Forensic Science International: Digital Investigation 36 (2021) 301120

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2021 EU - Proceedings of the Eighth Annual DFRWS Europe

Bringing order to approximate matching: Classification and attacks on similarity digest algorithms

Miguel Martín-Pérez [a], Ricardo J. Rodríguez [a, *], Frank Breitinger [b]

[a] Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, Spain
[b] Hilti Chair for Data and Application Security, Institute of Information Systems, University of Liechtenstein, Fürst-Franz-Josef-Strasse, 9490, Vaduz, Liechtenstein

ARTICLE INFO

Article history:

Keywords:
Approximate matching
Fuzzy hashing
Similarity hashing
Similarity digest algorithm
Bytewise
Classification scheme

ABSTRACT

Fuzzy hashing or similarity hashing (a.k.a. bytewise approximate matching) converts digital artifacts into an intermediate representation to allow an efficient (fast) identification of similar objects, e.g., for blacklisting. They gained a lot of popularity over the past decade with new algorithms being developed and released to the digital forensics community. When releasing algorithms (e.g., as part of a scientific article), they are frequently compared with other algorithms to outline the benefits and sometimes also the weaknesses of the proposed approach. However, given the wide variety of algorithms and approaches, it is impossible to provide direct comparisons with all existing algorithms. In this paper, we present the first classification of approximate matching algorithms which allows an easier description and comparisons. Therefore, we first reviewed existing literature to understand the techniques various algorithms use and to familiarize ourselves with the common terminology. Our findings allowed us to develop a categorization relying heavily on the terminology proposed by NIST SP 800-168. In addition to the categorization, this article presents an abstract set of attacks against algorithms and why they are feasible. Lastly, we detail the characteristics needed to build robust algorithms to prevent attacks. We believe that this article helps newcomers, practitioners, and experts alike to better compare algorithms, understand their potential, as well as characteristics and implications they may have on forensic investigations.

1. Introduction

According to NIST SP 800-168, "approximate matching is a promising technology designed to identify similarities between two digital artifacts" (Breitinger et al., 2014a). This identification of similarities between two or more artifacts can happen on three different levels of abstraction: *bytewise*, when the comparison relies on the raw sequence of bytes that form the digital artifacts; *syntactic*, when the internal structures of the digital artifacts under analysis are used instead of merely byte sequences; or *semantic*, when the comparison relies on contextual attributes to interpret the digital artifacts and estimate their similarity. Furthermore, algorithms may either compare artifacts directly (e.g., Levenshtein distance or Hamming distance), or they may first convert them into an intermediate representation (e.g., a fingerprint, hash, digest) that can then be compared. This latter case is often referred to as fuzzy hashing or similarity hashing and aims at complementing cryptographic hash functions by allowing identifying *similar* objects instead of completely identical objects.

In this article we focus on algorithms/literature that operate on the byte-level[1] and utilize an intermediate representation, i.e., a digest/fingerprint. We define these kinds of algorithms as *similarity digest algorithms* (SDA)[2] which is the acronym used for the remainder of this article. These algorithms gained popularity around 2006 when `ssdeep` was published by Kornblum (2006). Over the years, many more algorithms have been proposed such as `sdhash` by Roussev (2010), `MRSH-V2` by Breitinger and Baier (2012b) or `TLSH` by Oliver et al. (2013), to name a few.

In order to compare algorithms, the community mostly focuses

* Corresponding author.
E-mail address: rjrodriguez@unizar.es (R.J. Rodríguez).

[1] Inputs/artifacts are treated as a byte stream and are processed without any interpretation of the data.
[2] In this paper, we use the SDA interchangeably as a singular and plural acronym.

https://doi.org/10.1016/j.fsidi.2021.301120

on obvious metrics such as runtime efficiency or precision and recall rates. However, due to the various design decisions researchers and practitioners have made during the development, we argue that a more fine granular comparison is necessary as there may be instances where precision and recall are insufficient. For instance, some implementations have difficulties handling extremely small files, while others are susceptible if the difference in file size between two objects is too large (e.g., 5 MiB vs. 5 GiB). Consequently, this article has the following contributions:

- The first categorization for SDA, allowing the community to better discuss and compare the various existing algorithms. Categorizations are useful for scientific fields, as they allow structuring a domain.
- A comprehensive discussion of the algorithms with respect to the categorization and its implication for practitioners.
- A discussion of the categorization with respect to why these characteristics are important and how practitioners may contribute from it, describing an abstract set of attacks.
- In addition, we also provide insights on the desirable properties to build a robust SDA against attacks.

The rest of this paper is organized as follows: Subsequently, we introduce the Terminology and Background which outlines terms that are utilized throughout this article. The core of this work is the Towards a Classification Scheme for Similarity Digest Algorithms section which presents our developed categorization. Attacks against Similarity Digest Algorithms outlines some attacks that are possible on algorithms followed by the section on Building a Robust Similarity Digest Algorithm describing the desired properties to build a robust SDA, according to our findings. The last section concludes this article.

2. Terminology and Background

In comparison to traditional and cryptographic hash functions that mostly divide an input into blocks and use bit operations like XOR to generate a final, fixed-length hash value (fingerprint), SDA require a more sophisticated processing. Often these algorithms select features (sometimes also referred to as chunks in literature), compress them and merge them to a final signature (similarity digest). For this work, we define the following terminology which is similar to NIST SP 800-168 (Breitinger et al., 2014a):

1 ***Features*** are the basic characteristics that can be extracted from digital artifacts and allow the comparison between two or more objects. Sample features could be a single bit, byte sequences, or offsets in an object.
2 ***Mapping functions*** allow the processing of features, e.g., to compress them or to encipher them using cryptographic hashing. Note, NIST as well as other literature frequently utilize the term *compression functions* as this is the most common behavior of this function. However, we feel that mapping function is more accurate, as theoretically the function could also expand the feature. We refer to the features obtained as result of compression functions as *processed* features.
3 ***Similarity digests*** are the final output of SDA and can be seen as an aggregation of (processed) features.
4 ***Similarity functions*** allow comparing two similarity digests and returns a similarity score, which is often a numerical value. This value is then denoted by *similarity score* (note, although it is a numerical value that often ranges between 0 and 100, it is not necessary a percentage value).

