Hindawi
Security and Communication Networks
Volume 2017, Article ID 1306802, 17 pages
https://doi.org/10.1155/2017/1306802

Research Article

Similarity Digest Search: A Survey and Comparative Analysis of Strategies to Perform Known File Filtering Using Approximate Matching

Vitor Hugo Galhardo Moia and Marco Aurélio Amaral Henriques

Department of Computer Engineering and Industrial Automation (DCA), School of Electrical and Computer Engineering (FEEC), University of Campinas (UNICAMP), 13083-852 Campinas, SP, Brazil

Correspondence should be addressed to Vitor Hugo Galhardo Moia; vhgmoia@dca.fee.unicamp.br

Received 26 May 2017; Accepted 3 August 2017; Published 26 September 2017

Academic Editor: Bernardete Ribeiro

Digital forensics is a branch of Computer Science aiming at investigating and analyzing electronic devices in the search for crime evidence. There are several ways to perform this search. Known File Filter (KFF) is one of them, where a list of interest objects is used to reduce/separate data for analysis. Holding a database of hashes of such objects, the examiner performs lookups for matches against the target device. However, due to limitations over hash functions (inability to detect similar objects), new methods have been designed, called approximate matching. This sort of function has interesting characteristics for KFF investigations but suffers mainly from high costs when dealing with huge data sets, as the search is usually done by brute force. To mitigate this problem, strategies have been developed to better perform lookups. In this paper, we present the state of the art of similarity digest search strategies, along with a detailed comparison involving several aspects, as time complexity, memory requirement, and search precision. Our results show that none of the approaches address at least these main aspects. Finally, we discuss future directions and present requirements for a new strategy aiming to fulfill current limitations.

1. Introduction

The development and popularity of technology and Internet have been beneficial as well as problematic to modern society. The good side is that we have at our disposal cutting-edge technology for a decreasing price and high-speed Internet connection at will. Besides, people usually own multiple devices (desktop computers, smartphones, tablets, etc.) sharing all their information among them. The downside of this trend is the huge amount of data generated even by average users. Forensics examiners, whose duty is to investigate and analyze electronic devices in the search for crime evidence, are some facing this problematic side of technology development.

To deal with the overwhelming amount of data available nowadays [1], forensics examiners use methods to reduce the volume of data effectively processed. One way is by focusing on separating relevant from irrelevant information, using the Known File Filter (KFF) method. In such case, they eliminate *good* files from analysis (the ones belonging to operating system, known software, among other inoffensive ones: white list) and/or separate *bad* ones (illegal or suspicious objects: black list), using in both cases databases of known interest objects [2].

Suitable candidates to perform the aforementioned pre-processing are cryptographic hashes (MD5, SHA-1, SHA-2, etc.). NIST (National Institute of Standards and Technology) [3] provides a hash database of *good* files that can be used by examiners to do this filtering process. However, such method is vulnerable to even small changes in the input, since the hash of the original object will be completely different from the one of an object sharing the same content but differing in a single bit. Fixing this issue demands storing hashes for every new version of an object. As software, libraries and operating

systems files get updated constantly, keeping a hash database for every single change is infeasible and also very difficult to keep up to date.

Approximate matching functions appear as candidates to mitigate these and other issues. These new techniques aim at detecting similarity between objects by creating a digest in a way that similar objects will produce similar digests. This way, any change in the input will reflect in a minor change to its representation. When comparing two digests, a score related to the amount of content shared between them is given. Usually, if a minimal and predefined value is achieved, they can be considered similar.

The use of approximate matching tools in digital forensics is very beneficial, especially when used as a data reduction technique. Breitinger et al. [4] show that this sort of functions can increase the file identification rate significantly (from 1,82% using the traditional cryptographic hash to 23,76% with approximate matching). However, this benefit comes at a price: The high costs compared to traditional hash functions. Most techniques available nowadays are far more expensive than SHA-1, for example. Besides, they require a particular function to determine the similarity of two digests, which increases, even more, the whole process.

Although approximate matching functions are very costly compared to traditional hashes, the major bottleneck during forensic investigations based on KFF is not the hashing step (producing digests) but the search for similar pairs. Usually, the way an examiner performs the investigation is by brute force (all-against-all): Every digest in the database of known interest objects is compared to every digest created for the target device under analysis. This way, the complexity of this search is quadratic, and the whole process becomes very time-consuming. To cope with this excessive amount of time required, new strategies have been proposed in the literature using approximate matching efficiently and avoiding the brute force approach. These methods referred to as similarity digest search strategies aim at performing quick lookup procedures when dealing with large data sets by use of approximate matching digests. Here, the concept of the nearest neighbor problem [5] is extended to find similarity digests.

In this work, we present the state of the art of the strategies related to the similarity digest search process involving approximate matching functions. We classify them according to their main technology and perform a comparison, analyzing their characteristics to point out their strengths and weakness. Among all aspects discussed, the main ones are related to the time complexity of a lookup procedure, memory requirement, and search precision. In the end, future directions are provided along with a set of requirements for a new strategy aiming to fulfill current limitations. To the best of our knowledge, this is the most complete work aiming at presenting and comparing similarity digest search strategies.

We emphasize that this paper is an extension of the one to be published in SBrT 2017 [6], encompassing a more complete and detailed analysis of a larger set of strategies.

The rest of the paper is organized as follows: Section 2 introduces the main concepts of approximate matching and the tools used by the similarity digest search strategies evaluated in this work. Section 3 details how the strategies operate, followed by Section 4 with the corresponding evaluation. A discussion about the findings of this paper is presented in Section 5. Section 6 concludes the paper and Section 7 gives future directions.

2. Approximate Matching

Approximate matching functions are defined by NIST [7] as a *"promising technology designed to identify similarities between two digital artifacts. It is used to find objects that resemble each other or find objects that are contained in another object."* Broder [8] defines resemblance when two objects resemble each other, while containment is when one object is contained inside another.

NIST classifies such functions according to their operational level, as follows:

(i) Bytewise: matching relies on the byte sequence of the object. Since it does not try to interpret data or consider any structure within it, this level is more efficient than others and format independent. Bytewise functions are also known as similarity hashing or fuzzy hashing.

(ii) Syntactic: this relies on the internal structure of the object. For this reason, it is format dependent but does not interpret the content of the object to produce results. For example, the structure of a TCP network packet could be used to match packets from the same source and or destination.

(iii) Semantic: this relies on the contextual attributes of the object. It is also known as perceptual hashing or robust hashing, and it is closer to human perception. In this operational level, the object is interpreted and hence format dependent. Therefore, it is more expensive. The similarity of the content of a JPG and PNG images is an example, in which their byte structures are different due to encoding, but the pictures are the same.

In this work, we focus on the bytewise level because of its interesting characteristics: format independency and efficiency. In a triage process, forensics examiners must produce results as quickly as possible dealing with a huge amount of data. This way, approximate matching appears as a suitable candidate for a first step into separating devices that potentially have evidence from those that do not. More details about the syntactic and semantic levels can be found in Dorneles et al.'s [9] work.

2.1. Cryptographic Hashes × Approximate Matching. In comparison to traditional hash functions (e.g., MD5, SHA-1, and SHA-2) where every bit changed in the input is expected to cause a dramatic change in the digest and only binary answers are provided (two objects are equal or not), approximate matching functions provide a confidence measure about the similarity shared between two objects. Some methods provide an answer in a fixed interval: 0-100, 0-128, and so on; others provide a value indicating dissimilarity as it increases

from the perfect similarity value (where two objects are identical or very similar).

We can also compare traditional hashes to approximate matching functions regarding the digest length. The output size of traditional hashes is fixed independently of the input, while approximate matching produces either a fixed or variable size output (proportional to the input), depending on the method. They also are more expensive than traditional hashes in both processes: digest generation and comparison. Approximate matching methods need a special function designed to compare digests, requiring a more complex computation and processing due to their singularities and largest digests.

2.2. Applications. The range of applications for this kind of functions is vast. One can use them in identifying new versions of documents and software, embedded objects (e.g., jpg file inside a word document), objects in network packets (without reconstructing the packet flow), locating variants of malware families, clustering, code reuse (intellectual property protection and/or bug detection), detection of deleted objects (fragments remaining on disk), deduplication on storage systems (e.g., cloud computing: save storage and bandwidth), and cross-device deduplication, among others [10–12]. Also, Bjelland et al. [13] present other common scenarios in which approximate matching can be used, showing practical experiments in which forensics can benefit from this technology. In one experiment, they look for emails using a small set given as leads to figure out other similar ones. The search uses an email database, and by the results unexpected information of alternative conversations was revealed. The other scenario presented by Bjelland determined successfully that cryptographic software was downloaded in a machine by just analyzing recorded network traffic.

