Hashing:** This category encompasses algorithms that split the input into features measuring the similarity of the input by feature sequences. Well-known representatives of this category are ssdeep, sdhash and MRSH-v2.
- **Byte Sequence Existence:** This category comprises the algorithms that identify the existence (or similarity) of byte sequences (called blocks) in the input. The similarity score is calculated by comparing the number of common blocks between similarity digests. Well known representatives are the algorithms SimHash, mvHash-B and LZJD.
- **Locality-Sensitive Hashing:** This category is made up of algorithms that map objects into buckets, grouping similar objects in the same bucket with high probability. Exemplary algorithms are Nilsimsa, TLSH and FbHash.

To overcome the time-consuming manual testing, Breitinger et al. (2013) proposed FRASH, a framework performing a structured assessment of bytewise AM algorithms. The main aspects of FRASH are depicted in Fig. 2. Inspired by previous evaluations, FRASH made use of two test classes: (1) efficiency tests that are composed of three sub-tests runtime efficiency, fingerprint comparison and compression; and (2) sensitivity & robustness tests which contains four sub-tests, single-common-block correlation, fragment detection, alignment robustness and random-noise-resistance. FRASH was a CLI-Ruby implementation and wrote its output to a text file without any visualisation. Well-defined interfaces, adaptable modules or appropriate documentation to integrate further algorithms or test cases were not supported by FRASH.

As the creation of AM algorithms and their assessing moved on, the National Institute of Standards and Technology (NIST) defined AM as “a promising technology designed to identify similarities between two digital artifacts […] to find objects that resemble each other or to find objects that are contained in another object” (Breitinger et al., 2014). The NIST Special Publication explicitly refers to FRASH and includes its test cases. However, no further concept extension nor an implementation were given.

To sum up the related work considerations: so far the evaluation of an AM algorithm was often done manually (i.e., not based on an automated framework) and only in relation to a competing algorithm. Hence, the outcome of an evaluation was not a comprehensive recommendation for action that catalogs the strengths and weaknesses of the most important contemporary algorithms. This

lack of clarity is the main motivation for our proposed design and its implementation in the scope of the framework FRASHER.

3. Concept and design

In this section we explain our concept and design of an automated evaluation framework for AM.

3.1. Requirements

Based on the canonical software engineering process, we first gather requirements on both the functional and non-functional level as the basis of our concept. The non-functional requirements are summarised in Table 1.

We consider *modularity* as the most important non-functional requirement, as it enables an exchange and expansion of future modules. Additionally, modularity supports the definition of *well-defined interfaces* as a further non-functional requirement to link new algorithms or additional test cases to the framework. Our specification aspect *usability* means that even non-experts in the field of AM shall be able to use it, e.g., by a detailed documentation of framework usage, its source code, and the provision of a configuration file to perform the actual tests. A further usability aspect is the visualisation of the output in terms of a graphical representation.

The functional requirements are listed in Table 2. Our first expectation, *resemblance*, reflects the respective use case class from Breitinger et al. (2014), the same holds for the second functional requirement *containment*. Third, *efficiency* is measured with respect to the run time performance of the algorithm and its memory consumption. Furthermore the functional requirement *realistic test cases* demands that the framework tests reflect actual use cases in a digital forensic examination and *resistance against obfuscation* assesses the algorithm's ability not to be bypassed by an attacker.

3.2. Design

An overview of the framework is given in Fig. 1. The architecture follows the modularity principle, where each module has a dedicated task and may communicate with other modules through interfaces. The `playbook module` is a core module to enable the requirement usability. The playbook holds the configuration for every test-run, i.e., it specifies, which algorithms, test cases and files are to be used and manipulated for the tests. The playbook should be implemented by a common text-based configuration format, e.g., the JavaScript Object Notation (JSON) format. The framework command line interface tells the framework, which playbook to run. The actual tests are performed by the `test_case module`, which receives as input the test configuration of the playbook and the test files from the test file repository of the framework. The test cases may manipulate the files in different ways and log the similarity scores returned by the respective algorithm. The `test_case module` appends its results to the playbook. Hence, tests can be repeated with the same or slightly modified configurations based on previous test results by adapting the playbook configuration. The `eval module` is responsible to evaluate the test results from the playbook and to visualise them for a better understanding.

Table 1
Non-functional requirements of an automated evaluation framework.

Requirement	Explanation
Modularity	The framework is split into different modules, e.g., an approximate matching algorithm to test, test cases, configuration.
Well-defined interfaces	Approximate matching algorithms or test cases are linked to the framework via predetermined functions.
Open-source	The framework and its source code are publicly available, e.g., via a GitHub repository.
Usability	The framework may easily be used by an ordinary computer scientist, e.g., due to an easy configuration of the evaluation and a visualisation of its output.
Adaptability	The framework may be adapted to further hardware or software platforms.

4. Evaluation test cases

In Table 2 we list our proposed functional requirements. Based on previous work and these requirements we reflect and propose evaluation test cases. In particular, various previously proposed test cases are collected, updated, and extended. We also consider challenges that play a major role in the daily routine of digital forensics to satisfy the requirement of a realistic test case.

4.1. Efficiency

Efficiency was already proposed in FRASH as an evaluation test case. We consider it as a functional requirement as stated in Table 2, too. As proposed in Breitinger et al. (2013); Breitinger and Roussev (2014), it is evaluated by three separate tests:

The *generation efficiency* measures the runtime efficiency (i.e., the practical execution time) of the similarity digest computation in relation to the input size. Generation efficiency is an important key figure as it reflects the ease with which the respective algorithm is able to process a certain amount of data to produce the similarity digest.

Comparison efficiency expresses the runtime efficiency to compare digests. As in the original FRASH framework this test entails an all-vs-all test. However, it does not include the aforementioned step of fingerprint generation.

Finally, *compression efficiency* evaluates the compression rate of the algorithm. It measures the size of the similarity digest in relation to the input data length and returns a percentage value, hence it is calculated by:

$$\text{compression} = \frac{\text{output length}}{\text{input length}} \cdot 100$$

4.2. Sensitivity & robustness

The test cases of this class resemble the ones that were introduced in the original FRASH. They are all performed in a one-vs-one fashion, that is the algorithm is tested with a pair comprising of a variant and its underlying original file. In Section 4.4 we evaluate the one-vs-all test cases, that is a manipulated file amongst many similar ones. In what follows only a rough description of the respective test is given, we explain the test details in Section 6.2.

4.2.1. Single common block

Tests the capabilities of an AM algorithm to detect common fragments in otherwise completely different files. The common fragment is decreased in size making it harder for the algorithms to register the commonality until it is no longer present.

4.2.2. Fragment detection

Identifies the minimum correlation between an original file and a fragment of it. This covers the containment requirement; the use case is explicitly stated by Breitinger et al. (2014). An original file is

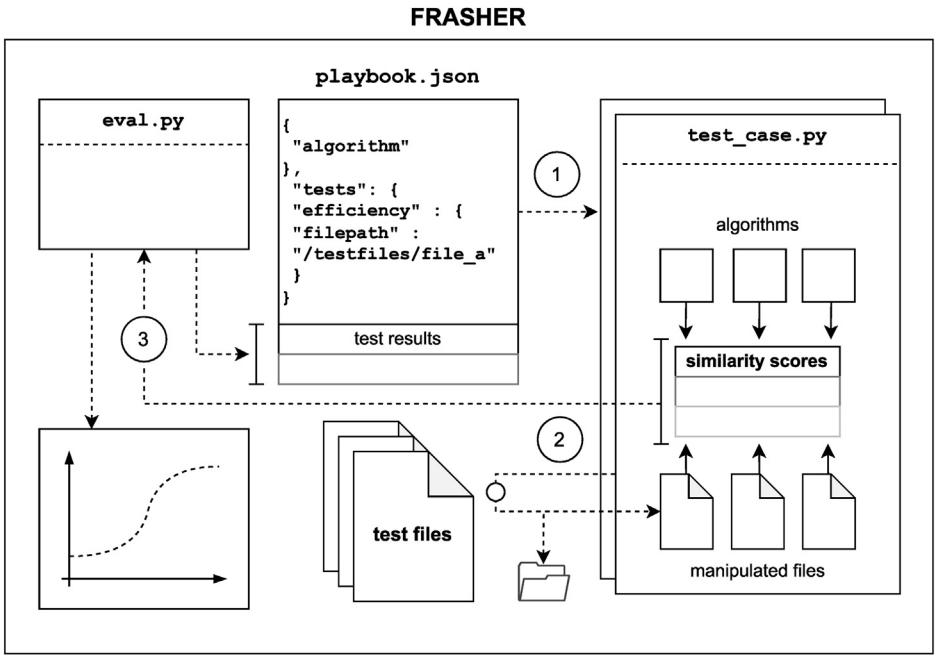

Fig. 1. Design of an automated evaluation framework with *modularity* as its key design principle (such as FRASHER). Each test-run starts with the playbook configuration, then the test cases create their own manipulated files and execute different algorithms on them. Finally, the results are evaluated, visualised and appended to the playbook. The newly created test files are saved and can be used in future test-runs.