Background. Prior works primarily focused on directly comparing two or more algorithms and mostly rely on the metrics such as runtime efficiency or precision and recall (Roussev (2011); Lee and Atkison (2017)). On the other hand, some researchers dedicated their time to inspecting implementations in all detail. For instance, Breitinger et al. (2012) ran various tests on `sdhash` and found that the implementation does not consider every byte for the similarity digest generation, which they denote by *coverage*. As a result, it is possible that two similarity digests are completely identical despite the artifacts been (slightly) different. Likewise, NIST SP 800-168 discusses various use cases as well as properties of special interest (Breitinger et al., 2014a).

However, there is not a clear, well-established way to provide direct comparisons between all existing algorithms. We aim at filling this gap with this article. To the best of our knowledge, we are the first to establish a classification of SDA to facilitate the description and comparison of these algorithms.

3. Towards a Classification Scheme for Similarity Digest Algorithms

To develop our classification scheme, we surveyed the widely used as well as some niche algorithms and implementations as summarized in Table 1. In the following we provide a more detailed description of our Methodology followed by the Phases of a Similarity Digest Algorithm section. The core of this section is Proposed Classification Scheme.

3.1. Methodology

In order to derive our classification scheme, we reviewed relevant literature, i.e., descriptions of the various algorithms, as well as secondary literature such as comparisons of algorithms, security evaluations, or suggested properties for approximate matching (e.g., as suggested by Breitinger and Baier (2012a)). Our starting point were articles discussing properties and security features of algorithms. Thus, we started with more general/broader articles followed by articles describing specific algorithms and implementations. For each article, we extracted relevant information (characteristics) describing behavior, features and peculiarities of algorithms. Lastly, we tried to align these characteristics as good as possible. As result, we obtained our proposed classification scheme as summarized in Table 2.

3.2. Phases of a Similarity Digest Algorithm

Similar to traditional hash functions, SDA have two major working stages:

1. During the **artifact processing and digest generation phases**, an algorithm receives data (i.e., a byte sequence) as input, processes the input and returns a similarity digest. In detail, there are several steps: First, features are extracted from the input (*feature generation phase*), which are then further processed (*feature processing*). Some algorithms may have an optional phase to select the features that will compose the digest (*feature selection phase*). If this phase does not exist, all features form the digest. In addition, some algorithms may have the optional phase to delete duplicate features (*features deduplication phase*). This phase may be associated with the next stage when it uses digests instead of features as input. Lastly, processed features are transformed to form the similarity digest (*digest generation phase*), which is the final output of the SDA. Note, the order of phases is not fixed, e.g., an algorithm may first select features and then do the processing.

Table 1
Proposed classification for similarity digest algorithms; the last two columns are discussed in Section 3.4.

Algorithm	Reference	Previous classification	New classification
dcfldd	Harbour (2002)	Block-Based Hashing	Feature Sequence Hashing
ssdeep	Kornblum (2006)	Context Trigger Piecewise Hashing	Feature Sequence Hashing
md5bloom	Roussev et al. (2006)	Context Trigger Piecewise Hashing	Feature Sequence Hashing
MRS hash	Roussev et al. (2007)	Context Trigger Piecewise Hashing	Feature Sequence Hashing
sdhash	Roussev (2010)	Statistically-Improbable Features	Feature Sequence Hashing
MRSH-V2	Breitinger and Baier (2012b)	Context Trigger Piecewise Hashing	Feature Sequence Hashing
SimHash	Sadowski and Levin (2007)	Block-Based Rebuilding	Byte Sequence Existence
mvHash-B	Breitinger et al. (2013)	Block-Based Rebuilding	Byte Sequence Existence
LZJD	Raff and Nicholas (2017)	(none)	Byte Sequence Existence
Nilsimsa	Damiani et al. (2004)	Locality-Sensitive Hashing	Locality-Sensitive Hashing
TLSH	Oliver et al. (2013)	Locality-Sensitive Hashing	Locality-Sensitive Hashing
saHash	Breitinger et al. (2014b)	(none)	Locality-Sensitive Hashing
FbHash	Chang et al. (2019)	(none)	Locality-Sensitive Hashing

Table 2
Categorization scheme for similarity digests algorithms. The possible values of each dimension/procedure are separated by semicolons.

	Phase	Dimension/ Procedure	Characteristic
Artifact processing and digest generation phases	Feature generation	Length	Static; Dynamic
		Support function	Trigger function; Unique; None
		Intersection	Yes; No
		Cardinality	Fixed; Variable
	Feature processing	Mapping function	Hashing; Encoding; Identifier; None
		Bit reduction	Ratio; None
	Feature selection	Selection function	Minimum probability; Block matching; Block similarity; Minimum value; None
		Domain	Feature; Processed feature
		Coverage	Full; Partial
	Digest generation	Digest size	Fixed; Input dependent; Input dependent with max
		Storing structure	Processed feature concatenation; Set concatenation; Set; Counter
		Order	Absolute; Set-absolute; Processed feature-aware; None
		Requirements	Minimum features; Diversity; Document frequency; None
	Feature deduplication	Type	Consecutive; In-scope; None
		Occurrence phase	Digest generation; Digest comparison
	Digest comparison	Requirements	Minimum commonality; Minimum amount; Similar input size; None
		Output score	Binary value; Interval; Half-bounded
		Score trend	Ascending; Descending
		Space sensitivity	Partial; Total; None

2. The **digest comparison phase** is independent from the previous phases and requires two similarity digests as input (obtained from the same SDA). Using some similarity metric, the digests are compared and a similarity score is returned. This score provides insights about the similarity (or dissimilarity) between the digests.