2.3. Approximate Matching and Locality-Sensitive Hashing. It is important to pay attention to an usual misleading of concepts between two different functions: approximate matching functions and locality-sensitive hashing (LSH). The idea behind the LSH is to map similar objects into the same bucket with high probabilities. This method is a general mechanism that can be used for the nearest neighbor search (NNS) and data clustering. A broader view on LSH techniques can be found in [5]. Approximate matching functions, on the other hand, are designed to produce a digest from the object and by the comparison of two object digests establish a confidence measure about their similarity. In this work, we focus on strategies to perform KFF using approximate matching functions, where two data sets of digests must be compared efficiently.

2.4. Techniques. Next, we will present briefly the approximate matching functions used by the strategies discussed in this work.

2.4.1. Block-Based Hashing. Most basic approximate matching function, where data is broken into fixed-size blocks and hashes, is computed for each one of them. The final signature is the concatenation of all hashes. The similarity between two objects is measured by counting the number of common blocks. The dcfldd tool [14], an extension of the disk dump tool dd, implements this scheme.

Although the block-based hashing scheme is computationally efficient and straightforward to implement, it suffers from alignment issues. The insertion/deletion of a single bit at the beginning of the input will affect the content of all remaining blocks, and their hashes will be completely different. Besides, it cannot detect containment.

2.4.2. ssdeep. Content Triggered Piecewise Hashing (CTPH) is another method aiming to detect content similarity in the bytewise level, proposed by Tridgell. Adapted from Tridgell's spam email detector (spamsum) algorithm, Kornblum developed ssdeep [15]. The main idea of this tool is to create variable size blocks using a rolling hash algorithm to determine when blocks start and stop (set boundaries). The rolling hashing produces a random value based on a window that moves through the input byte-by-byte. After the first value is generated, the next ones are created very quickly given the old hash, the removed part of the window, and the new added one. The algorithm adopted by ssdeep was inspired by the Adler-32 checksum [15].

When generating a ssdeep digest, a sliding window of fixed size (7 bytes) moves through the input, byte-by-byte, and whenever the rolling hash produces a specific output, based on the current bytes in the window, ssdeep identifies a trigger point, denoting the ending and beginning of a block. Then, all generated blocks are hashed using a cryptographic hash function (FNV) and the 6 least significant bits of each hash is encoded using a Base64 character. The final digest is the concatenation of all characters generated through the blocks. The detailed explanation of the whole process, trigger values, and required parameters can be found in [15].

To compare two digests, ssdeep uses the edit distance algorithm. This function counts the minimum number of operations required to transform one string (digest) into another, using weighted operations, like insertion, deletion, substitution (single character), and transpositions (two adjacent characters). The result is produced in a range of 0–100, where 0 means that the two objects are dissimilar and 100 means a perfect match.

This scheme is not as sensitive to alignment issues as the block-based hashing, and insertions/deletions are expected to modify only the corresponding part in the digest. Thus, the major part will still be the same allowing similarity detection. Although this is one of the most known schemes for approximate matching, it works only for relatively small objects of similar sizes. One way to increase its detection capabilities and allow comparisons of different objects size was using two digests per item instead of one. When creating the digest, ssdeep uses two different values as trigger point (derived from the object size, called block size b). The result is the creation of two digests, where the first corresponds to a trigger value b and the second $2b$. This way, it is only possible to compare objects if their block sizes are within a power of a two at most. Also, the first digest is always two times larger than the second, resulting in lengths up to 64 and 32 bytes, respectively.

ssdeep has been object of research in order to address its limitations, especially regarding performance [2, 16]. A security analysis was also done and concluded that ssdeep is vulnerable to an active attack [17].

2.4.3. sdhash. One of the most known methods of approximate matching is sdhash, proposed by Roussev [18] in 2010. This tool is based on the idea of identifying and picking from an object features that are least likely to occur in other ones by chance and use them to create the digest. Roussev [18] defines feature as a byte sequence (64 bytes) extracted from the input by a fixed-size sliding window that moves byte-by-byte. Shannon entropy is calculated for all extracted features and a set of the lowest ones in predefined intervals are selected [19]. These features are hashed using SHA-1 and the result split into five subhashes, used to insert the feature into a Bloom filter. Each filter has a limit on the number of elements (features) that can be inserted on it, and when it reaches its capacity, a new one is created. The final digest is a sequence of all Bloom filters, and its size corresponds to about 2.6% of the input.

In order to compare two digests (Bloom filter sequences), sdhash compares each filter from the first digest to every filter from the second digest, selecting the maximum score at each step. The final result is an average of all comparisons, ranging from 0 (nonmatch) to 100 (very similar objects, not necessarily identical).

Extensive research has been conducted over sdhash, exposing some vulnerabilities and presenting improvements [20–22]. Furthermore, Roussev [10] showed that sdhash outperforms ssdeep both in accuracy and in scalability. Besides detecting similarity, sdhash also detects containment.

2.4.4. MRSH and MRSH-V2. Multiresolution Similarity Hashing-v2 (MRSH-v2) [23] is an extension of MRSH [24] regarding efficiency and performance. It combines ssdeep and sdhash, by using a rolling hash over a 7-byte sliding window to identify triggers and define blocks of variable size. Then, a cryptographic hash function (FNV-1a) is used to hash each block and the result is split into five subhashes, where the $k \cdot \log_2(m)$ bits of each part are used to address a Bloom filter (k is the number of subhashes and m the Bloom filter size). The final digest is a sequence of Bloom filters, just like sdhash, as well as the comparison function. The digest length is about 0.5% of the input. Although this method is faster than sdhash, its precision and recall rates are worse.

2.4.5. TLSH. Proposed by Oliver et al. [25], TLSH is a locality-sensitive hashing (LSH) scheme used to find similarities among objects by the use of digests, like approximate matching functions. TLSH processes an input object by using a sliding window (5 bytes) moving byte-by-byte, where, in each step, six trigrams (combinations of the window characters) are selected to populate a 128-bit array of bucket counts using a mapping function (Pearson hash). Then, the quartile points of the array are calculated. A fixed length digest of 35 bytes is the output generated by TLSH, where the first 3 bytes composed the header and the remaining 32 bytes the body. The header is constructed based on the quartile points, the object size, and a checksum, where one byte is reserved for each of these parameters. The body is constructed in a way that each array position is compared to the quartile points, and the result is a bit pair defined according to the quartile value it ranges on.

Two functions are required to compare digests. The first one, distance header, takes as input the objects size and quartile ratios to produce an output. The other one, Distance Body, calculates the hamming distance between the digest bodies. The final result is a sum of both functions, scoring 0 (zero) for identical (or nearly identical) objects or higher, as dissimilarity increases.

Further research showed that TLSH is more robust to random changes and adversarial manipulations than ssdeep and sdhash [21]. However, this scheme focuses on resemblance detection and does not seem to work well for containment detection.

2.4.6. Nilsimsa. Damiani et al. [26] propose Nilsimsa as a method to detect spam messages. The technique consists in using a fixed-size sliding window (5 bytes) that goes byte-by-byte through the input and produces trigrams of possible combinations of the input characters. The trigrams are mapped to a 256-bit array, called accumulator, using a particular hash function. Every time a position in the accumulator is selected, its value, initially set to 0, is incremented. After processing the entire input, all accumulator positions whose value is above a threshold are set to 1 in a new array; the remaining ones are set to 0. This new array is the final digest (32 bytes).

To compare two digests, Nilsimsa checks the number of identical bits in the same position. The result is adjusted, and the range varies from 0 (dissimilar objects) to 128 (identical or very similar objects).

3. Strategies for Similarity Digest Search

The major bottleneck in digital forensics investigations based on approximate matching and KFF is the similarity digest search. An examiner, who usually has a reference list containing objects of interest, needs to compare each object from this set to each one got from the target system under analysis. The goal is to find similar objects by their digests, using one of the approximate matching tools described in Section 2.4. As stated by Harichandran et al. [11], this challenge is related to the nearest neighbor problem, but here we need to identify similar objects by their digests only.

It is important to mention that this problem is different from finding exact matches, which can be solved efficiently with ordinary databases [27]. The similarity search involves finding similar objects sharing a certain degree of commonality (higher than a threshold) only by the comparison of their digests. The objects found this way are separated for a further and deeper analysis (black list) or eliminated from the investigation (white list).