Table 2
Functional requirements of an automated evaluation framework.

Requirement	Explanation
Resemblance assessment	Test cases to evaluate object similarity and cross correlation.
Containment assessment	Test cases to evaluate fragment and embedded object detection.
Efficiency tests	Efficiency with respect to runtime and compression efficiency.
Realistic test cases	Test cases must reflect practical use cases.
Resistance against obfuscation	Test cases to measure the difficulty in presence of an active adversary to circumvent a blacklist.

gradually trimmed by a fixed percentage value creating a fragment. This fragment is decreased in size until a correlation with the original file is no longer possible. Again we measure the result of the fragment detection with respect to the limit size of the fragment. A more advanced fragment detection test may contain two or more fragments, though. The fragment detection test runs in two modes. When using *random cutting* the first cut-off from the original file is done randomly at the beginning or the end of a file and then continued at random positions. When creating the fragments through *end side cutting*, the file is only trimmed at the end. In the current work we only present the results of *random cutting*. For random generated data the distribution of features in the common fragment tend to be evenly distributed which is why it is sufficient to cut the fragment randomly. In case of real file types fragment features may be unevenly distributed, which is why cutting from one end might lead to sudden jumps in the similarity score whenever an important feature is deleted from the fragment.

4.2.3. Alignment robustness

Evaluates the resilience of the AM algorithm against changes in the length of files. + This is a similar challenge to log files, which can quickly grow in size. Test files are enlarged by adding random bytes either to their beginning or end in a continuous manner. This way it can be evaluated how well the algorithms can identify files that are no longer the same size but have become larger.

4.3. Adversarial resilience

Adversarial resilience addresses the obfuscation requirement of Table 2. It describes the resilience of an algorithm against deliberate attacks performed by third parties. These kinds of attacks are for instance important to consider as part of data loss prevention filters. An adversary who is able to deliberately trigger the filter through false positives might be able to mislead an administrator into lowering the sensitivity of the filter. Alternatively, an adversary might be able to reduce the similarity score of a blacklisted artifact to exfiltrate sensitive data by circumventing the filter. These tests are inspired by rigorous attempts in the past to circumvent the robustness of various algorithms (Baier and Breitinger, 2011; Breitinger et al., 2012; Breitinger and Baier, 2012a; Oliver et al., 2014; Chang et al., 2015). Known attacks on AM algorithms have also been collected by Martín-Pérez et al. (2021).

According to Martín-Pérez et al. (2021), attacks can be divided into *attacks against the similarity score* and *attacks against impeding the last phases of a similarity digest algorithm*. While the former can be done either by *reduction of similarity* or *emulation of similarity*, the latter can be done either by *impeding the digest generation phase* or *impeding the digest comparison phase*. However, all of these attacks require an intelligent attacker who knows exactly how the algorithms work. We consider sample obfuscation attacks presented by Martín-Pérez et al. (2021) in what follows.

The goal of *Impeding the Digest Generation Phase* is to craft an input that cannot be processed, this might be because the input does not meet the level of diversity among the features that is needed for generating a digest. Or through avoiding byte sequences that match with the values needed by trigger functions.

Impeding the Digest Comparison Phase addresses an attacker who intentionally creates inputs that do not meet the minimum amount

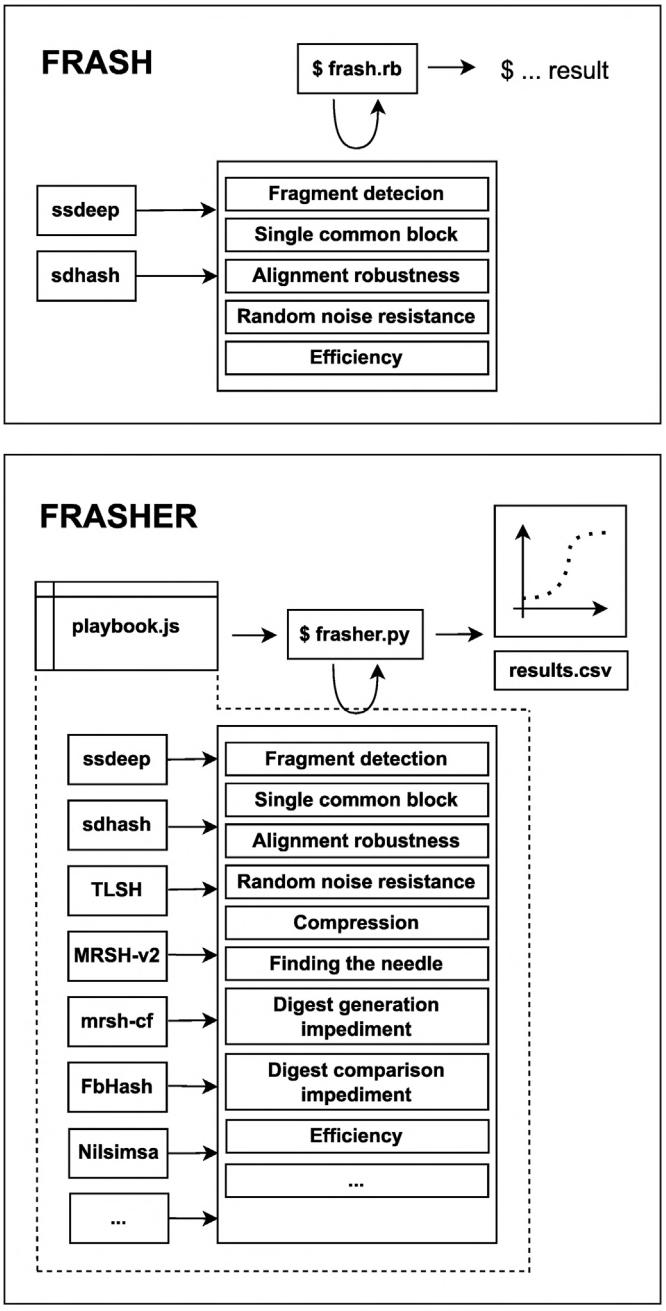

Fig. 2. Comparison of our framework FRASHER to its predecessor FRASH.

of comparison requirements. Hence the comparison phase fails to identify similar files. This can be tested by creating a file that is made up of ever smaller parts of the original input file. At one point the manipulated file lacks the significant features to be hashed (after the deduplication phase).

4.4. Finding the needle

Our *finding the needle* test case is inspired by the digital forensics need to detect a case-relevant artifact (e.g., a picture of child abuse or a malware) on an analysed device, hence it addresses the use case of blacklisting. We therefore cast a different view on the *sensitivity & robustness* test case.

If a test makes use of a haystack, that is a corpus of some thousands of files, a common test corpus in the digital forensics community should be used, e.g., the *t5-corpus* (Roussev, 2011; Breitinger and Roussev, 2014), the *NapierOne* data set (Davies et al., 2022) or the *Govdocs1* corpus (Garfinkel et al., 2009), but any sufficiently large collection of files may be used.