3.3. Proposed Classification Scheme

These previously mentioned phases led to our classification scheme as summarized in Table 2 where we identified various dimensions/procedures for each phase each having various characteristics.

3.3.1. Feature Generation Phase

To start, SDA process the input on the byte level with the aim to identify and extract features. We distinguish between the following dimensions/procedures: The feature ***length*** can be either of a predefined fixed length (*static*) or can be *dynamic* (variable). In the latter case, features are usually identified with a ***support function***[3] allowing to identify the feature boundaries. Currently, we differentiate between two different support functions. A *trigger function* compares its output with a predefined value to determine the feature boundaries. On the other hand, a *unique* function can be used to build a set of unique features from the input using set theory.[4] When the feature size is static, an SDA may not need a support function to identify boundaries. These cases are denoted by *none*. An example for static features would be dcfldd which uses 512 bytes. In contrast, ssdeep and MRSH-V2 use trigger functions (rolling hash) to identify the features which results in variable sized features.

The ***intersection*** dimension in the feature generation can be *yes* (a byte may belong to more than one feature) or *no* (a byte can belong to exactly only one feature). Examples would be ssdeep and sdhash which have distinct and overlapping features,

[3] Note, it is important to differentiate between a support function and feature selection (phase 3). For instance, sdhash uses static features of 64 bytes (no support function); and uses a selection function to choose best ones.

[4] One may consider this as part of the feature selection phase. However, in this particular case the feature changes if it is in the set wherefore we placed it in feature generation.

Table 3
Classification of similarity digest algorithms according to our proposed classification scheme (*feature generation*, *feature processing*, and *feature selection* phases).

Algorithm	Feature generation				Feature Processing		Feature Selection		
	Length	*Support Function*	*Intersection*	*Cardinality*	*Mapping Function*	*Bit Reduction*	*Selection Function*	*Domain*	*Coverage*
dcfldd	Static (512)	None	No	Variable (L/512)	Hashing	None (128)	None	(n/a)	Full
Nilsimsa	Static (3)	None	Yes	Variable (6L)	Hashing	None (8)	None	(n/a)	Full
ssdeep	Dynamic (L/ 64)	Trigger function	No	Fixed (64)	Hashing	Ratio (6/32)	None	(n/a)	Full
md5bloom	Static (512)	None	No	Variable (L/512)	Hashing	Ratio (40/128)	None	(n/a)	Full
MRS hash	Dynamic (234)	Trigger function	No	Variable (L/234)	Hashing	Ratio (44/128)	None	(n/a)	Full
SimHash	Static (1)	None	Yes	Variable (8L)	Identifier	None (8)	Block matching	Feature	Partial
sdhash	Static (64)	None	Yes	Variable (L)	Hashing	Ratio (55/160)	Minimum probability	Feature	Partial
MRSH-V2	Dynamic (320)	Trigger function	No	Variable (L/320)	Hashing	Ratio (55/64)	None	(n/a)	Full
mvHash-B	Static (20, 50)	None	Yes	Variable (8L)	Encoding	Ratio (1/32)	Block similarity	Feature	Full
TLSH	Static (3)	None	Yes	Variable (6L)	Hashing	None (8)	None	(n/a)	Full
saHash	Static (1)	None	Yes	Variable (4L)	None	None (8)	None	(n/a)	Full
LZJD	Dynamic (1+ $\log_{256}L$)	Unique	No	Variable (L/(1 +$\log_{256}L$))	Hashing	None (128)	Minimum value	Processed feature	Partial
FbHash	Static (7)	None	Yes	Variable (L)	Hashing	None (64)	None	(n/a)	Full

respectively.

The forth dimension is ***cardinality***, which describes the amount of features SDA attempt to produce for a given input which can be *fixed* or *variable* where it frequently depends on the input length L. For instance, ssdeep aims at having 64 features while dcfldd and MRSH-V2 produce L/512 and L/320, respectively. In other words, the cardinality is strongly related to the feature-length dimension.

3.3.2. Feature Processing Phase

After identifying features, there may be a processing phase which includes two dimensions. A ***mapping function*** (or compression function as suggested by NIST) applies some form of mapping to the feature which can be of the characteristic: hashing, encoding or identifier. We only found one algorithm, saHash, that does not apply any compression function as its feature size is already small. Most commonly, SDA use *hashing* to obtain pseudo-random processed features where cryptographic as well as non-cryptographic algorithms are used. For instance, MRSH-V2 uses FNV-1a function (Fowler et al., 2011) to reduce features from 320 bytes to 64 bits while sdhash relies on SHA-1. Instead of hashing, the mapping function might be a more general *encoding* function to process the features. For instance, mvHash-B applies a majority vote step followed by Run Length Encoding on compress bit sequences. The last approach that we found is a feature *identifier* function (that works as a mapped feature) and generates feature identifiers of the selected features from a fixed set. For instance, SimHash yields an identifier for every feature that matches with one of its 16 fixed features.

The mapping function is often followed by a ***bit reduction*** procedure which consists of selecting few bytes (or bits) from the mapped feature. This bit reduction can be expressed as a *ratio* between the input and the output. For instance, the MRSH-V2's input to the bit reduction function is a 64 bit FNV-1a hash that results in a 55 bits output. Hence, the ratio is 55/64. Similarly, the sdhash inputs are 160 bit SHA-1 hashes, while the outputs are 55 bits, which results in a ratio of 55/160. We have used *none* when the SDA does not have a bit reduction procedure (that is, all the bits of the mapped feature are used). For completeness in Table 3, in these cases we have written between parenthesis the output size (in bits).

3.3.3. Feature Selection Phase

This phase, performed by some algorithms, selects specific features (e.g., the most unique ones) by applying a ***selection function*** which can be based on:

- *minimum probability* (i.e., most unique), when the features least likely to occur are selected as they are considered as the most significant features (e.g. sdhash).
- *block matching*, when a predefined set of blocks are used to seek and replace the sequence of features by a sequence of block identifiers (e.g. SimHash).
- *block similarity*, when the most similar blocks are identified (e.g. mvHash-B).
- *minimum values*, when the algorithm selects a subset of processed features with the lowest values (e.g. LZJD).