Most approximate matching tools perform the similarity search by the naive brute force method, which means that

each object from the target system is compared to all objects from the reference list (all-against-all comparison). However, this process is too time-consuming for large data sets. Also, the comparison process of such tools requires a particular function, which is usually very expensive. The complexity of this search is $O(r \cdot n)$, where r is the number of digests in the reference list and n the number in the target system.

Traditional approaches aiming at finding exact matches usually create indexes for the digests (using traditional hash functions like SHA-1, SHA-2, SHA-3, etc.) of the reference list and store them in sorted lists, balanced trees, or hash tables. The complexity of a single query in such cases is $O(\log(r))$ for sorted lists and trees and $O(1)$ for hash tables, which are lower than $O(r)$ from the brute force similarity search.

To cope with this problem, researchers have proposed some techniques aiming to reduce the time involved in the similarity digest search. In the following subsections, we will present and compare these approaches, which seek to deal with the growing amount of data in forensic investigations by reducing somehow the required time for the search process. We present the strategies proposed so far (from the best of our knowledge) and classify them regarding the main technology used. We discuss their characteristics and limitations, as well as their working process, which encompasses two phases [28]:

(i) Preparation phase: the reference list objects (black list, e.g.) have their digest created using the chosen approximate matching tool, and then these digests are organized somehow to improve the lookup procedure.

(ii) Operational phase: the target system has digests created for its objects and, for each one, a comparison is done using the material compiled in the preparation phase.

Finally, we end this section presenting a comparison of the approaches and discussing our findings. For the rest of this section, we will refer to r and n as the number of digests in the reference list and the target system, respectively.

3.1. The Naive Brute Force Approach. The naive method for pursuing a similarity digest search process is by brute force (all-against-all comparison). Every object in the target system is compared to all objects in the reference list. This approach can be performed by all approximate matching tools presented in Section 2.4. In the preparation phase, the forensics examiner needs to create digests for the reference list objects using the chosen approximate matching tool and store them in a file, database, or any other structure. In the operational phase, the examiner needs to create a digest for each object of the target system and compare it to all reference list digests, using the comparison function of the approximate matching tool. The best match is the one sharing the higher similarity with the queried object, and if it is above a threshold, the corresponding object is separated.

The major drawback of this approach is the time complexity, which is $O(r \cdot n)$ or $O(r)$ for a single query. For larger data sets, the search can take days or weeks using common hardware [27].

3.2. Distributed P2P Search. This strategy aims at performing the search in a distributed way through a peer-to-peer fashion. Each node in the network has to manage part of the data of the reference list. Basically, under a request for a search of a given digest, it is calculated in which nodes similar items should be stored based on the distance to the nodes reference digests. The queried digest is then sent to the nodes sharing the higher similarity in order to be compared with the reference data stored on them. The nodes return whether there is a similar digest or not. This method assumes that similar digests will always be distributed to the same nodes.

Although this approach seems interesting due to the distributed processing, which could decrease the time taken in investigations, it presents some drawbacks, as extra storage requirement, a high number of machines to work with, and network delays. The first problem comes when one node leaves the network. Since each node manages one reference point, another one must come up and take this reference point to maintain data availability. However, this is not the ideal solution if it takes longer for a new node to enter the network. A solution would involve storing extra data (redundancy) on each node, in a way that even though some nodes are gone, data can still be recovered, which would increase even more the storage requirement of the system.

Another problem inherited from this approach is the high number of machines needed to maintain both availability and scalability. Communication delays could also degrade the quality of an investigation process. For this strategy, we will present two approaches: DHTnil and iCTPH. Both approaches do not address the problems aforementioned.

3.2.1. DHTnil: Distributed Hash Tables with Nilsimsa. DHTnil is an efficient lookup strategy for finding similar digests, based on the Nilsimsa approximate matching tool and DHT (Distributed Hash Tables). Its main goal is to identify spam emails. According to Zhang et al. [29], DHTnil stores digests in different nodes in a way that digests of similar emails are stored on one of a few nodes. It divides the Nilsimsa digest space into some subspaces (with no overlap) managed by the DHT nodes. Space is divided based on a point set, where every point is a core of a subspace (reference point) attributed to a node. Chord was chosen as DHT as well as the Voronoi diagram to divide the multidimensioned space, using Euclidean distance to verify similarity. As each subspace is managed by a DHT node, the similarity digest search involves only comparing the digest to the ones stored in a few chosen DHT nodes.

In the preparation phase, DHTnil requires that forensics examiners create digests for the reference list objects and generate the reference points. These points are a few digests selected from the reference list to represent the subspaces. The selection could be randomly or carefully chosen. Next, the remaining digests are stored on the corresponding DHT nodes where the distance from the queried digest and the reference point is smaller.

In the operation phase, a digest is created for the queried object, and the subspace it belongs to is evaluated. The DHT node selected and its neighbors are searched for similarity,

where the digest is compared to all other ones stored in the nodes. The number of matches is returned [29].

The main drawbacks of this approach are the already mentioned ones inherited by distributed P2P systems and also due to the approximate matching tool chosen (Nilsimsa), which suffers from significantly high false positive rates [25]. This approach also performs unnecessary lookups per digest and hence demands a high time complexity ($O(r)$), equal to the brute force method, although in practice the time is expected to be smaller. This high complexity is due to the number of items in each node being proportional to the number in the set (considering a uniform distribution of the items).

3.2.2. iCTPH: Distributed Hash Tables with ssdeep. A similar approach to DHTnil is iCTPH, which also uses Chord to store and lookup digests, but instead of Nilsimsa it uses ssdeep tool. The iDistance technique is used to map similar digests into near clusters, stored in different nodes. It divides the vector space into clusters, identified by reference points, and the digests are mapped into a cluster according to their distance from the reference point, using the edit distance algorithm (minimum number of operations required to transform one string into the other) [30].

Jianzhong et al. [30] explain that, in the preparation phase, one must compute the digest of all objects in the reference list, choose a set of points as reference ones, and then decide which cluster each digest belongs to. In the operational phase, each queried object has its digest calculated. iCTPH generates a query interval for each cluster related to the queried item and performs a comparison with all digests of the clusters corresponding to that interval. The number of similar digests found is returned.

iCTPH has the same drawbacks of DHTnil: poor approximate matching tool (ssdeep) [10] and some unnecessary lookups per digest, resulting in a time complexity equal to brute force: $O(r)$ (single query). Just like DHTnil, this approach also inherits the limitations of a P2P system.

3.3. Indexing Strategy. Winter et al. [27] present a different approach for similarity digest search, called Fast Forensic Similarity Search (F2S2). The authors use an indexing strategy based on ssdeep digests (but not restricted to it) to avoid the overwhelming amount of time required by the naive brute force method. It builds an index structure over the n-grams (n consecutive bytes) contained in a digest. All digests with the same n-gram queried are returned in a lookup procedure. They are suitable candidates for being similar to the queried item, and the comparison is restricted to these candidates only. Here we are interested in finding exact matches on the level of n-grams. The results presented in the paper point out an impressive speedup compared to brute force method.

The index structure chosen was a particular kind of hash table, containing two parts: a central array (referred to as index table) and variable size buckets, which can store multiple entries per bucket (n-grams).

The n-grams of a digest b with l bytes are $b_1 \cdots b_n$, $b_2 \cdots b_{n+1}, \ldots, b_{l-n+1} \cdots b_l$. They serve as lookup key and provide a link to all digests containing the same values. Two parts compose n-grams. The first one is used as the entry in an address function, responsible for mapping keys (n-grams) to positions in the index table, using the k leading bits of the n-gram (since a digest of CTPH is Base64 encoded, it is necessary to decode it before selecting the bits). The other part, called e-key, is used to identify the n-gram and it is part of the bucket entry, as well as the ID, a link to the corresponding digest [27].

In the preparation phase of F2S2, digests are created for all reference list objects using ssdeep and an ID is assigned to each one. Then, an index table is created, and the digests are inserted on it. However, they are not added directly. A sliding window goes byte-by-byte mapping each n-gram and digest ID to a position in the index table, inserting it in a new bucket or adding it to an existing one (as long as they share the same n-gram but have different ID), according to Figure 1.

In the operational phase, the first step is loading the index structure into main memory. Then, digests are created for each item in the target system, and their n-grams are extracted. A lookup procedure using these n-grams selects all digests (candidates) in the index containing the same n-gram queried. Finally, the ssdeep comparison function is executed for the queried digest and each candidate to confirm the similarity. We highlight that since ssdeep digests have two signatures for each object (one using block size b and another $2b$), the lookup procedure is done for both.