4.4.1. Resemblance – object similarity detection

As stated by Breitinger et al. (2014), this use case deals with identifying related artifacts. The object similarity test makes use of a haystack. Our suggestion is to take ten examples of a file type which are all similar in size. Every file is manipulated in ten ways each resulting in a unique needle that poses a challenge, e.g., on the alignment robustness, random noise resistance. The algorithm has to find the needle based on the original unmanipulated file amongst the others in the haystack. If the matching score between the needle and the original unmanipulated file in the haystack has the highest value amongst all others, it is considered a TP. If other files have the highest similarity score, this is seen as a FP. A TP amongst reasonable few FP constitutes a success of the experiment and is recorded.

4.4.2. Resemblance–cross correlation

The cross correlation test takes two completely different files of approximately the same size as containers and inserts a significantly smaller object (i.e., the needle) into each file at random positions. The size of the inserted needle is reduced until a similarity digest algorithm can no longer correlate the two files through a similarity score >0. We call the final size of the needle, which is still correlated by the AM algorithm, the *limit size* of the cross correlation test. Note, Roussev (2011); Breitinger and Roussev (2014) utilize the term *single-common-block correlation*.

5. FRASHER

Our framework FRASHER extends its predecessor FRASH with respect to its conceptual and functional design. First, we give an overview of FRASHER in Section 5.1 and reflect on our framework with respect to the requirements of an automated AM evaluation framework. Subsequently, we explain the usage of FRASHER.

5.1. Overview of FRASHER

FRASHER follows the non-functional and functional requirements listed in Tables 1 and 2, respectively. In what follows we describe to which extent FRASHER fulfills the requirements outlined in Section 3. A feature overview and a comparison to FRASH is depicted in Fig. 2. The figure shows that FRASHER is extensible in terms of its supported algorithms, test cases and evaluation procedures.

The most important non-functional requirement is *modularity*. In contrast to the monolithic predecessor, FRASHER provides the modules proposed in Fig. 1. Fig. 2 lists the currently available AM schemes and test cases. Currently the algorithms `ssdeep`, `sdhash`, `TLSH`, `MRSH-v2`, `mrsh-cf`, `Nilsimsa` and `FbHash` are supported. Due to its modular architecture, further algorithms, such as `ssdeeper`, `FbHash-E` or `SiSe`, can easily be integrated.

FRASHER provides interfaces, which are reflected by its folder structure as given in Fig. 3. The lib directory provides the functionality of our framework, while the testdata directory stores the test corpus of files, which serve as input for the different test cases. The lib/fuzzy_hashes folder contains the APIs to all AM algorithms. In addition, the executables (in Python or Windows Portable Executable format) of integrated algorithms are stored here. To evaluate an additional algorithm, the respective *.py or *.exe file

has to be stored in this directory. The lib/helpers folder contains the auxiliary methods and code needed for the bytewise manipulation of the test files which are used as input. Lastly, the lib/frasher_testcases t folder contains a variety of test cases that represent the actual challenges and use cases for the algorithms. Further test cases can be integrated here. The evaluation directory provides the code for the subsequent result analysis and visualisation, while the config directory stores all the playbooks.

The source code of FRASHER and its documentation are publicly available (under the Apache 2.0 licence) to the digital forensics community via GitHub: https://github.com/warlmare/FRASHER.

FRASHER is easily useable due to its visualisation component and its central configuration via the playbook file based on the JSON format as shown in Listing 1. All test-runs are specified in the tests section in the respective playbook file. These are then executed consecutively for each algorithm specified in the algorithm section. The results of all test rounds specified with rounds are averaged, printed to the console and finally appended in JSON format to the playbook. In addition, the measured values are stored in CSV files for later reuse in the results-path specified by the user. For instance, in our sample playbook in Listing 1, we evaluate `ssdeep`, `mrsh-cf`, and `TLSH`. Additionally, test-runs with respect to efficiency, cross correlation (i.e., single common block), and alignment are performed, where each test is executed ten times and the outcomes are additionally stored (next to the playbook) as CSV files in the results folder.

```
{
  "algorithm": {
    "ssdeep",
    "mrsh-cf",
    "tlsh"
  },
  "tests": {
    "efficiency" : {
      "testfiles" : {
      "randomgenerate" : "1KB - 10KB",
      "stepsize" : "500 byte"
      }
    },
    "single_common_block": {
      "testfiles" : "testfiles/docx",
      "blockfiles" : "testfiles/jpg"
    },
    "alignment": {
    "testfiles" : "testfiles/docx"
    }
  },
  "rounds": 10,
  "results-path": "results/x"
}
```

Listing 1. Sample file *example playbook.json* shows playbook syntax.

FRASHER follows the adaptability requirement. An important aspect with respect to this requirement is the programming language. While FRASH made use of Ruby, FRASHER is implemented in Python 3.9. The framework itself, as well as the four algorithms `ssdeep`, `FbHash`, `Nilsimsa` and `TLSH`, can be run in Windows, Linux and Unix environments.[1] Any other currently used algorithm within FRASHER can only be used on the platform, on which the precompiled executable of the respective algorithm is provided.

[1] Since there is an original Python library available for these two algorithms.

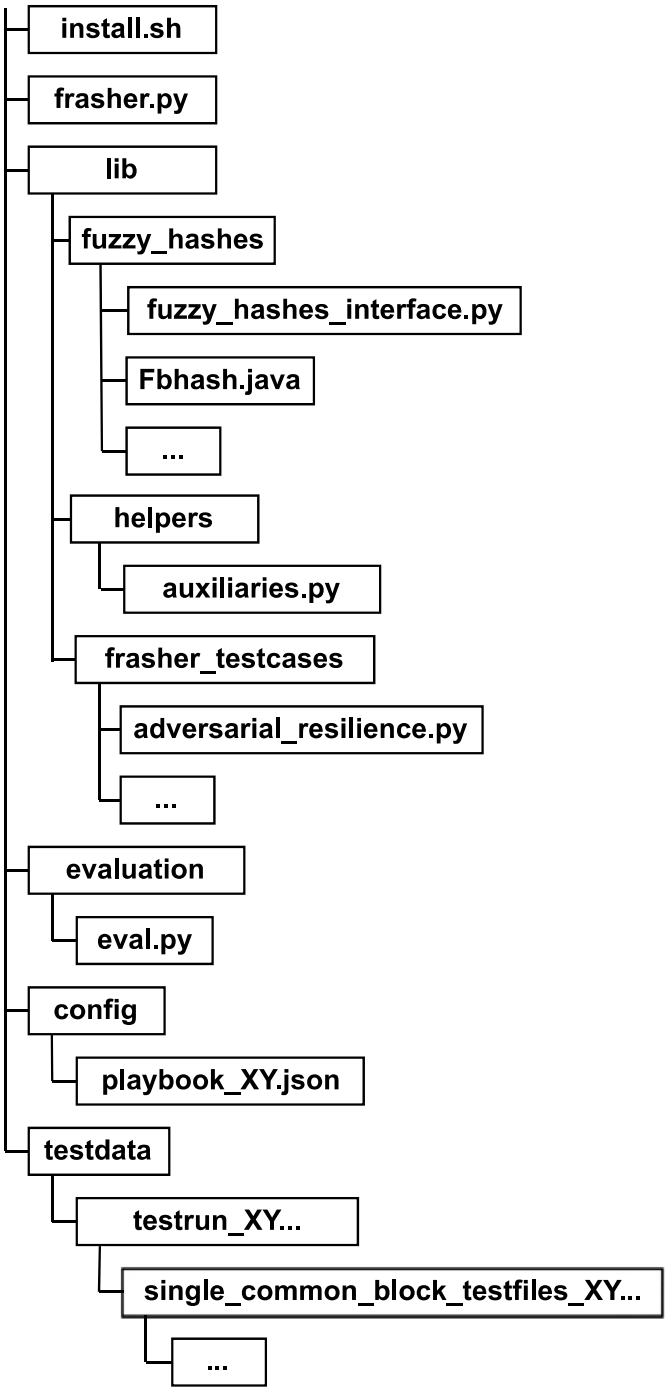

Fig. 3. Module-based structure of FRASHER.