The ***domain*** of these selection function is either *feature* or *processed feature*. For instance, sdhash selects features by their entropy and it then calculates the processed features, which are finally added to the digest. On the contrary, LZJD first calculates the processed features and then selects which ones will form the digest.

Coverage reflects if all bytes of the input are considered in the similarity which can be *full* (like ssdeep or MRSH-V2) or *partial* (like sdhash, where there may be gaps between features).[5] In the latter case, it is important to understand that two inputs may yield a perfect SDA match but they may be not be identical (i.e., they have different cryptographic hash values).

3.3.4. Digest Generation Phase

This phases encompasses the construction of a similarity digest. The ***digest size*** can be *fixed* when it is always the same length, regardless the input length (e.g., FbHash or saHash); *input dependent*, when there is a correlation between the input size and the digest size (e.g., dcfldd concatenates MD5 hashes for each feature of 512 bytes) or *input dependent with max* when the size depends on the input length but has a maximum length (e.g.,

[5] Although we included *coverage* in the "feature selection phase," one should note that partial coverage can also be caused by the "feature generation phase" which may ignore bytes.

ssdeep yields a digest within 32 and 64 characters but may not reach its max for small inputs).

Regarding the ***storing structure***, it can be a *processed feature concatenation* (i.e., the processed features are just concatenated used by dcfldd or ssdeep) or it can be *set concatenation* which describes the idea of adding processed features to a set with a maximum capacity. When the set reaches its capacity (is full), a new empty set is appended. Up till now, all implementations of the *set concatenation* method use Bloom filters and the comparison process described by Roussev et al. (2006); Roussev (2010, 2012). When the number of processed features has a cap, they can be stored in a single *set* like in the case of LZJD. Lastly, some algorithms (e.g. Nilsimsa and SimHash) aggregate features during the selection phase which is denoted by *counters*.

The storing ***order*** between features is also a characteristic of SDA. The possible values are: *absolute*, when the position of every feature is conserved (for instance, dcfldd or ssdeep store all features in order); *set-absolute*, when the generated digest keeps the order between different feature sets, but is unaware of the feature order within a particular set. For example, the SDA that use set concatenation (based on Bloom filters) for feature storing have this characteristic value, as the Bloom Filters do not keep the order (e.g. md5bloom); *processed feature-aware*, when the storing structure does not keep the order of the features contained within, but the processed feature itself helps determine the order. Thus, the intersection between features allows for detecting changes (i.e., the similarity score varies) when the input is rearranged (e.g., TLSH and saHash); or *none*, when the SDA does not consider the order between features (e.g., SimHash and LZJD).

Some algorithms have ***requirements*** to yield a digest. The requirements can be either to reach *minimum features*, to have enough *diversity* on the input, or to have a *document frequency* that contains the frequency of every feature on a training set. When SDA do not have requirements, we use *none*. In this regard, some algorithms have a feedback phase, during which they perform configuration adjustments and restart the process from the beginning when the generated digests do not meet the minimum requirements expected.

3.3.5. Features Deduplication Phase

The *features deduplication phase* is an optional phase implemented by some algorithms to eliminate duplicate or redundant processed features. The ***type*** can be *consecutive*, when several consecutive features are reduced to a short sequence or *in-scope*, when identical features in the same scope are eliminated. When this phase does not exist, the type is *none*. This phase can take place (***occurrence phase***) in two different places, either during the *digest generation* or during the *digest comparison* phases. For example: algorithms using Bloom Filters (BF) deduplicate by design during the generation phase as BFs behave like sets and do not store duplicate features.

3.3.6. Digest Comparison Phase

This phase allows comparing two digests which results in a similarity score, i.e., how similar or dissimilar are two digests. In order to return a similarity (dissimilarity) score, some algorithms impose one or more ***requirements***. The digest comparison may require either a *minimum commonality* between the digests, a *minimum amount* of features in every digest, or a *similar input size* of both inputs to compare them. Examples are provided in Section 3.3.7.

The ***output score*** is the similarity score between two given digests and can be either a *binary* value, i.e., inputs are similar or not (yes/no). Additionally, the output score could be an *interval* (usually between $[0, 1]$ or $[0, 100]$) or *half-bounded*. The latter case indicates that there is a lower (or upper) boundary which is common when measuring dissimilarity where 0 is (almost identical) but there is no upper boundary. Related to the output score is the ***score trend*** which can be *ascending* (the higher the output score, the higher similarity of digests) or *descending*.

Another characteristic of the comparison function is its ***space sensitivity***, which expresses whether the function is sensitive to a different order of the same features. This sensitiveness can be *total*, when the total order of the features is considered, or *partial*, when it only considers the order of features partially.

3.3.7. Examples: ssdeep and sdhash

To illustrate our proposed classification scheme, we applied the scheme to two relevant algorithms:

ssdeep uses a rolling hash as a *support function* to divide the input into roughly 64 distinct features (*length*, *intersection*, and *cardinality*). Once the features are generated, they are hashed using FNV hash (*mapping function*) and the six least relevant bits (*bit reduction*) generate a base64-encoded character. As all processed features belong to the digest (i.e., *full coverage*), the feature selection phase is omitted. During the digest generation process, the remaining input after the 63rd feature is considered as the last feature (i.e., its *digest size* is fixed). The digest is built by concatenating the processed features (*order* and *structure*). If the number of features is less than 32 (*requirements*), ssdeep does not yield any digest and restarts the whole process, readjusting the rolling hash (*support function*) accordingly. For the digest comparison phase, ssdeep first reduces the consecutive processed features that are duplicated to three characters (*type* and *occurrence*) and then evaluates if there are at least 7 consecutive common features between the digests to perform the comparison (*requirements*). The concatenation of features implies the sensitiveness to the feature order (*space sensitive*). The similarity score is an interval $[0, 100]$, where 0 indicates no similarity while 100 indicates identical inputs (*output score* and *score trend*).