The main drawback of this approach is the chosen approximate matching tool since ssdeep is less accurate than others especially when comparing objects of different sizes [10]. The proposed strategy does not work with more precise tools, such as sdhash, since it does not support digests represented by Bloom filters, which cannot be ordered/indexed. Also, since the number of candidates sharing the same n-grams as query digest is proportional to the number of entries in the index, the time complexity of F2S2 is the same as brute force, $O(r)$, even though experiments have shown a speedup factor above 2000 compared to brute force approach [27].

3.4. Bloom Filter-Based Search

3.4.1. The MRSH-NET Strategy. Breitinger et al. [31] present a new similarity digest search strategy designed to work with Bloom filters, reducing the lookup complexity of a single query from $O(r)$ to $O(1)$. This approach, called MRSH-NET, is intended to work with sdhash and mrsh-v2 approximate matching tools and uses a single, huge Bloom filter to represent all the objects of a reference list. However, due to the characteristics of Bloom filters, the method is restricted to membership queries only: Does this set contain any similar object to the queried one? If so, an affirmative answer is returned, but it does not point out similar object(s).

MRSH-NET uses sdhash/mrsh-v2 to first extract features from the reference list objects and later insert them into a single Bloom filter instead of having one or multiple filters per object. This procedure aims to avoid the expensive brute force approach and hence speed up the similarity digest search process. To decide whether or not an object is inserted in the filter, the match decision is based on a sufficiently large number of succeeding features found in the filter. If this

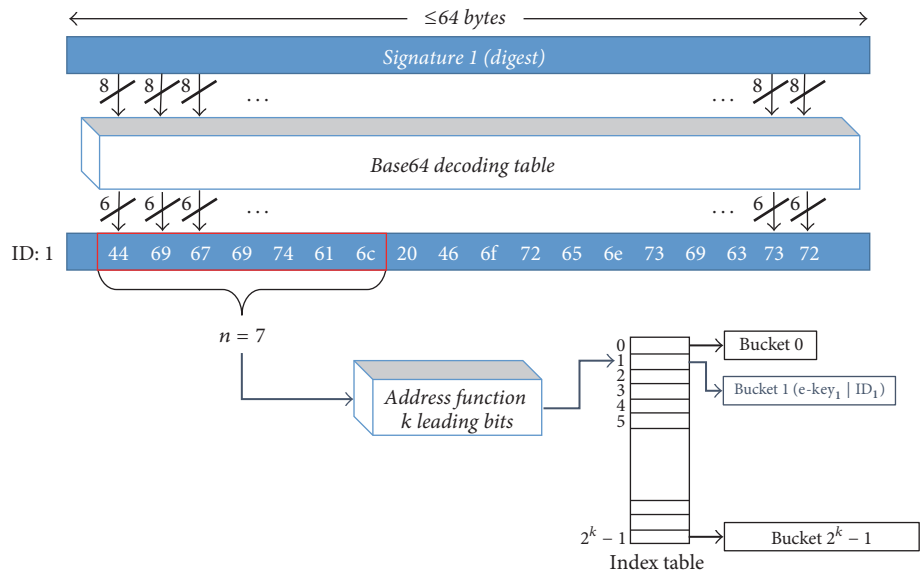

FIGURE 1: Inserting n-gram in the index table.

number is achieved, the object is considered part of the set, which could be a black list, for example.

In the preparation phase, one has to extract the features from all objects in the reference list, create a single, huge Bloom filter, and then insert them into the filter.

The operational phase involves loading the structure into main memory, extracting the features of the queried object, and checking in the Bloom filter for its presence or not. If the object has more than a threshold of succeeding features found in the filter, the object is said to be part of the filter and can be separated; otherwise, it is discarded, and the lookup procedure moves to another object.

One of the main drawbacks of this approach is the fact that it only decides about membership but does not identify an object. Also, the filter has to fit into the main memory due to efficiency reasons, another possible problem when the reference list set increases, as shown in the next section.

3.4.2. Bloom Filter-Based Tree Structure. As a form to mitigate the limitation of MRSH-NET [31], which only answers whether an object is present in the filter or not but does not identify the object, Breitinger et al. [32] propose a new similarity digest search strategy based on the well-known divide and conquer paradigm. In this approach, the authors build a Bloom filter-based tree data structure to store digests and efficiently locate similar objects. The time complexity of a single lookup is $O(\log_x(r))$, where x is the degree of the tree. Even though the complexity is higher than the previous method (MRSH-NET, $(O(1))$), this approach can return the actual matching object(s). However, this strategy lacks a working prototype and only exists in theory.

The basic idea of the Bloom filter-based tree approach is to recursively divide a given set S of similarity digests into X subsets. First, each object has its features extracted (e.g., by the sdhash method) and inserted into the root node of the tree, a huge Bloom filter. Then, S is divided into X subsets containing n/X elements; a child node of the root node is created for each subset and the objects inserted in each corresponding new filter. This procedure is applied recursively. Finally, an FI (File Identifier) is created in the leaf (a link to a database containing the digest of the related Bloom filter) as well as an FIC (File Identifier Counter), initially set to zero and incremented in a lookup procedure when FI is reached. An example of the construction of a Bloom filter-based tree is illustrated in Figure 2.

One of the main advantages of the scheme is the lookup operation. It is not necessary to compare a digest of a target system against all reference list digests but only to a subset of nodes in the tree structure. Also, as most comparisons will yield a nonmatch for blacklisting cases, the search ends in the root node. The tree is only traced down to a leaf if a match is found in the root, which means that the queried object (feature) is present in the reference list, and now we only have to determine which object it belongs to, by tracing down the tree and locating the corresponding FI. The match decision on whether an object is inserted in the tree data structure or not is based on a threshold, representing the number of following features required to be found in the tree. Every time we identify a leaf containing the features queried, we increase FIC. In the end, the highest FIC is compared to the threshold, and if its value is equal or higher, we can say that the object is present in the set and take the corresponding FI to reach it. Once we have found the candidate similar to the queried object, we might perform the conventional comparison using the approximate matching tool chosen.

The preparation phase of this approach consists in extracting the features of all objects in the reference list, creating the Bloom filter-based tree structure, and then inserting all features on it, including the corresponding metadata (FI and FIC).

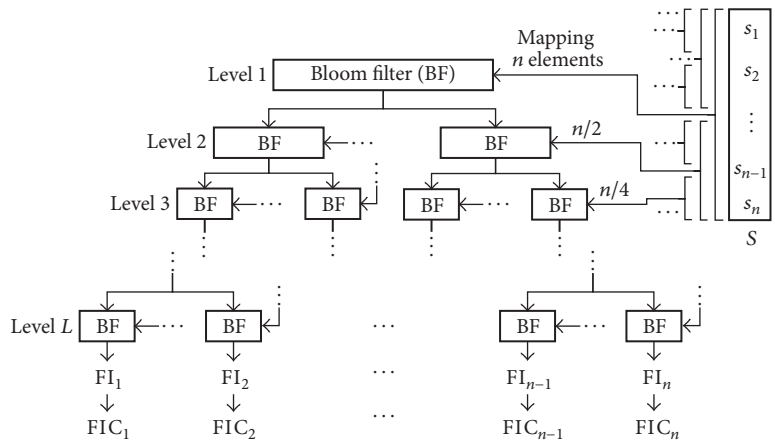

Figure 2: Bloom filter-based tree construction (binary tree). Adapted from [32].

The first step of the operational phase is to load the Bloom filter tree into main memory. Then, for each queried object, we need to extract its features and check the filter for their presence. In the end, the object with the highest FIC is returned, and the similarity can be confirmed by using the conventional comparison function of the chosen approximate matching tool (sdhash/mrsh-v2).

One problem with this approach is the fact that there is no prototype to validate the model and compare it to other similarity digest search strategies. Moreover, the tree structure is very memory-consuming, since several large Bloom filters will be required to store all data from the reference list.

3.5. Cuckoo Filter: A New Alternative for Bloom Filters. Gupta and Breitinger [33] present a new strategy for the similarity digest search problem, called MRSH-CF. This approach is based on Cuckoo filters, a new method to overcome some limitations of Bloom filters. Their strategy is an alternative/extension for the MRSH-NET approach, where Cuckoo filters replace the Bloom filters. Although the authors presented their approach using the mrsh-v2 tool, since using with sdhash would be harder to integrate into the system, we will adapt this method to work with sdhash to compare it with the other strategies.