The functional requirements are mostly reflected by the test cases. Fig. 2 gives an overview of the test cases available via FRASHER, which cover the functional requirements listed in Table 2.

5.2. *Using* FRASHER

For both the Linux and Windows implementations, a `MAKEFILE` is provided that installs all the necessary packages via the system's package installer and all required Python libraries via `pip`. In order to use all available algorithms and for optimal performance as a Linux distribution currently Ubuntu 21.04 is recommended.

In Section 5.1 the structure of a playbook was described. The user of FRASHER configures the evaluation (e.g., the algorithms, test cases and number of test rounds) via the JSON file to be executed. The framework can be invoked via its CLI command, which at the time of writing has the following syntax:

```
$ frasher.py [-h] [-v] [-d] PATH
```

−**h** prints usage instructions on the screen.
−**v** is the verbose mode that prints additional information during the test-runs.
−**d** subsequently visualises the results that have been added to a playbook.
PATH is the path to the playbook that specifies all further information for the tests.

The only mandatory argument is `path` to a playbook that holds all the necessary configuration information for the evaluation tests. The tests specified in the playbook section tests will be run in the given order using the specified test files as input. Therefore, all tests need to have their test files specified in advance. When files are manipulated, they are copied into separate folders following the syntax: `datetime_testname`. The existing test files in the folders are named according to the specifications of the tests. All test files are saved, so that tests can be repeated anytime, e.g., when using the same test data but different algorithms.

In order to add further algorithms, the user first has to anchor the new algorithm within the file *lib/fuzzy_hashes/algorithm.py*. This is where all the interfaces to the algorithms are stored. Next, the new algorithm must inherit from the class Algorithm, in which all the necessary functions of the test framework are defined, e.g., functions to create the filter, compare a file against another file or against the filter, etc. FRASHER offers the respective wrapper functions and skeleton files to simplify the integration of own developments.

Just as with the existing algorithms, new algorithms can either be implemented by using their Python libraries (if available), or by calling a precompiled executable, since separate Python libraries are not available for most algorithms. In the latter case, the string output of the algorithms is parsed, which in turn represents the input for the framework.

6. Sample evaluation results and assessment

In this section we present a proof of concept of FRASHER with respect to sample test cases. It summarises our practical findings for the algorithms `ssdeep`, `TLSH`, `mrsh-cf`, `MRSH-v2`, `sdhash` and `FbHash`.[2] We also provide sample output of the evaluation and visualisation component.

The test environment in which all our tests were conducted is as follows: VM with 16x Intel Xeon Platinum 8280 CPU @ 2.70 GHz, 2 × 16 GB DDR4-3200 RDIMM ECC, 800 GiB SSD, Ubuntu 21.10 Linux 5.13.0−40-generic x86_64.

We used the latest version of the algorithms available at the time of writing: `sdhash` v4.0, `MRSH-v2` v2.0, `ssdeep` v2.14.1, `TLSH` v4.11.2, `mrsh-cf` v1.0, `FbHash` v2.0.

Furthermore, Table 3 shows the data sets used. We used the *t5-corpus* for the efficiency test cases in Section 6.1. In order to also have compressed text file formats available in our *finding the needle* test cases in Section 6.4 (which uses real data as test candidates), we extended the original *t5-corpus* with the first 500 docx, first 400 pptx, and first 300 xlsx files (numerically ascending; to preserve the mixing ratio of office documents in the *t5-corpus*) from the *NapierOne* data set (Davies et al., 2022). For all remaining tests, we intentionally used (pseudo-)randomly generated data from the `/dev/urandom` device as proposed by Roussev (2011) to avoid that two independently generated targets have content in common. An embedded object is a smaller piece of randomly (and independently) generated data that is embedded in our target(s) with the explicit purpose of creating commonality.

Table 3
Data sets used for evaluation.

Data set	html	pdf	text	doc	ppt	jpg	xls	gif	docx	pptx	xlsx
t5-corpus	1093	1073	711	533	368	362	250	67	–	–	–
t5-corpus_extended	1093	1073	711	533	368	362	250	67	500	400	300

6.1. Efficiency test case

The playbook specifies both the number of test-runs and the test files used for all three efficiency tests. Our generation efficiency results are given in Table 4, they reveal `sdhash` as the most run-time efficient algorithm in terms of digest generation (i.e., to generate the respective Bloom filter). For example, in our case of the 1.9 GB large t5-corpus, the algorithm took ∼6.4 seconds to generate its filter on average after repeating the experiment in 10 iterations. We define the `sdhash` value with a relative generation efficiency of 1. Then, other algorithms provide a higher generation run-time with 42% generation time overhead with `MRSH-v2`, 68% overhead with `mrsh-cf`, 131% overhead with `TLSH`, 143% overhead with `ssdeep`. `FbHash` requires significantly more time to generate its filter.

The fingerprint comparison efficiency test was done for both an all-vs-all comparison and one-vs-all comparison. All tests were repeated 10 times and the results were averaged. In case of an all-vs-all test all unique hash pairs in the digest file should be compared (for n hashes, this yields $\frac{n\cdot(n-1)}{2}$ comparisons). In the one-vs-all test, the desired file (which can be specified in the playbook just like the corpus (in this case, the /t5/000001.doc was used) is compared with the previously generated filter (which leads to n comparisons for a data corpus with n files). The results in Table 4 for these two tests show that `TLSH` performs best for both an all-vs-all and an one-vs-all comparison. Further promising candidates in case of the chosen t5-corpus are `ssdeep` and `mrsh-cf`. Almost the same is true for the one-vs-all comparison, with `MRSH-v2` being rather slow. `FbHash` misses a switch for an all-vs-all comparison, while we also consider it unusable in a one-vs-all scenario for realistic use in digital forensics, since the comparison takes far too long. We are aware of a more time- and memory-efficient version of `FbHash` called `FbHash-E` (Singh et al., 2022), but it could not be tested because its source code was not published at the time of writing.

For compression, it is generally recommended to provide a compact similarity digest, as this usually enables a faster comparison and requires less storage space (Breitinger et al., 2014). The results of the compression efficiency test, as shown in Table 5, show that `ssdeep` and `TLSH` produce similar and high compression rates, while `MRSH-v2`, `mrsh-cf`, and `sdhash` create a less compressed output (worsening in this order). `FbHash` produces a very large digest file of almost 20 GB, which is why the one-vs-all comparison takes so long.

[2] The algorithm `Nilsimsa` (v.0.3.8) has been added to FRASHER in the meantime, but was not originally tested. Note that FRASHER supports the integration of further algorithms.

Table 4
Generation efficiency and *Comparison efficiency* in case of All-vs-All and One-vs-All comparisons; carried out 10 times each and averaged.

Algorithm	Digest generation (sec)	Comparison (all-vs-all) (sec)	Comparison (one-vs-all with /t5/000001.doc) (sec)
MRSH-v2	9.084s	144.746s	1.086s
sdhash	6.393s	110.592s	0.298s
TLSH	14.791s	2.617s	0.01s
FbHash	1838.597s	NA	463.721s
mrsh-cf	10.771s	12.569s	0.507s
ssdeep	15.576s	22.688s	0.01s

Table 5
Compression efficiency of tested algorithms using *t5-corpus* with total size of 1911.81 MB.

Algorithm	Digest file size (bytes)	Compression ratio (%)
MRSH-v2	28.67 MB	1.500%
sdhash	61.52 MB	3.218%
TLSH	394.11 KB	0.021%
FbHash	19.81 GB	1036.087%
mrsh-cf	33.55 MB	1.755%
ssdeep	485.45 KB	0,025%

6.2. Sensitivity & robustness

In this section, we present our results for the test cases *single-common-block correlation*, *fragment detection*, *alignment robustness*, as described in Section 4.2.

6.2.1. Single-common-block correlation test results

This test shows how well increasingly smaller common fragments within two, otherwise completely different files, can be matched, e.g., a common picture embedded in two PowerPoint presentations.