sdhash extracts an arbitrary number of features, considering 64 overlapped bytes from a given input (*feature size*, *intersection*, and *cardinality*), without any prior input processing (*support function*). Then, the feature selection takes place, in which a subset of features (*domain*) are chosen based on their entropy (*selection function* and *coverage*). Next, sdhash hashes the selected features using SHA-1 hash (*mapping function*) and uses 55 bits (*bit reduction*) of the 160 bits of the SHA-1 hash. Then the processed features are inserted into Bloom filters (*structure* and *order*), starting the digest generation phase. When a filter is full, a new filter is created and appended to the digest (*digest size* is input dependent). However, when the entropy is low and no features are selected, sdhash is unable to yield any digest (*requirements*). Additionally, if a processed feature is already contained in the current filter, it is discarded (*type* and *occurrence*; features deduplication phase). Regarding the digest comparison phase, the digest comparison function calculates similarity based on the similarity between Bloom filters, considering only the highest similarity between every pair of filters. Furthermore, sdhash requires 16 features per filter to compute the similarity between two filters (*requirements*). As the Bloom filters do not keep any order in the elements contained in the filter, the algorithm is unable to know if two set of features keep the same order (*space sensitivity*). As for ssdeep, the similarity score is an interval $[0, 100]$, where 0 indicates no similarity while 100 indicates identical inputs (*output score* and *score trend*).

3.4. Classification of state-of-the-art SDA

Table 3 and Table 4 classify the state-of-the-art SDA that we have considered for our paper in chronological order. In total, we

Table 4
Classification of similarity digest algorithms according to our proposed classification scheme (*digest generation*, *feature deduplication*, and *digest comparison* phases).

Algorithm	Digest generation				Feature Deduplication		Digest comparison			
	Digest Size	*Storing Structure*	*Order*	*Requirements*	*Type*	*Occurrence*	*Requirements*	*Output Score*	*Score Trend*	*Space Sensitivity*
`dcfldd`	Input dependent	Processed feature concatenation	Absolute	None	None	(n/a)	None	Interval	Ascending	Total
`Nilsimsa`	Fixed	Counter	Processed feature-aware	None	Consecutive	Comparison	Minimum commonality	Interval	Ascending	None
`ssdeep`	Input dependent with max	Processed feature concatenation	Absolute	Minimum features	Consecutive	Comparison	Minimum commonality, Similar input size	Interval	Ascending	Total
`md5bloom`	Input dependent	Set concatenation	Set-absolute	None	In-Scope	Generation	None	Interval	Ascending	Partial
`MRS hash`	Input dependent	Set concatenation	Set-absolute	None	In-Scope	Generation	None	Interval	Ascending	Partial
`SimHash`	Fixed	Counter	None	None	None	(n/a)	Similar input size	Half-bounded	Descending	None
`sdhash`	Input dependent	Set concatenation	Set-absolute	Diversity	In-Scope	Generation	Minimum amount	Interval	Ascending	Partial
`MRSH-V2`	Input dependent	Set concatenation	Set-absolute	None	In-Scope	Generation	Minimum amount	Interval	Ascending	Partial
`mvHash-B`	Input dependent	Set concatenation	Set-absolute	Diversity	In-Scope	Generation	Similar input size	Interval	Ascending	Partial
`TLSH`	Fixed	Counter	Processed feature-aware	None	None	(n/a)	None	Half-bounded	Descending	None
`saHash`	Fixed	Counter	Processed feature-aware	None	None	(n/a)	None	Binary value	(n/a)	Total
`LZJD`	Fixed	Set	None	None	None	(n/a)	None	Interval	Ascending	None
`FbHash`	Fixed	Counter	Processed feature-aware	Document frequency	None	(n/a)	None	Interval	Ascending	None

reviewed 13 algorithms released between 2002 and 2019. For readability, we have split the results into two tables according to the phases outlined in Table 2: Table 3 contains the *feature generation*, *feature processing*, and *feature selection* phases; Table 4 details the *digest generation*, *feature deduplication*, and *digest comparison* phases.

Before discussion existing algorithms with respect to our classification, let us have a look at exiting classifications which are mostly based on categories creators assigned to their algorithms. Based on Gayoso Martínez et al. (2014); Lee and Atkison (2017); Moia and Henriques (2017), there are the following categories:

1 **Block-based hashing** which consists of algorithms that use cryptographic hashes, generating and storing features for every block of a fixed size;
2 **Context trigger piecewise hashing** which is made up of algorithms that split the input into contexts, defined as a sliding window on the input bytes when the trigger function is activated;
3 **Statistically-improbable features** which comprises algorithms that use a selection function based on statistically improbable features, as its own name indicates;
4 **Block-based rebuilding** which consists of algorithms that choose blocks (randomly selected or pre-fixed) and generate the digests selecting the most similar blocks to the input; and
5 **Locality-sensitive hashing** which is made up of algorithms that map objects into buckets, grouping similar objects in the same bucket with high probability.

However, these categories only consider specific aspects of the algorithms, instead of a complete view on the full behavior of the algorithms. This produces a misunderstanding in how an algorithm works, which eventually can lead to wrong decisions when selecting an SDA for a specific purpose (e.g., comparing algorithms with totally different behavior, or even not comparing them with other similar algorithms). Therefore, we propose a new and simpler classification based on the complete behavior as follows:

1 **Feature Sequence Hashing:** this category encompasses the algorithms that split the input into features and maps them, measuring the similarity by feature sequences.
2 **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.
3 **Locality-Sensitive Hashing:** finally, this category is as in the previous classification. It is made up of algorithms that map objects into buckets, grouping similar objects in the same bucket with high probability.

Table 1 shows the algorithms that we consider in this paper, presenting their previous and new classification according to our proposal. For every algorithm we also detail their references.