Cuckoo filter [34], a modification of Pagh and Rodler [35] Cuckoo hashing, is a minimized hash table for performing membership queries, by using the Cuckoo hashing to resolve collisions. Data is transformed in a fingerprint before being inserted into the structure, where it is stored in buckets. The filter has an array of buckets of size b, referring to the number of fingerprints that can be stored on it. Also, there is a load factor (α) which describes the percentage of the filter being used in order to decide if the filter needs to be resized.

Cuckoo filters are composed of a hash table and three hash functions. Each key (entry value) is hashed by two of the hash functions (H_1 and H_2), responsible for assigning the key to buckets. The first value is tried and checked in the bucket for an empty space. If there is nothing there, the value is placed in this bucket. Otherwise, the key is stored in the second bucket. However, if there is already a key stored on it, this value is placed in another bucket, and the process repeats for this registry until all keys are allocated. If a cycle happens, where the same bucket is visited twice, it means that the table is not big enough and needs to be resized, or the hash function needs to be replaced. The third hash function (H_3) is used to store the key in the structure in a compressed form, hashing it and using only f bits as the tag size [34].

According to Fan et al. [34], the main benefits achieved with this structure compared to Bloom filters are the support for adding and removing items dynamically, better lookup performance, and less space requirement for some applications (related to some false positive rates). Although insertion operations are more complex due to possible keys reallocations, its time complexity is the same one as that for Bloom filters ($O(1)$). The deletion and lookup complexities are also $O(1)$.

To allow similarity identification, MRSH-CF stores only object features in the filter instead of the whole object. Based on MRSH-NET, this new Cuckoo filter-based strategy considers an object match when a specific number of features are found in the structure.

The preparation phase of this strategy requires the extraction of the features from the reference list objects using the chosen approximate matching tool. Then, the Cuckoo filter structure is created, and all features are inserted on it.

In the operational phase, the first step is to load the structure into main memory. The queried objects have its features extracted and checked within the filter. If a number equal to or higher than a threshold of features is found, the queried object is said to be part of the set; otherwise, it is put away, and the lookup procedure continues with the next object.

The main limitations of this approach are the same ones of MRSH-NET: membership queries and high memory consumption. The Cuckoo filter strategy only gives binary answers: the object belongs to the set or not. It does not point out which is similar object, which could be enough for some problems, but for KFF it represents a limitation.

3.6. Other Strategies. There are other similarity digest search strategies not addressed in this paper for a particular reason or because we are not aware of them. A particular one is

Table 1: Similarity digest search strategies: characteristics.

Strategy	Tools	Main technology	Input	Output (t - threshold)	Match decision	Insert/ remove elements	Owning database
Brute force (sdhash)	sdhash	Bloom filters	sdhash digest	Digest $\geq t$	Bloom filter comparison	√/√	×
Brute force (ssdeep)	ssdeep	Rolling Hash	ssdeep digest	Digest $\geq t$	Edit distance	√/√	×
Brute force (TLSH)	TLSH	LSH	TLSH digest	Digest $\geq t$	Header/body distance	√/√	×
DHTnil	Nilsimsa	DHT (chord) + Voronoi diagram	Bit vector	Number of matches $\geq t$	Adapted Euclidean distance	√/√	×
iCTPH	ssdeep	DHT (chord) + iDistance	ssdeep digest	Number of matches $\geq t$	Edit distance	√/√	×
F2S2	ssdeep	Indexing (n-grams) + hash table	ssdeep digest	Candidates sharing the same n-gram queried	Edit distance	√(∗)/√	√
MRSH-NET	sdhash, mrsh-v2	Single, huge Bloom filter	Object features	Yes/No (consecutive features found in the filter $\geq t$)	Bloom filter matches	×/×	√
BF-based tree	sdhash, mrsh-v2	Bloom filter tree structure	Object features	Candidate with highest number of features found in the filter $\geq t$	Bloom filter matches	×/×	√
MRSH-CF	sdhash, mrsh-v2	Cuckoo filter	Object features	Yes/No (consecutive features found in the filter $\geq t$)	Cuckoo filter matches	×/√	√

Observation: (∗): the data set increase (beyond its real capacity) is allowed at the cost of performance.

proposed by Chawathe [28, 36] which is based on a locality-sensitive hashing (LSH) method. However, due to the lack of data presented in the paper, we choose not to include it in our analysis.

4. Comparison of Similarity Digest Search Strategies

In this section, we present a comparison of all strategies for similarity digest search discussed previously. We divide our analysis into two parts: characteristics and evaluation. We first present the strategies regarding some characteristics and then an assessment of time and space required by the approaches, along with other topics such as false positive rates and detection capability.

4.1. Characteristics. Our first analysis involves comparing the approaches regarding their main characteristics. Table 1 presents our results followed by a discussion of the evaluated aspects and their importance for the performance of the strategies.

4.1.1. Supported Tools and Technology. Some strategies can be used with any tool while others are restricted to a specific one. In the latter case, we may have to use a tool with low accuracy depending on the strategy. On one hand, this would decrease the time to perform a similarity digest search, but, on the other hand, it would increase the process of manual inspection of the results due to the high number of false positives. Also, some tools could miss relevant data in the search. Another point of consideration is that depending on the chosen technology to perform the search, we need more computational power than we have, as the cases of iCPTH and DHTnil. They require several machines working together to solve the problem efficiently. This equipment may not be available in a forensics agency, and therefore these methods would not be appropriate.

4.1.2. Strategy Input/Output. Each approach aiming at reducing the time for the similarity digest search requires a different input format. Some strategies require only the digest created by the supported similarity tool while others require a precomputed set of values generated by a subprocess of the tool. The naive brute force method requires as input only the digest created by the chosen tool, while other methods require the object features extracted by sdhash or mrsh-v2 in an intermediary phase of the digest generation (MRSH-NET, Bloom filter tree, and MRSH-CF). Some strategies give to the forensics examiner a list of possible candidates found above some threshold (e.g., brute force ones and F2S2) or the most similar digest (Bloom filter tree), where the examiner needs to inspect manually. Before that, the examiner can use the corresponding approximate matching tool to confirm the similarity and eliminate some of the candidates. Other tools (e.g., DHTnil and iCPTH) give just the number of matches found or just the presence or not of the queried item in the set (MRSH-NET and MRSH-CF), which could be sufficient

in a blacklisting case to separate the corresponding media for further and deeper analysis.

4.1.3. Strategy's Match Decision. The match decision method usually incorporates the characteristics of the approximate matching tools supported and take advantage of their structure. F2S2 creates n-grams of a ssdeep digest based on the assumption that the tool encodes each "feature" extracted from the object in one byte in the digest. This way, similar objects will have similar features, and then the strategy can catch them by comparing these small parts (n consecutive bytes). MRSH-NET, BF-based tree, and MRSH-CF take the features extracted from sdhash/mrsh-v2 and insert them into a single, huge Bloom filter, BF-based tree structure, or Cuckoo filter, respectively. The match decision in such cases is the number of following features found in the structures. F2S2 does not create false positives due to its match decision method since it does not decide whether two digests are similar or not [27]. However, MRSH-NET, BF-based tree, and MRSH-CF approaches do, according to the explanation presented below.

According to Breitinger et al. [32], the false positive probability for an object is calculated by $p_f = p^r$, where p is the false positive probability for a single feature and r the number of following features required to be found in the filter to be considered a match. While r can be adjusted according to the desired false positive rate, p is defined by $p \approx (1 - e^{-kz/m})^k$, where k is the number of independent hash functions, z the number of features inserted into the Bloom filter, and m the filter size [32]. This way, the strategy decides when a match is found by a number of features present in the filter, and based on it, it can create more or less false positives depending on the parameters used by the strategy.

On the other hand, we have strategies in which a match decision does not depend on the strategy itself, but on the tool under consideration. The chosen comparison method is the tool's comparison function, and it is responsible for establishing when a match is found or not. The most simple strategy, the brute force one, is an example of such case. When using sdhash, the method is the comparison of Bloom filters. Using ssdeep, it changes to the edit distance, while for TLSH it uses a distance header/body (Hamming distance approximation) function. Changing the tool does not interfere in the strategy itself, but only in the comparison function, which is tool-related.