Our tests were performed with (pseudo-)random test files. This applies both to the container files as well as to the fragment itself. Note, this also means that there is a potentially large amount of features that each algorithm can reflect in its hash. If the fragment is taken arbitrarily from a real file, the number of features may be much smaller and the performance therefore different. To obtain a normalised result, 10 test-runs are performed and then the average values of the 10 test-runs are plotted into the output graph.

At the beginning of our test, the fragment is half the size of the two containers and is then reduced constantly in 16 KB steps of the initial fragment size.[3] Once the fragment is smaller than 1% of the two containers its size is further reduced in 100 bytes steps. This test was done for containers of sizes 512 KB, 2048 KB. The averaged graphs for the 512 KB and 2048 KB test-runs are depicted in Fig. 4 ssdeep can correlate the two files as long as the fragment size is $>(9\% - 13\%)$. Specifically, in case of 2048 KB test files, the correlation was no longer possible once the fragment went below $\sim 13\%$, in case of 512 KB any fragment below $\sim 9\%$ could no longer be matched.[4]

TLSH consistently exhibits a too high similarity score which decreases more slowly than the actual size of the single common fragment. The algorithm especially becomes error-prone when dealing with small fragments that are less than ca. 10% of the size of the original. This manifests itself in a very stagnant distance score (in our case similarity score) at the end of the curve.

mrsh-cf performs best with reliably detecting any correlation between two objects, as the common fragment size consistently matches the actual similarity score. Even if the common fragment size is less than 1%, mrsh-cf is able to correctly express the similarity between two containers to the second decimal place.

MRSH-v2 consistently returns a lower similarity score that misrepresents the actual amount of shared content in the containers. The same applies to sdhash.

FbHash is as reliable as mrsh-cf at assessing similarity. However it is no longer capable of detecting similarity when the single common fragment is smaller than 1%.

6.2.2. Fragment detection test results

The fragment detection test reveals the smallest fragment of an input that the algorithm can still reliably correlate with its original. We used the *random cutting* mode for our tests as an example, as described in Section 4.2. To obtain a normalised result, we use (pseudo-)randomly generated files with the same size as input files for each of the 10 test-runs. FRASHER then plots the average values of the 10 test-runs into the output graphs. In order to also evaluate the influence of the size, this procedure is repeated three times for files with 10 KB, 30 KB and 65 KB size, respectively.

The results are depicted in Fig. 5 and show that ssdeep struggles when more than 60% of the original file is removed. TLSH can no longer match a file of which more than 70% have been cut off. Moreover, the tests provide similar results for the different tested sizes. mrsh-cf outperforms its competitors in such a one-vs-one scenario as it consistently reflects through its similarity score the correct proportionality. The similarity score of MRSH-v2 always represents the similarity score of the smaller file which in this case leads to consistent values close to 100%. Once the cut off is bigger than 90% the similarity score approaches 0 rapidly. sdhash similarly returns a very high similarity that, although it decreases, does not resemble the true proportion of the relation between the original file and a fragment of it. In light of the fact that FbHash also consistently returns too high of similarity score, mrsh-cf is the only algorithm that performs well in this scenario.

6.2.3. Alignment robustness test results

The alignment robustness test analyses the impact of adding byte sequences to a file (e.g., a log file) and the ability of an algorithm to still correlate the enlarged file with the original file. In our test cases multiple blocks of random bytes were inserted sequentially at both the beginning (head) and the end (tail) of a file copy (i.e., in test case (1) bytes are only added at the beginning; in test case (2) bytes are only added at the end), up to a configurable maximum size, which is here set to 500% of the original file size. This means that genuinely new content was added to the original file. However, it should be mentioned that in a real world scenario with log files, the additional elements mostly adhere to the syntax of the preexisting log file, i.e., the file size might increase but the variance in file content stays the same. Any algorithm with a deduplication phase might therefore output different results in a real world scenario.

The results in Fig. 6 show the findings obtained when performing the alignment robustness test with test files that had an

[3] Please note that the exact step size is configurable via the playbook.

[4] Remember that the percentages are derived from a fragment that is already half the size (i.e., 50%) of the respective test files at the beginning.

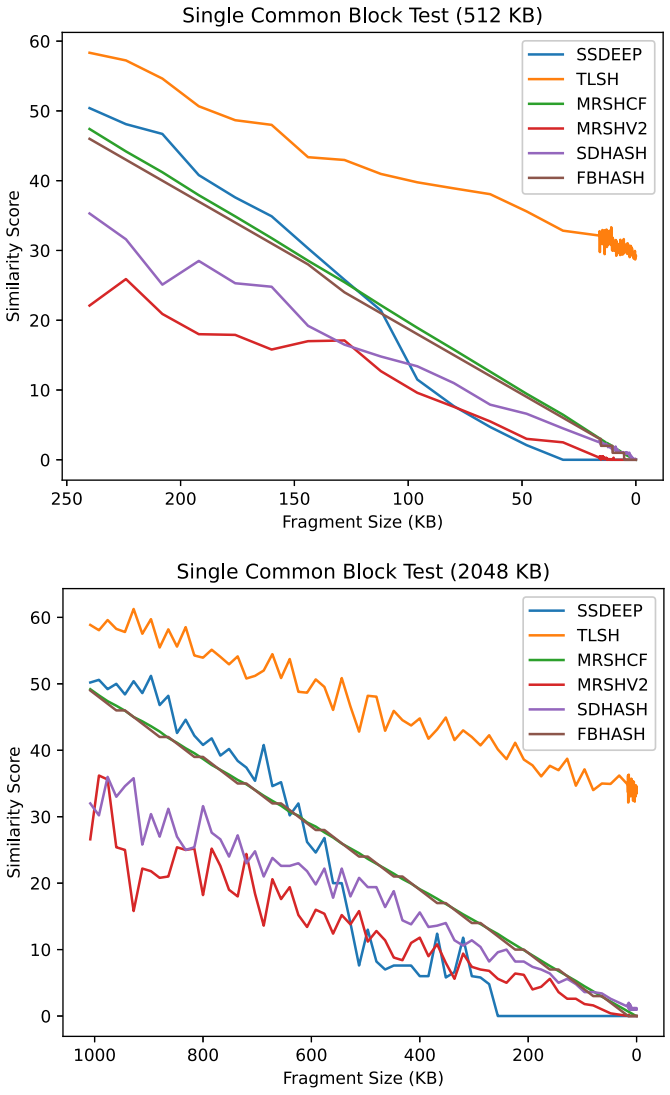

Fig. 4. *Single-common-block* test results and their visualisation in the case of 512 KB and 2048 KB file sizes.

original size of 30 KB. The tests were executed 10 times and averaged. In both test cases, i.e., for the head and tail scenario, `ssdeep` stops correlating the original and its manipulated copy at all as soon as ca. 300% of random bytes have been added to either head or tail. `TLSH` seems to perform similar as it is shown by its similarity score reaching 0 once ca. 300% of bytes have been pre- or appended to a file. `mrsh-cf` seems to perform well. However, its similarity score changes much less in the interval from 400% to 500% than it did in the interval from, e.g., 100%–200%. Its predecessor `MRSH-v2` behaves similarly but is prone to stagnate around certain similarity score values even with successive enlargement of the file.[5] `sdhash` continually returns high similarity scores that only serve as vague indicator of relation. This shows that its similarity score is not a reliably indicator of proportionality when matching a file that is embedded in another file that is multiple times its size. `FbHash` displays a continually decreasing similarity score that nevertheless is constantly too high. `mrsh-cf` is the algorithm who's similarity score is most closely reflecting how much the original file makes up

[5] Note that some of these detailed results were primarily taken from the CSV files, which are also created alongside the plotted diagrams.