4. Attacks against Similarity Digest Algorithms

In this section, we study some possible attacks against SDA and the characteristics of the algorithms facilitating these attacks. We do not claim that this section is complete and other attacks may exist. We distinguished between two types of attacks:

- Attacks against the similarity score, which can be separated in the sections Reduction of Similarity and Emulation of Similarity.
- Attacks against impeding the last phases of an SDA, which can be divided into the sections Impeding the Digest Generation Phase and Impeding the Digest Comparison Phase.

Adversary model. We assume an intelligent adversary who is knowledgeable about the processes and techniques used by SDA. Thus, the adversary is able to classify the SDA according to our classification scheme, either by reverse engineering it, by performing code source analysis, or by revising the literature/documentation of the algorithm.

For the sake of simplicity, our attack scenarios assume an SDA with an interval score where the low value (0) indicates no similarity and the high value (100) indicates perfect similarity.

4.1. Reduction of Similarity

Our first attack aims to minimize the similarity score between two inputs. Consequently, the adversary is interested in crafting a

new digital artifact from a given artifact knowing that they will be eventually be compared (e.g., blacklisting). In summary, if an SDA utilizes the following dimension/procedure characteristics, exploiting it in this way is possible:

- *Feature length*: static; *support function*: none.
- *Mapping function*: hash function.
- *Storing structure*: set concatenation.
- *Requirements (digest comparison)*: minimum commonality.
- *Selection function*: block matching or minimum value.
- *Selection function*: block similarity; *Intersection*: yes.

Each of the listed points is explained in the upcoming paragraphs. You may interpret the above listing as follows (we illustrate it with the first bullet point): "*if an SDA utilizes a static feature length and does not make use of a support function, exploiting it is possible.*"

One of the most trivial attacks for SDA consists in splitting an input into features of a static size (not using any support function), adding one byte at the beginning of the input will modify all subsequent features (all offsets shift). This kind of attack, for example, was described in (Baier and Breitinger, 2011) against `dcfldd`.

If the mapping function of the SDA is a hash function, a straightforward attack requires changing 1-bit in each (or at least the majority) of the features. Remember that hash functions have an avalanche effect, meaning that despite how similar two inputs are, their outputs will differ by roughly 50% of the bits. This property of hash functions, which is the reason why they are commonly used for data integrity and file identification of a seized device in digital forensics (Harichandran et al., 2016), becomes a challenging aspect for similarity digest algorithms. Likewise, Oliver et al. (2014) demonstrated that random changes on the input can reduce the similarity score. In particular, they evaluated these attacks against `ssdeep`, `sdhash`, and `TLSH`.

It may be possible to optimize the previously mentioned exploit as demonstrated by (Baier and Breitinger, 2011) on `ssdeep`. The authors showed that `ssdeep` requires a minimum commonality (digest comparison requirement) of no less than 7 consecutive common features. If an adversary knows which features are going to be extracted from a digital artifact, s/he is required to modify 1 out of 7 features for dropping the similarity score to zero. Recall, the trigger function of `ssdeep` generates up to 64 features, and thus modifying at most $\lfloor 64/7 \rfloor = 9$ bytes is sufficient.

The SDA using set concatenation are currently relying on the Bloom filter storing structure. While this is a space-efficient probabilistic data structure, it has one issue often denoted as shifting meaning that half of the features of one filter are displaced to the next filter. For instance, imagine two BF, BF_1 and BF_2, each containing 10 features. If an adversary adds 5 features to BF_1 (e.g., add prepending data to the artifact), their overlap (similarity) drops significantly. The real world impact was discussed by (Breitinger and Baier, 2012a) who showed that this attack drops the similarity of approximately 28.

The SDA that select features based on block matching or minimum values can be defeated searching these particular set of blocks or features and modifying them accordingly. For instance, `SimHash` counts the amount of 16 blocks of 1 byte (Sadowski and Levin, 2007). If an adversary seeks these blocks and modifies them in the crafted input, the similarity score reduces. Similarly, `LZJD` stores the 1000 lowest hashes of the features (Raff and Nicholas, 2017). An adversary who detects the input characteristics used to form these features can modify them to reduce the similarity between objects.

Lastly, SDA based on block similarity (selection function) and features overlap (intersection dimension yes) is susceptible to this kind of attack as well. An intelligent adversary can forge the input to obtain a block that is similar to another block. For example, `mvHash-B` uses two fixed blocks to calculate the block similarity, while it counts the consecutive equal features to generate the digest. An adversary can attack this algorithm just changing the amount of consecutive features (incrementing or decrementing it by one). As a consequence, this will cause that the subsequent derived features change (Singh et al. (2016)).

4.2. Emulation of Similarity

The counter attack to the previous section is *emulation of similarity*, where an adversary is interested in crafting a new digital artifact that yields a high similarity score to a known artifact (e.g., allow list). In summary, if an SDA utilizes the following dimension/procedures characteristics, exploiting it is possible:

- *Cardinality*: fixed; *Support function*: trigger function.
- *Coverage*: partial.

If the SDA has a fixed cardinality combined with a trigger function as a mapping function, an adversary can craft an input that generates several small features with the beginning of the input filling up the similarity digest. For instance (Baier and Breitinger, 2011), showed that it is possible to include trigger sequences (i.e., features) in the EXIF image data. Consequently, the authors were able to manipulate an image to match any given similarity digest.

Another issue with SDA using partial coverage is that an adversary may be able to modify the gaps being cautious that these modifications do not alter the feature generation and selection phase. For instance, according to Breitinger et al. (2012), up to 20% of the content of an input can be modified without influencing the generated digest of `sdhash`. This claim was later demonstrated by Chang et al. (2015). Likewise, `SimHash` is also vulnerable to this attack: an adversary can modify any byte of the input if these modified bytes not match with any fixed block. Similarly, in `LZJD` any data can be altered after the last selected feature, as long as it does not generate a processed feature with a value lower than the maximum of the selected processed features.