4.1.4. Inserting/Removing Elements and Owning Database. With respect to the insertion of new elements, some strategies allow the increase of the database dynamically, while others need to be constructed considering the knowledge of the maximum number of objects stored on it. The brute force ones, DHTnil, iCPTH, and F2S2 are examples which allow the data set to increase dynamically without requiring the database recreation. F2S2, which uses a chaining hash table, is a particular case in which the insertion beyond its real capacity is allowed at the cost of performance degradation (linear as the table fills). Other strategies as MRSH-NET and BF-based rely on technologies (Bloom filters) which require the knowledge about the maximum number of items beforehand to adjust some parameters, like false positives rates, for instance. Although it is possible to insert as many items into Bloom filters as we want, its false positive rates will increase and degrade the search quality, compromising the results of the strategy. In such cases, a new data structure will have to be created and adjusted to the new number of items. MRSH-CF is another case which needs the maximum number of objects before the structure creation since the hash table and buckets have a fixed and predefined size. However, this structure is more robust than Bloom filters as it can store multiple elements in each bucket without altering the false positives rates significantly.

Removal operations are also possible for most strategies, except for the ones based on Bloom filters, in which, once we insert several elements, we cannot distinguish the positions set in the filter for an item from another. This way, removing elements is not possible.

Some strategies have their own technology to store the similarity data, as F2S2 (hash table), MRSH-NET (Bloom filter), BF-based tree (Bloom filter tree), and MRSH-CF (Cuckoo filter). Others, like the brute force ones, DHTnil, and iCPTH, require a database to store the digests. These strategies may use whatever storage technology the examiner want: ordinary databases, files, xml, and so on. Although the aforementioned methods scale better to set increasing, they may have their efficiency degraded depending on the chosen technology, causing some extra delays.

4.2. Evaluation. Our second analysis evaluates the approaches concerning time and space requirements, false positives rates, and detection capabilities. We present in this section a discussion of these aspects and summarize ours results in Table 2.

4.2.1. Memory Requirements. We evaluated all similarity digest search strategies related to the amount of memory required for different data set sizes, varying from 1 GiB to 1 TiB. Our results are shown in Table 2, describing the amount required (MiB or GiB) and compression rate for each strategy. The details of our calculation are presented in Appendices A and B.

We highlight that some strategies, as MRSH-NET and BF-based tree, have their structure size adjusted for practical issues, since they are based on Bloom filters whose size has to be a power of two (2^c, for $c \in \mathbb{N}$). This way, when calculating the filter size and getting a result of $2^{31.27}$ bits, for instance, we need to adjust their size for 2^{32}. Although this modification can almost double the theoretical size of the filter in some cases, it is necessary for practical implementations. Other strategies have a minor effect regarding this issue, as MRSH-CF, which also needs some adjustments in the tag size. After defining the size for the item representation in the filter according to the false positive rate and number of entries in each bucket, we may have a decimal number. In such case, we choose the next integer not to increase the false positive rates. The result is an increase in the structure size, but not as significant as it happens with the Bloom filter approaches.

Table 2: Similarity digest search strategies: performance assessment of different properties.

Strategy	Memory requirements				Single lookup complexity	False positives	Resemblance/containment detection
	1 GiB	10 GiB	100 GiB	1 TiB			
Brute force (sdhash)	25.60 MiB (2.50%)	256.00 MiB (2.50%)	2.50 GiB (2.50%)	25.60 GiB (2.50%)	$O(n)$	No	✓/✓
Brute force (ssdeep)	0.19 MiB (0.02%)	1.87 MiB (0.02%)	18.75 MiB (0.02%)	192.00 MiB (0.02%)	$O(n)$	No	✓/×
Brute force (TLSH)	0.07 MiB (0.01%)	0.68 MiB (0.01%)	6.84 MiB (0.01%)	70.00 MiB (0.01%)	$O(n)$	No	✓/×
DHTnil	32.49 MiB (3.17%)	33.05 MiB (0.32%)	38.68 MiB (0.04%)	96.43 MiB (0.01%)	$O(n)$	No	✓/×
iCTPH	96.62 MiB (9.44%)	98.30 MiB (0.96%)	115.18 MiB (0.11%)	288.43 MiB (0.03%)	$O(n)$	No	✓/×
F2S2	1.71 MiB (0.17%)	17.07 MiB (0.17%)	170.70 MiB (0.17%)	1.71 GiB (0.17%)	$O(n)$	No	✓/×
MRSH-NET	16.00 MiB (1.56%)	128.00 MiB (1.25%)	1.00 GiB (1.00%)	16.00 GiB (1.56%)	$O(1)$	Yes	✓/✓
BF-based tree	176.00 MiB (17.19%)	1.79 GiB (17.90%)	17.64 GiB (17.64%)	336.00 GiB (32.81%)	$O(\log(n))$	Yes	✓/✓
MRSH-CF	14.00 MiB (1.37%)	140.00 MiB (1.37%)	1.37 GiB (1.37%)	14.00 GiB (1.37%)	$O(1)$	Yes	✓/✓

The brute force, DHTnil, iCTPH, and F2S2 do not suffer from such issue.

The first point to mention about our analysis is the memory growth rate of some strategies, as shown in Figure 3. While some of them have a linear behavior (brute force ones, F2S2, MRSH-NET, BF-based tree, and MRSH-CF), others do not (DHTnil and iCTPH). The latter group presents some specific costs which do not scale linearly since they have minimum setting costs necessary for operation, as the need for storing the Chord finger table and reference points list in each node (which in our case are kept fixed for all data set sizes). These two values are counted in the final memory requirement and, as the digests of both approaches have a short length, this setting cost stands out for small data sets.

According to our results, we see a significant disparity from one approach to another, especially increasing the data set size. This fact is noticed comparing the brute force (TLSH) and BF-based tree approaches. For a 1 TiB data set, the difference is ≈4915 times. The main reason for this is related to the approximate matching tool being used, a fact that can be corroborated comparing all other approaches with the ones using sdhash. Comparing the brute force approaches using TLSH (35 bytes) and ssdeep (up to 96 bytes) with the one using sdhash (vary according to the object size: ≈2.6%), we can see another great disparity, sdhash being 374.50 and 136.53 times more expensive than TLSH and ssdeep, respectively. The same applies to F2S2 and the methods using Bloom/Cuckoo filters (MRSH-NET, BF-based tree, and MRSH-CF), where the former beats the others since it is based on ssdeep and the others on sdhash.

Due to efficiency reasons, the structures should fit into main memory, a major problem for some strategies as the reference list grows. By the results, we can see that BF-based tree approach stands out due to its high memory consumption in comparison to the others. It has a bad compression rate, consuming about 336 GiB of memory for 1 TiB data set size (corresponding to 32.81% of the whole set size). Given a blacklisting case and the increasing size of data nowadays, since images and videos are becoming larger due to high-quality standards, 1 TiB is a reasonable size to consider. Therefore, this approach becomes impractical for examiners to handle. On the other hand, for the same amount of data, F2S2 consumes only 1.71 GiB in its compressed form (0.17%), which is easier to deal with. Other strategies like brute force (TLSH), brute force (ssdeep), DHTnil, and iCTPH consume even less, about 70, 192.00, 96.43, and 288.43 MiB, respectively. The structure size must be taken into consideration when choosing a strategy. A good choice would be the one that fits at the hardware specifications of the processing machine since loading the entire structure into main memory is the desirable form to have a more efficient search.

4.2.2. Lookup Complexity. The lookup complexity gives us an idea on how the strategies would scale in response to the reference list data set increase, presented in the form of *Big-O* notation. Table 2 shows the asymptotic upper bound for performing a single lookup. Our results indicate MRSH-NET and MRSH-CF as the best options for performing efficient queries, with time complexity of $O(1)$, although they are limited to only membership queries. Besides, experiments corroborate this statement since they indicate that MRSH-NET (best case) is about 12 times faster than sdhash brute force [31]. The BF-based tree strategy comes up as our third option, with $O(\log(n))$ complexity. All other approaches presented complexity equal to the naive brute force method ($O(n)$).

Although the lookup complexity is an important and necessary measurement for evaluating similarity digest search strategies, in some cases having experiments on time spent in the process is a more accurate form of comparison.