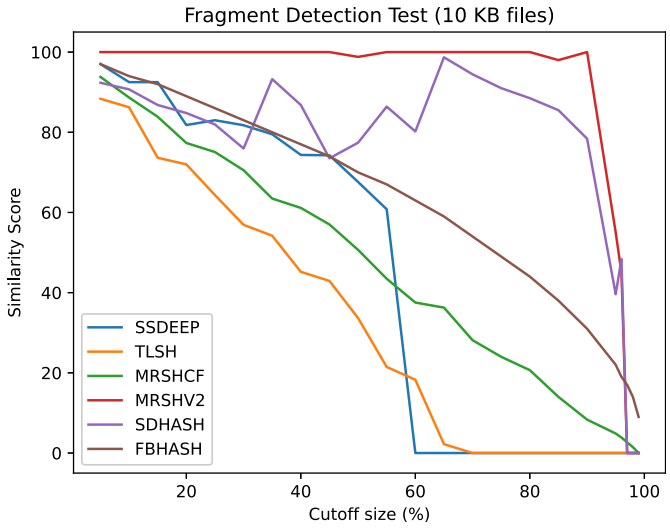

Fig. 5. *Fragment detection* test results and their visualisation in the case of 10 KB file size.

of the content of the enlarged test file. It therefore is the most alignment robust. All algorithms except for `sdhash` showed the same behaviour, regardless of whether the file was pre- or appended to. The same tests were performed with files of 10 KB and 60 KB size. The results did not differ significantly from those shown in Fig. 6.

6.3. Adversarial resilience

6.3.1. Digest generation impediment

With regard to adversarial resilience, the behavior of the algorithms in the presence of active impediment during the digest generation and digest comparison phases was examined. To investigate the algorithms resilience during the digest generation phase the algorithm was fed input files with successively smaller levels of diversity and size. This way the lower bounds for each algorithm in terms of variance and input size are explored. From an alphabet of 100 unique ASCII characters, random string sequences are created in which each character of the alphabet occurs at least once. Gradually the size of this alphabet is reduced while the length of the total string sequence remains the same. This test is performed with byte sequences of a minimum length of 10 up to 3000 bytes. The algorithms have to hash every given input and successfully compare with a similarity score >0 in order to succeed.

- `ssdeep` and `mrsh-cf` can hash and match every input starting from a 10 byte size which is the minimum size in this test.
- `TLSH` needs a minimum of 50 bytes and a variance of at least two unique characters in any input.
- `MRSH-v2` needs at least 2900 byte in input file size in order to hash any input that is made up of more than 3 characters. Any smaller file that is made up of between 1 and 100 unique characters cannot be reliable hashed.
- `sdhash` revealed itself to require a minimum input size of 512 bytes. In addition it has a lower bound regarding uniform content. A file has to be made up of at least 8 unique characters at a size of 750 bytes in order to be hashed and matched.
- `FbHash` requires a variance of at least 3 characters in any input file in order to be able to hash and match it.

This test reveals that in case of `MRSH-v2` and `sdhash` its easily feasible for an attacker to produce files of considerable size and

small variance that cannot be hashed or compared by any of these two algorithms. This could enable an attacker to get files through any filter that these two algorithms are part of.

6.3.2. Digest comparison impediment

To test the resilience of the algorithms during the digest comparison phase, the algorithms ability to deal with repetitions in the content of input files is examined. This test is similar to an alignment robustness test but instead of adding genuinely new content the file is appended to itself. Ideally any algorithm should be able to sustain its matching capabilities in face of repetitive content, meaning that it should either return a similarity score that incalculates the repetition and gets smaller as the file grows, or a similarity score that reflects the entirety of shared content (100% similarity score even when file b is just a multiple of file a). For this test 5 KB of a file are chained together multiple times and all algorithms have to hash and compare the original file and the multiplication of its content. This test was done for both the file types contained in the extended *t5-corpus* (that includes compressed text formats, i.e., xlsx, docx, pptx) and for random generated files. The trend of the results is the same for all data. Table 6 shows our results for random generated files.

The algorithms general tendency expressed in Table 6 is observable for all file types and shows that the similarity score for the algorithms `ssdeep`, `TLSH` and `mrsh-cf` (with decreasing sensitivity in this order) can be lowered through deliberate duplication. In the case of `ssdeep`, a similarity score of zero can be achieved in some cases when chaining together four instances of the same file.

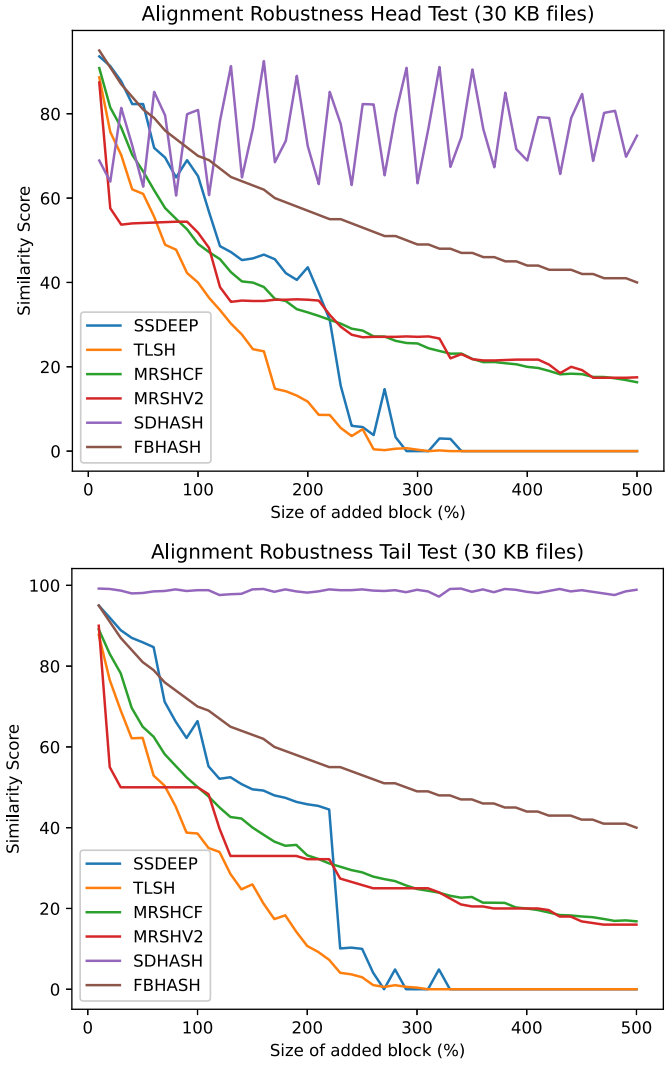

Fig. 6. *Alignment robustness* test results and their visualisation with random blocks added to either head or tail of a file.

6.4. Finding the needle

The results of the *needle in a haystack* test cases show that success depends on the file type and varies from algorithm to algorithm. As an example, we explain the results of the test case *Resemblance - Object similarity detection* as explained in Section 4.4. In this test case, there exist 10 needles that challenge the algorithm's ability to find a file that has undergone changes among many other files. Each test was repeated ten times, using ten different input files for each needle. `FbHash` is not evaluated in this scenario. Its poor run-time efficiency means that it takes 15 min to perform a one-vs-all test on the extended t5-corpus that is 3.29 GB large and includes compressed text formats (i.e., xlsx, docx, pptx).[6] For our test, the needles are created in the following way:

- *needle_1* and *needle_2* are created through cutting of the first and last 20% of a file, similar to the *fragment detection* test.
- *needle_3* and *needle_4* are generated by overwriting the first and last 50% of a file with random bytes.
- *needle_5* and *needle_6* are created by prepending and appending 200% of the files original size in random bytes as in the *alignment robustness* test.
- *needle_7* and *needle_8* are created by inserting a block that has 50% of the length of the original file from an unrelated file outside of the haystack but of the same file type. The blocks are inserted in different offsets, once at the beginning (*needle_7*) and once at the end (*needle_8*).
- *needle_9* is created by compressing the original file using the `Zlib` library.
- *needle_10* is created by cutting the first and last 20% of a file, deleting the header and footer information from a file.