4.3. Impeding the Digest Generation Phase

Here the goal of the adversary is to complicate the digest generation phase, i.e., to modify an input in a way that the algorithm is unable to generate a similarity digest due to the lack of necessary conditions. In summary, if an SDA utilizes the following dimension/procedures characteristics, exploiting it is possible:

- *Support function*: trigger function; *Requirements (digest generation)*: minimum features.
- *Requirements (digest generation)*: diversity.

The SDA using a trigger function as a support function to generate features and needing to reach a minimum amount of features as requirements to yield a similarity digest are vulnerable to this kind of attack. This is possible because the trigger function can be manipulated by means of a forged input to yield insufficient features for generating the digest. For example, `ssdeep` can be attacked with an input that intentionally avoids byte sequences that match with the value of the trigger function. As a consequence, only few features are generated and `ssdeep` cannot create a digest. Note, algorithms may implement countermeasures like adjusting the support function appropriately.

Other algorithms need input diversity to generate a digest. This requirement can also be exploited by an intelligent adversary. For instance, `sdhash` discards all the features and does not generate a digest if the entropy of input features is too low. Likewise,

mvHash-B compares the features with two blocks and keeps the identifier of the most similar block. The generation of digest is based on how many consecutive identifiers of a given type are found. Hence, if all features are similar to a unique block, there is only one sequence of identifiers which is then insufficient to generate the digest. This attack may require many changes and therefore may be impractical for real world scenarios.

4.4. Impeding the Digest Comparison Phase

Lastly, we contemplate an adversary aiming to hamper the digest comparison process. To achieve this, an adversary crafts an input so that the similarity digest generated cannot be compared or, if comparable, the similarity score is low. In summary, if an SDA utilizes the following dimension/procedures characteristics, exploiting it is possible:

- *Type*: consecutive; *Requirements (digest comparison)*: minimum commonality.
- *Requirements (digest comparison)*: minimum amount.

Algorithms having a feature deduplication phase of the type consecutive and also need a minimum commonality as a requirement to compare digests are vulnerable to this kind of attack. An intelligent adversary may forge an input such that the digest, computed after deduplication, is smaller than the number of common elements expected and therefore, the digest is incomparable. For instance, ssdeep is vulnerable to this attack.

Likewise, if SDA require a minimum number of elements to compare digests, an adversary can forge an input such that the algorithm yields a similarity digest with less elements than required for comparison. For instance, sdhash needs 16 features per Bloom Filter to compare two filters (Roussev and Quates, 2013). However, as its selection function (feature selection phase) is based on entropy, an adversary may manipulate an input to have low entropy so that the features are discarded. If the input yields a similarity digest with only one filter and less than 16 features, the comparison is impossible. Similar attacks are possible against MRS hash and MRSH-V2: an adversary can forge an input such that the matching of the trigger function is avoided, generating so few features that the sets unmeet the comparison requirements.

5. Building a Robust Similarity Digest Algorithm

The following outlines the characteristics that we consider desirable to build a robust SDA where robust is defined with respect to resilience against attacks. Note, we are not focusing on perfect error rates nor does this section consider runtime efficiency.

The feature length should be small, static, and with intersections (overlapping (intersection yes) allows detecting the swapping of features). The algorithms that use a dynamic feature size need a support function that splits the input in some way. This function is however susceptible to be attacked, and thus it should be avoided (this may be negligible if it generates an exhaustive amount of features).

Regarding the feature processing phase, the use of a hash as a mapping function implies that a small change in a feature (e.g., modifying just one bit) causes a totally different processed feature. Hence, one may preferably use a mapping function that generates similar outputs for similar inputs (maybe some sort of recursive SDA). Bit reduction is acceptable if the output provided by the mapping function output is large. In any event, the final amount of used bits should be somewhat resistant to collision attacks and should preserve the similarity relationship between outputs. Regarding the feature selection phase, a full coverage is desirable to impeder that uncovered gaps could be used to hide data and still obtain good similarity scores.

The desirable characteristics for the digest are to have an input dependent size, absolute order, and without any requirements. The requirements in the digest generation phase impose conditions on which an attacker can focus to hamper the digest generation process. Likewise, a deduplication process is undesirable because it deletes information.

Note, the desired digest size is hard to answer. As stated by Breitinger et al. (2014a), a fixed size digest is preferable. However, our study of SDA revealed that is difficult to design a robust algorithm with this characteristic. We have found three SDA that have this characteristic, but they have some limitations. For instance, the use of features with dynamic length and a fixed size digest has problems when comparing inputs with very different size (for example, ssdeep). Likewise, the selection of a limited cardinality of features as a representation of the whole (as LZJD does) implies a partial coverage, leaving gaps that could be exploited by an attacker inserting arbitrary content. Last, counting features in a limited set of counters (as TLSH or Nilsimsa do) provokes the loss of the order between features, which allows for the input rearranging without affecting to the similarity score. This issue is partially solved by considering feature intersection (processed feature-aware). The last limitation is the solution less vulnerable to attacks, but still we believe that to keep the order between features with an input-dependent digest is more valuable than having a better performance for using a fixed size digest.

Regarding the digest comparison phase, to have comparison requirements facilitate an attacker to obstruct the comparison process, as happened before. Last, a comparison function with total space sensitivity is also desirable as it allows to identify modifications in the order of features.

6. Conclusions

Over the past decade, many approximate matching a.k.a. similarity digest algorithms have been released to the digital forensics community. However, there is a lack of a clear classification scheme which makes it difficult to compare them. Therefore, this article first discussed the relation between approximate matching algorithms and similarity digest algorithms where the latter case requires an intermediate representation (e.g., a fingerprint, a similarity digest) that can be compared. Focusing on SDA, this article then introduced a classification scheme that facilitates the description and comparison of these algorithms which can be helpful for newcomers, practitioners and experts to discuss approaches.