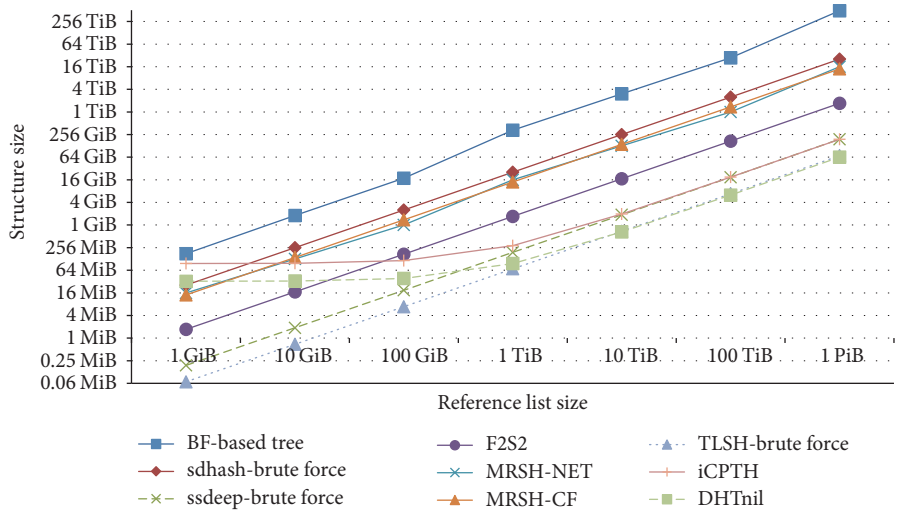

Figure 3: Memory requirements: growing behavior of the strategies according to the data set increasing.

Most strategies are much faster than brute force for normal operating conditions and yet have the same time complexity, as F2S2, for instance. Winter et al. [27] show that calculating the complexity of this approach requires two steps: finding candidates (digests sharing the same n-gram as the queried item) and similarity calculus. The first task can be accomplished with $O(\log(n))$ for fixed index table or $O(1)$ for dynamic resizing of the index table, while the second one presents complexity similar to brute force: $O(n)$. The reason for this high complexity is because the effort required is proportional to the size of the reference list. Then, when summing the complexity of the two steps, we get a complexity of $O(n) + O(\log(n)) \approx O(n)$ (single lookup) for both dynamic resizing and fixed index table. However, in practice, the benefits achieved by F2S2 will depend mainly on the efficiency and effectiveness of the candidates' selection. As we will not compare the queried item with all reference list digests but with a restricted set of those sharing the same n-grams with it, we expect a much faster process. According to Winter et al. [27] experiments, F2S2 achieved speedup above 2000 times faster than ssdeep brute force.

Other strategies may also be faster than the brute force approach, as DHTnil and iCPTH. In both methods, instead of computing r operations for a single lookup procedure just like the brute force approach, we compute only $p + l$ operations, where p denotes the number of reference points chosen and l the sum of the number of digests presented in each selected node. The sum $p + l$ is expected to be much smaller than r, resulting in a significant reduction in the search time in practice. However, the time complexity for this approach is the same as the brute force one ($O(r)$), since $p + l$ is proportional to the data set size and increases with it.

This way, it is important to analyze the strategies regarding their time complexity and their running time for a more accurate comparison, since some schemes may perform better in practice. We did not evaluate the running time for the strategies since some of them do not have a prototype, their source code is unavailable, and/or they require a complex environment for performing the experiments (e.g., DHTnil and iCPTH which are P2P-based approaches). However, we planned as future work to analyze the strategies and derive equations to estimate their time for both phases, the preparation and operational one. This will allow us to study how the approaches scale in practice and in which conditions one is better than the other.

4.2.3. False Positives. As discussed in Section 4.1.3, some approaches create false positives in the similarity digest search, as MRSH-NET, BF-based tree, and MRSH-CF. Others do not have the match decision incorporated in the strategy (brute force) and rely on the decision given by the tool.

DHTnil, iCTPH, and F2S2 are a middle term class since they have the match decision associated with the strategy. However, they do not create new false positives because the approach does not decide which objects are similar. The process is delegated to a function derived from the approximate matching tool and their goal is to separate a small set of possible candidates only. The tool's comparison function is later used to compare the set with the queried item to decide the similarity and hence reduce the number of objects to be manually inspected by forensics examiners.

4.2.4. Resemblance/Containment Detection. Detecting both resemblance and containment is a desirable property in any approximate matching tool since an examiner can find either of the objects that resemble each other as those contained in another. However, most approximate matching techniques are designed to detect only resemblance, the most basic operation mode. sdhash is the only one that can efficiently identify both modes. This statement is corroborated by Lee and Atkison [37] showing that sdhash performed the best regarding this aspect in comparison to most tools. Since the strategy is mostly tied to the tool it uses, it becomes limited to the sort of detection performed by it.

Most strategies analyzed in this work cannot detect containment. Only those based on sdhash have this ability (MRSH-NET, BF-based tree, and MRSH-CF). There is no current analysis about the effectiveness of the strategies and the amount of data shared between two objects in order to detect containment. However, as these strategies encode the features extracted by sdhash in the filter structures, any element sharing the same features will be a possible candidate for both detection modes. Since the strategies require a minimum number of following features found in the filter to consider it a match, any object fragment (small piece of data) already stored in the set is expected to have its features found in the filter and be considered a match.

4.2.5. Approximate Matching Tool's Precision. The search precision is more tied to the tool than the strategy itself. In some cases, the strategy only reduces the number of comparisons the examiner should do. While some approaches are based on tools like sdhash, which have interesting characteristics and can detect resemblance and containments for a variety of object sizes without compromising the results, others rely on limited tools. An example is the classic ssdeep, limited to only comparing objects of similar sizes and not suitable for dealing with large objects. This is corroborated by Roussev [10] and Breitinger et al. [38], showing that sdhash outperforms ssdeep in accuracy and scalability. Furthermore, Breitinger and Roussev [39] presented an evaluation of ssdeep, mrsh-v2, and sdhash using real data (extracted from the t5 corpus database [10]). They have analyzed the precision and recall rates of these tools and pointed out that sdhash had the best overall performance. The authors also stated that even though the precision rates of ssdeep and sdhash are high, the recall of all tools was relatively low.

With respect to Nilsimsa tool, Oliver et al. [25] state that even though this technique has powerful capabilities for resemblance detection, it suffers from significantly higher false positive rates than TLSH. Harichandran et al. [11] mention that TLSH is less powerful than sdhash for cross correlation.

Considering the tools' precision aspect, the strategies using sdhash are a better choice than the ones using ssdeep, Nilsimsa, or TLSH since the final result will be more accurate and scalable. Besides, it can detect both detection modes efficiently.

5. Discussion

All similarity digest search strategies presented in this work either show a high cost associated with memory requirements, have an approximate matching function not as suitable as the best ones available nowadays, or have high costs related to the lookup procedure. In the first case, we have the Bloom filter-based tree strategy which incorporates sdhash as similarity function, with good results from detection rates in both resemblance and containment. However, the tree structure is too memory-consuming and becomes infeasible to work with for large data sets. On the other hand, F2S2 presents better scalability regarding memory consumption, but it uses ssdeep as similarity function, which has several limitations that can compromise and/or restrict an analysis.

Other strategies that use a good similarity tool (sdhash), as MRSH-NET and MRSH-CF, for instance, with low lookup complexity and less memory consumption compared to the BF-based tree approach, are restricted to membership queries, which limits their application. The two P2P strategies (DHTnil and iCPTH) require the smallest memory in average, but they use weak approximate matching tools (Nilsimsa and ssdeep), require several machines working together, and may suffer from network communication delays. The fewer the machines are used, the more the P2P strategies become similar to brute force. The more the machines, the higher the costs and delays.

The brute force methods are very time-consuming, independent of the chosen similarity tool due to the high number of comparisons when comparing large data sets. Any other strategies can perform better than this naive approach. They are not suitable candidates for conducting investigations since the amount of data for each case has been increasing very fast [1]. Besides, they are strongly dependent on the tool in terms of precision, time, and memory requirement.

Our findings show that none of the similarity digest search strategies presented so far address at least the most desirable aspects: low memory requirement, high detection capabilities (for both resemblance and containment), and efficient lookup procedure. New strategies are required to address the overwhelming amount of data forensics examiners have to deal with. For this reason, in Section 7 we present some requirements for a new similarity digest search strategy, aiming to fulfill the most desirable points discussed so far and also other complementary characteristics.

6. Conclusion

This paper presented the state of the art of similarity digest search strategies aiming at reducing the time taken in forensics investigations. By using white/black lists, examiners can lessen the amount of time it takes in the search for evidence and/or reduce the volume of data analyzed. We detailed the working process of each strategy and performed a comparison of them regarding their characteristics to point out strengths and weakness. Among all aspects evaluated, we highlight the three most important ones: time complexity, memory requirement, and search precision. Our analysis showed that even though some strategies outperform others in some aspects, they fail in others. There is no currently suitable approach combining at least the most relevant requirements, exposing an opportunity on the field.

7. Future Directions

Future work on KFF techniques can be divided into two categories: approximate matching tools and similarity digest search strategies.