In Table 7 in Appendix A we present the test results of the *finding the needle* test for all file types. The results reveal how many out of ten files that were manipulated as needles the algorithms managed to find among the extended *t5_corpus*. The results reveal that compression as in *needle_9* is a problem for all algorithms regardless of the file type. This needle was not correctly found in most cases. Compressed text file formats, such as `.docx, .xlsx, .pptx`, however, are not any more challenging for the algorithms as their uncompressed counterparts `.doc, .xls, .ppt`. Overall, for the five algorithms tested so far, the `mrsh-cf` and `MRSH-v2` algorithms are consistently able to find the most needles regardless of the actual file type, with `MRSH-v2` performing slightly better. The third most accurate algorithm is `ssdeep` which performs better than `TLSH` (the fourth best) in most cases. Both algorithm fail in most cases to identify needle_5 and needle_6, both of which test the alignment robustness. `sdhash` performs significantly worse than the others, it can mostly only detect two needles (needle_2 and needle_6).

[6] i.e., the combined time of performing the following test would have taken 10 days with `FbHash`'s current speed.

Table 6
Similarity scores for the same 5000 bytes of randomly generated data; concatenated several times; averaged over 10 test-runs.

Multiplication factor	Similarity scores					
	ssdeep	TLSH	mrsh-cf	MRSH-v2	sdhash	FbHash
x 1 (5 KB)	100	100	100	100	100	100
x 2 (10 KB)	69	67	96.3	100	100	99
x 4 (20 KB)	0	39	94.3	100	100	99
x 8 (40 KB)	0	11	93.3	100	100	99
x 16 (80 KB)	0	0	92.8	100	100	99
x 32 (160 KB)	0	0	92.6	100	100	99

6.5. Summary

Our sample evaluations reveal valuable insights into the capabilities of the similarity digest algorithms at hand (ssdeep, TLSH, mrsh-cf, MRSH-v2, sdhash, FbHash). With the test cases of single-common-block correlation, fragment detection and alignment robustness, FRASHER is capable of determining the limitations of the algorithms in an idealised setting using mostly randomly generated files.

Using the *finding the needle* test cases, FRASHER also examines the performance of the AM algorithms with real world data, which has revealed that mrsh-cf and MRSH-v2 hold up well against the much more widely known ssdeep.

Singh (2021) made the point that there is no conclusive recommendation as to which algorithm is preferable for a given use case. With this work revealing that mrsh-cf is preferable when trying to find files that have changed (revealed through needle in a haystack - object similarity detection), it is also important to consider the ubiquitous use of mrsh-cf or MRSH-v2.

7. Conclusion and future work

This article reviewed the requirements, design, and test cases of an automated evaluation framework in the scope of AM. We introduced FRASHER as an open-source, Python, automatic AM evaluation framework. It was inspired by its predecessor FRASH, however, it is updated in terms of modularity, programming language, and test scenarios. In addition, FRASHER simplifies the evaluation process through its visualisation component, which produces CSV files and plots graphs from the taken measurements and thus helps to identify the most suitable candidate depending on the respective problem. We additionally provided a demonstration of FRASHER by performing sample test cases.

This paper is specifically intended to make FRASHER publicly available to the digital forensics community. Additionally we provide detailed test results for sample test cases for six different AM algorithms, with seven AM algorithms now integrated in the framework. As future work, we plan mainly to provide further test results and their in-depth analysis that include more details. Specifically, we will extend our tests to other AM algorithms and examine further test results for the attacks presented, focusing on the design weaknesses of various algorithms (e.g., how an adversary can outsmart them, how to artificially increase or decrease the similarity score of an algorithm).

An additional aspect we plan to research is the use of FRASHER as a synthesis framework for test data that actively challenges the capabilities of the algorithms. Recently, fuzzy hashes have been combined with machine learning applications to detect patterns in hashes that implicate the presence of malware in the respective files (Martín-Pérez et al., 2021; Peiser et al., 2020; Lazo, 2021). With our framework, we are already able to detect threats to the validity of machine learning models that use fuzzy hashes as input files. In this area of research, it is of utmost importance to know what weaknesses individual hashes have, as they can be actively exploited in attacks on the models.

Appendix A. *Finding the needle* test results

The results of the *finding the needle* test are listed in Table 7.

Table 7
Results for different *needle in a haystack* test cases.

Algorithm	File Type	needle_1	needle_2	needle_3	needle_4	needle_5	needle_6	needle_7	needle_8	needle_9	needle_10
ssdeep	PDF	10/10	10/10	10/10	10/10	2/10	1/10	10/10	10/10	0/10	10/10
	DOC	10/10	10/10	9/10	8/10	2/10	3/10	6/10	7/10	0/10	10/10
	GIF	10/10	10/10	10/10	10/10	2/10	2/10	10/10	10/10	0/10	10/10
	HTML	9/10	10/10	5/10	5/10	0/10	0/10	9/10	9/10	0/10	9/10
	JPG	10/10	10/10	10/10	10/10	5/10	9/10	9/10	9/10	5/10	10/10
	PPT	10/10	10/10	10/10	10/10	8/10	6/10	10/10	10/10	0/10	10/10
	TEXT	10/10	10/10	4/10	9/10	2/10	1/10	10/10	10/10	0/10	9/10
	XLS	10/10	10/10	5/10	6/10	0/10	0/10	8/10	9/10	0/10	10/10
	DOCX	10/10	10/10	10/10	10/10	6/10	8/10	9/10	8/10	1/10	10/10
	XLSX	10/10	10/10	10/10	10/10	7/10	5/10	10/10	8/10	0/10	10/10
	PPTX	10/10	10/10	9/10	10/10	5/10	6/10	9/10	10/10	0/10	10/10
TLSH	PDF	8/10	6/10	9/10	1/10	0/10	0/10	8/10	9/10	0/10	0/10
	DOC	5/10	5/10	5/10	4/10	0/10	0/10	1/10	1/10	0/10	2/10
	GIF	10/10	10/10	10/10	10/10	0/10	0/10	10/10	10/10	2/10	8/10
	HTML	10/10	10/10	7/10	9/10	6/10	5/10	5/10	7/10	0/10	10/10
	JPG	9/10	10/10	10/10	8/10	0/10	0/10	9/10	9/10	4/10	10/10
	PPT	8/10	5/10	10/10	5/10	0/10	0/10	7/10	7/10	5/10	4/10
	TEXT	10/10	10/10	9/10	6/10	0/10	1/10	8/10	7/10	0/10	10/10
	XLS	5/10	8/10	8/10	6/10	1/10	3/10	9/10	9/10	0/10	5/10
	DOCX	10/10	10/10	9/10	9/10	0/10	0/10	5/10	6/10	1/10	9/10

(continued on next page)

Table 7 (*continued*)

Algorithm	File Type	needle_1	needle_2	needle_3	needle_4	needle_5	needle_6	needle_7	needle_8	needle_9	needle_10
	XLSX	10/10	8/10	10/10	10/10	0/10	0/10	1/10	1/10	0/10	9/10
	PPTX	9/10	9/10	9/10	10/10	0/10	0/10	9/10	7/10	4/10	7/10
mrsh-cf	PDF	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	0/10	10/10
	DOC	10/10	10/10	10/10	10/10	10/10	10/10	9/10	9/10	4/10	10/10
	GIF	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	0/10	10/10
	HTML	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	0/10	10/10
	JPG	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	5/10	10/10
	PPT	10/10	9/10	10/10	9/10	10/10	9/10	10/10	10/10	9/10	8/10
	TEXT	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	0/10	10/10
	XLS	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	0/10	10/10
	DOCX	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	1/10	10/10
	XLSX	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	0/10	10/10
	PPTX	10/10	10/10	7/10	10/10	10/10	10/10	9/10	9/10	10/10	10/10
MRSH-v2	PDF	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	0/10	10/10
	DOC	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	4/10	10/10
	GIF	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	0/10	10/10
	HTML	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	0/10	10/10
	JPG	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	3/10	10/10
	PPT	10/10	9/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	8/10
	TEXT	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	0/10	10/10
	XLS	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	0/10	10/10
	DOCX	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	1/10	10/10
	XLSX	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	0/10	10/10
	PPTX	10/10	10/10	10/10	10/10	10/10	10/10	10/10	10/10	9/10	10/10
sdhash	PDF	0/10	10/10	0/10	0/10	0/10	10/10	0/10	0/10	0/10	0/10
	DOC	0/10	10/10	0/10	0/10	0/10	10/10	0/10	0/10	0/10	0/10
	GIF	1/10	5/10	0/10	0/10	1/10	7/10	0/10	0/10	0/10	6/10
	HTML	0/10	10/10	0/10	0/10	0/10	10/10	0/10	0/10	0/10	0/10
	JPG	0/10	10/10	0/10	0/10	0/10	10/10	0/10	0/10	0/10	0/10
	PPT	0/10	10/10	0/10	0/10	0/10	10/10	0/10	0/10	0/10	0/10
	TEXT	0/10	10/10	0/10	0/10	0/10	10/10	0/10	0/10	0/10	0/10
	XLS	0/10	10/10	0/10	0/10	0/10	10/10	0/10	0/10	0/10	0/10
	DOCX	0/10	8/10	0/10	0/10	0/10	9/10	0/10	0/10	0/10	0/10
	XLSX	0/10	8/10	0/10	0/10	1/10	10/10	0/10	0/10	0/10	0/10
	PPTX	0/10	10/10	0/10	0/10	0/10	10/10	0/10	0/10	0/10	0/10