In order to develop the classification, we identified and describe the main six phases of SDA where five fall under "artifact processing and digest generation phase" and the last one is required to compare digests. Each phase consists of various dimensions & procedures which themselves are based on characteristics. For instance, the *feature generation phase* has among other the dimensions/procedures *length, support function* or *intersection* which can have the characteristics *static* or *dynamic* (length) or *trigger function* or *unique* (support function). Next, we enumerated several promising approaches that have been proposed during recent years. In addition, this work presented a set of attacks against SDA with respect to the classification scheme. We also highlight the desired properties that an SDA shall have to be robust against these attacks.

As future work, we aim to develop proofs of concepts of the attacks discussed in this paper. We also aim to design and implement a robust SDA considering the issues discussed previously in this article.

Acknowledgments

The research by Miguel Martín-Pérez and Ricardo J. Rodríguez was supported in part by the Spanish Ministry of Science, Innovation and Universities under grant MEDRESE RTI2018-098543-B-I00 and by the University, Industry and Innovation Department of the Aragonese Government under *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-20R). The research by Miguel Martín-Pérez was also supported by the Spanish National Cybersecurity Institute (INCIBE) "Ayudas para la excelencia de los equipos de investigación avanzada en ciberseguridad", grant numbers INCIBEC-2015-02486 and INCIBEI-2015-27300.

References

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

Breitinger, F., Astebøl, K.P., Baier, H., Busch, C., 2013. mvHash-B - a new approach for similarity preserving hashing. In: 2013 Seventh International Conference on IT Security Incident Management and IT Forensics, pp. 33–44.

Breitinger, F., Baier, H., 2012a. Properties of a similarity preserving hash function and their realization in sdhash. In: 2012 Information Security for South Africa. IEEE, pp. 1–8.

Breitinger, F., Baier, H., 2012b. Similarity preserving hashing: eligible properties and a new algorithm MRSH-v2. In: Rogers, M., Seigfried-Spellar, K.C. (Eds.), Digital Forensics and Cyber Crime. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 167–182.

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, pp. 1–16.

Breitinger, F., Guttman, B., McCarrin, M., Roussev, V., White, D., 2014a. Approximate Matching: Definition and Terminology. Techreport NIST Special Publication 800-168. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-168.

Breitinger, F., Ziroff, G., Lange, S., Baier, H., 2014b. Similarity hashing based on Levenshtein distances. In: Peterson, G., Shenoi, S. (Eds.), Advances in Digital Forensics X. IFIP Advances in Information and Communication Technology, vol. 433. Springer, Berlin, Heidelberg, pp. 133–147. https://doi.org/10.1007/978-3-662-44952-3_10.

Chang, D., Ghosh, M., Sanadhya, S.K., Singh, M., White, D.R., 2019. FbHash: a new similarity hashing scheme for digital forensics. Digit. Invest. 29, S113–S123. https://doi.org/10.1016/j.diin.2019.04.006. http://www.sciencedirect.com/science/article/pii/S1742287619301550.

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

Damiani, E., di Vimercati, S.D.C., Paraboschi, S., Samarati, P., 2004. An open digest-based technique for spam detection*. In: Proceedings of the 2004 International Workshop on Security in Parallel and Distributed Systems, pp. 559–564.

Fowler, G., Noll, L.C., Vo, K.P., Eastlake, D., Hansen, T., 2011. The FNV Non-cryptographic Hash Algorithm. Ietf-draft.

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

Harbour, N., 2002. Dcfldd version 1.0. http://dcfldd.sourceforge.net/. (Accessed 3 October 2020).

Harichandran, V.S., Breitinger, F., Baggili, I., 2016. Bytewise approximate matching: the good, the bad, and the unknown. J. Dig. Forens., Secur. Law 11.

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.

Lee, A., Atkison, T., 2017. A comparison of fuzzy hashes: evaluation, guidelines, and future suggestions. In: Proceedings of the SouthEast Conference. Association for Computing Machinery, New York, NY, USA, pp. 18–25. https://doi.org/10.1145/3077286.3077289.

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. Secur. Commun. Network. 2017, 1–17. https://doi.org/10.1155/2017/1306802.

Oliver, J., Cheng, C., Chen, Y., 2013. TLSH – a locality sensitive hash. In: 2013 Fourth Cybercrime and Trustworthy Computing Workshop. IEEE, pp. 7–13.

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.

Raff, E., Nicholas, C., 2017. An alternative to NCD for large sequences, lempel-Ziv Jaccard distance. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Association for Computing Machinery, New York, NY, USA, pp. 1007–1015. https://doi.org/10.1145/3097983.3098111, 10.1145/3097983.3098111.

Roussev, V., 2010. Data fingerprinting with similarity digests. In: Chow, K.P., Shenoi, S. (Eds.), Advances in Digital Forensics VI. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 207–226.

Roussev, V., 2011. An evaluation of forensic similarity hashes. Digit. Invest. 8, S34–S41. https://doi.org/10.1016/j.diin.2011.05.005 (the Proceedings of the Eleventh Annual DFRWS Conference).

Roussev, V., 2012. Managing terabyte-scale investigations with similarity digests. In: Peterson, G., Shenoi, S. (Eds.), Advances in Digital Forensics VIII. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 19–34.

Roussev, V., Chen, Y., Bourg, T., Richard, G.G., 2006. md5bloom: forensic filesystem hashing revisited. Digit. Invest. 3, 82–90. https://doi.org/10.1016/j.diin.2006.06.012 the Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06).

Roussev, V., Quates, C., 2013. Sdhash 3.4. https://github.com/sdhash/sdhash/blob/master/sdbf/sdbf_defines.h#L58. (Accessed 11 March 2020).

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.

Sadowski, C., Levin, G., 2007. SimHash: Hash-Based Similarity Detection. Technical Report. University of California, Santa Cruz.

Singh, M., Chang, D., Sanadhya, S.K., 2016. Security analysis of MVhash-B similarity hashing. J. Dig. Forens., Secur. Law 11, 21–34. https://doi.org/10.15394/jdfsl.2016.1376.