References

Baier, H., Breitinger, F., 2011. Security aspects of piecewise hashing in computer forensics. In: 2011 Sixth International Conference on IT Security Incident Management and IT Forensics, pp. 21–36. https://doi.org/10.1109/IMF.2011.16.

Breitinger, F., Baier, H., 2011. Performance issues about context-triggered piecewise hashing. In: Gladyshev, P., Rogers, M.K. (Eds.), Digital Forensics and Cyber Crime - Third International ICST Conference, ICDF2C 2011, Dublin, Ireland, October 26-28, 2011, Revised Selected Papers, pp. 141–155. https://doi.org/10.1007/978-3-642-35515-8_12. Springer volume 88 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*.

Breitinger, F., Baier, H., 2012a. Properties of a similarity preserving hash function and their realization in sdhash. In: 2012 Information Security for South Africa, pp. 1–8. https://doi.org/10.1109/ISSA.2012.6320445.

Breitinger, F., Baier, H., 2012b. Similarity preserving hashing: eligible properties and a new algorithm mrsh-v2. In: Rogers, M.K., Seigfried-Spellar, K.C. (Eds.), Digital Forensics and Cyber Crime - 4th International Conference, ICDF2C 2012, Lafayette, IN, USA, October 25-26, 2012, Revised Selected Papers, pp. 167–182. https://doi.org/10.1007/978-3-642-39891-9_11. Springer volume 114 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering.

Breitinger, F., Baier, H., Beckingham, J., 2012. Security and implementation analysis of the similarity digest sdhash. In: First International Baltic Conference on Network Security & Forensics *(nesefo)*.

Breitinger, F., Guttman, B., McCarrin, M., Roussev, V., White, D., et al., 2014. Approximate Matching: Definition and Terminology. NIST Special Publication, pp. 10–6028, 800.

Breitinger, F., Roussev, V., 2014. Automated evaluation of approximate matching algorithms on real data. Digit. Invest. 11, S10–S17. https://doi.org/10.1016/j.diin.2014.03.002.

Breitinger, F., Stivaktakis, G., Baier, H., 2013. Frash: a framework to test algorithms of similarity hashing. Digit. Invest. 10, S50–S58. https://doi.org/10.1016/j.diin.2013.06.006 (The Proceedings of the Thirteenth Annual DFRWS Conference).

Chang, D., Sanadhya, S.K., Singh, M., Verma, R., 2015. A collision attack on sdhash similarity hashing. In: Proceedings of 10th Intl. Conference on Systematic Approaches to Digital Forensic Engineering, pp. 36–46.

Davies, S.R., Macfarlane, R., Buchanan, W.J., 2022. Napierone: a modern mixed file data set alternative to govdocs1. Forensic Sci. Int.: Digit. Invest. 40, 301330. https://doi.org/10.1016/j.fsidi.2021.301330. URL: https://www.sciencedirect.com/science/article/pii/S2666281721002560.

Garfinkel, S., Farrell, P., Roussev, V., Dinolt, G., 2009. Bringing science to digital forensics with standardized forensic corpora. Digit. Invest. 6, S2–S11. https://doi.org/10.1016/j.diin.2009.06.016 (The Proceedings of the Ninth Annual DFRWS Conference).

Gupta, V., Breitinger, F., 2015. How cuckoo filter can improve existing approximate matching techniques. In: James, J.I., Breitinger, F. (Eds.), Digital Forensics and Cyber Crime. Springer International Publishing, Cham, pp. 39–52.

Kornblum, J., 2006. Identifying almost identical files using context triggered piecewise hashing. Digit. Invest. 3, 91–97. https://doi.org/10.1016/j.diin.2006.06.015. The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06).

Lazo, E.G., 2021. Combing through the Fuzz: Using Fuzzy Hashing and Deep Learning to Counter Malware Detection Evasion Techniques. *Microsoft 365 Defender Research Team*. URL: https://www.microsoft.com/security/blog/2021/07/27/combing-through-the-fuzz-using-fuzzy-hashing-and-deep-learning-to-counter-malware-detection-evasion-techniques/.

Lee, A., Atkison, T., 2017. A comparison of fuzzy hashes: evaluation, guidelines, and future suggestions. In: *Proceedings of the SouthEast Conference* ACM SE '17. Association for Computing Machinery, New York, NY, USA, pp. 18–25. https://doi.org/10.1145/3077286.3077289.

Martín-Pérez, M., Rodríguez, R.J., Breitinger, F., 2021. Bringing order to approximate matching: classification and attacks on similarity digest algorithms. Forensic Sci. Int.: Digit. Invest. 36, 301120. https://doi.org/10.1016/j.fsidi.2021.301120.

Martínez, V.G., Álvarez, F.H., Encinas, L.H., 2014. State of the art in similarity preserving hashing functions. In: 2014 International Conference on Security and Management (SAM'14), pp. 139–145. Worldcomp 2014.

Oliver, J., Forman, S., Cheng, C., 2014. Using randomization to attack similarity digests. In: Batten, L., Li, G., Niu, W., Warren, M. (Eds.), Applications and Techniques in Information Security. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 199–210.

Peiser, S.C., Friborg, L., Scandariato, R., 2020. Javascript malware detection using locality sensitive hashing. In: Hölbl, M., Rannenberg, K., Welzer, T. (Eds.), ICT Systems Security and Privacy Protection. Springer International Publishing, Cham, pp. 143–154.

Ribeiro, B., Moia, V.H.G., Henriques, M.A.A., 2017. Similarity Digest Search: A Survey and Comparative Analysis of Strategies to Perform Known File Filtering Using Approximate Matching. Security and Communication Networks, 1306802. https://doi.org/10.1155/2017/1306802, 2017.

Roussev, V., 2010. Data fingerprinting with similarity digests. In: Chow, K., Shenoi, S. (Eds.), *Advances in Digital Forensics VI - Sixth IFIP WG 11.9 International Conference on Digital Forensics, Hong Kong, China, January 4-6, 2010, Revised Selected Papers* (Pp. 207–226). https://doi.org/10.1007/978-3-642-15506-2_15. Springer volume 337 of *IFIP Advances in Information and Communication Technology*.

Roussev, V., 2011. An evaluation of forensic similarity hashes. Digit. Invest. 8, S34–S41. https://doi.org/10.1016/j.diin.2011.05.005.

Roussev, V., Richard, G.G., Marziale, L., 2007. Multi-resolution similarity hashing. Digit. Invest. 4, 105–113. https://doi.org/10.1016/j.diin.2007.06.011.

Singh, M., 2021. Essential Characteristics of Approximate Matching Algorithms: A Survey of Practitioners Opinions and Requirement Regarding Approximate Matching, 10087 arXiv:2102.

Singh, M., Khunteta, A., Ghosh, M., Chang, D., Sanadhya, S.K., 2022. Fbhash-e: a time and memory efficient version of fbhash similarity hashing algorithm. Forensic Sci. Int.: Digit. Invest. 41, 301375. https://doi.org/10.1016/j.fsidi.2022.301375. URL: https://www.sciencedirect.com/science/article/pii/S2666281722000543.