Since the search precision of the strategies is usually tied to the approximate matching tool under use, improvements on this topic are necessary. New research is needed to

compare the existing tools more precisely. sdhash, developed in 2010, is the most well-known tool and still the most accurate one, able to detect both resemblance and containment, a missing characteristic in newer tools. Improving sdhash regarding some of its limitations can be a target of future studies. Also, as new approaches have come up recently and have been compared mostly to sdhash to show their improvements, an evaluation of all available approximate matching tools according to their precision, time, and space is required to expose the big picture and show the deficiencies still existing in the field.

Concerning the similarity digest search strategies, future work involves analyzing the approaches to derive equations and estimate the amount of time taken in a practical scenario. An analysis of the effectiveness of current strategies (MRSH-NET, BF-based tree, and MRSH-CF) in detecting object containment is also required.

Given the limitations of current approaches, we present a set of requirements for a new similarity digest search strategy which seeks to address the main issues on investigations dealing with massive amounts of data. A desirable approach is the one which fulfills the following requirements:

(i) Having low memory consumption

(ii) Having efficient lookup procedure (low time complexity)

(iii) Supporting the most accurate approximate matching tools

(iv) Allowing both detection modes: resemblance and containment

(v) Returning the actual object(s) similar to the queried item (in contrast to membership queries)

(vi) Having no extra false positives in the process

(vii) Relying on its own storage structure

(viii) Inserting/removing elements dynamically

Ideally, this new strategy would fulfill all requirements. As shown by our analysis, none of the current approaches can address well all these aspects, since they mostly focus on improving only a particular aspect in detriment of the others. Finding a balance of these requirements would already be a significant improvement to the field, allowing more efficient investigations in an era of an overwhelming amount of data.

Appendix

A. Strategies

To estimate the amount of space required by each strategy for comparing them and analyzing how they scale with the data increasing, we developed and adapted some formulas. In the calculus, we first needed to estimate the number of files a data set with the chosen size would have. Then, we can calculate the number of digests that needed to be created of all objects, the number of entries in the hash tables, the size of the Bloom filters, and any other parameters required. To this end, we estimate an average object size and divide it by the data set size. The object size chosen was 512 KiB (this is an approximation of the size found in some known forensics data sets, as the *govdoc-corpus* (http://digitalcorpora.org/corpora/files (last accessed May 25, 2017).), for instance). For the rest of this section, we will consider n as the number of objects in the reference list.

A.1. Brute Force. The costs associated with memory requirement for the brute force approaches will be calculated for the two most used and known tools nowadays, which are ssdeep and sdhash. We will also consider the TLSH tool although none of the similarity strategies presented in the paper use it as approximate matching tool. We will show how TLSH would perform in an investigation as a brute force strategy due to its interesting characteristics (precision and recall rates and low digest length) and because it is a recently developed technique. Even though saHash [40] was newer, it was not chosen because it only works for objects of similar sizes and produces a minimum digests length of 769 bytes (≈22 times greater than TLSH).

(1) ssdeep and TLSH. The memory consumption of ssdeep and TLSH is calculated in the same way, by

$$m_{\mathrm{ss}} = n \cdot s_{\mathrm{ss}}, \tag{A.1}$$

where s_{ss} is the size of the digest created by ssdeep/TLSH.

(2) sdhash. To calculate the memory requirement for sdhash, first, we need to estimate the number of features present in an object (on average). According to Breitinger et al. [32], *"sdhash maps 160 features into a Bloom filter for every approximately 10 KiB of input file."* This way, we can calculate z (number of features) in the following way:

$$z = \frac{\left(\mu \cdot 2^{20} \cdot 160\right)}{\left(10 \cdot 2^{10}\right)} = 2^{14} \cdot \mu, \tag{A.2}$$

where μ is the reference list size (MiB) and 2^{20} and 2^{10} are factors to change, from MiB and KiB to bytes, respectively. Once we have calculated the number of features on the reference list, we can figure out how many Bloom filters will be needed to represent all these features and hence the memory requirement. To this end, we use

$$m_{\mathrm{sd}} = \frac{\left(z \cdot s_{\mathrm{bf}}\right)}{f_{\max}} \text{ (bits)}, \tag{A.3}$$

where s_{bf} is the size of each Bloom filter (bits) and $f_{\max}$ the maximum number of features allowed to be inserted by sdhash in each filter.

A.2. DHTnil. To calculate the memory requirement for DHTnil, we can use

$$m_{\mathrm{Dnil}} = \left(n \cdot s_{\mathrm{nil}}\right) + \left(n_{\mathrm{nodes}} \cdot \left(L_{\mathrm{ft}} + L_{\mathrm{rp}}\right)\right) \text{ (bits)}, \tag{A.4}$$

where s_{nil} is the size of Nilsimsa digests (256 bits), n_{nodes} is the number of nodes in the Chord network, L_{ft} the size of

the routing table of each node, and L_{rp} the size of the list of reference points stored in each node. We choose the number of entries in each finger table as $m = \log(N)$, where N is the number of nodes. The finger table size is

$$L_{ft} = (2 \cdot L_{id} \cdot m) + L_{idP} + L_{idS} \text{ (bits)}, \quad (A.5)$$

where L_{id} is the ID length of each one of the m entries (key and value), L_{idP} is the ID length of the predecessor node, and L_{idS} is the ID length of the successor node. These values are necessary information to the management of the Chord nodes. Also, L_{rp} refers to a list of digests which represent the reference nodes kept by each node to manage the search. Its length is

$$L_{rp} = n_{rf} \cdot s_{nil} \text{ (bits)}, \quad (A.6)$$

where n_{rf} is the reference points number chosen.

A.3. iCPTH. The same formula used by DHTnil is applied to iCPTH (see (A.4)). The difference here is the digest size, where ssdeep is used instead of Nilsimsa.

A.4. F2S2. Equation (A.7) estimates the amount of memory required for this strategy:

$$m_{F2S2} = n \cdot s_{ss} \cdot (1 + p_{fac}) + s_{name} \text{ (bytes)}, \quad (A.7)$$

where s_{ss} is the size of ssdeep digests, p_{fac} the payload factor added for the index (between 7 and 8), and s_{name} the length of each object name in the reference list.

A.5. MRSH-NET. To calculate the amount of memory required for the MRSH-NET approach, we can use the equation from Breitinger et al.'s work [31]:

$$m_{MRSH\text{-}NET} = \frac{k \cdot s \cdot 2^{14}}{\ln\left(1 - \sqrt[k \cdot r_{min}]{p_f}\right)} \text{ (bits)}, \quad (A.8)$$

where k denotes the number of subhashes, s the object set size in MiB, 2^{14} the number of features in the set s, r_{min} the number of following features required to produce a match, and p_f the probability of false positive for an object.

A.6. BF-Based Tree. Estimating the amount of memory required for this approach can be done using Breitinger's equations [31]. We first need to determine the size of the root Bloom filter, using

$$m_{BFroot} = \left\lceil -z \cdot \frac{\ln p}{\ln(2)^2} \right\rceil \text{ (bits)}, \quad (A.9)$$

where z is the number of features in the set (estimated by (A.2)) and p the false positive probability for a single feature, calculated by $p = \sqrt[r_{min}]{p_f}$. The parameter r_{min} is the number of consecutive features needed to be found in the filter and p_f the false positive probability for an object.

The next step involves calculating the level of the tree, using

$$h = \log_x(n), \quad (A.10)$$

where x is the degree of the tree (e.g., $x = 2$ for a binary tree).

Table 3: Similarity digest search strategies experiments: parameters.

Parameter	Value
s_{ss} (ssdeep)	96 (bytes)
s_{ss} (TLSH)	35 (bytes)
s_{nil}	256 (bits)
s_{bf}	2048 (bits)
f_{max}	160 (features)
k	5 (subhashes)
r_{min}	6 (features)
p_f	10^{-6}
p	0.1
x	2 (binary)
α	0.95
b	4 (items/bucket)
p_{fac}	8
s_{name}	10 (bytes)
$L_{id}/L_{idP}/L_{idS}$	160 (bits)
m	10 (nodes)

Then we calculate the memory required for the Bloom filter tree structure using

$$m_{BFtree} = m_{BFroot} \cdot h \text{ (bits)}, \quad (A.11)$$

where m_{BFroot} denotes the size of the Bloom filter root (see (A.9)) and h the level of the tree (see (A.10)).

A.7. MRSH-CF. To estimate the amount of memory required for MRSH-CF, we first need to compute the tag size for each item, which can be done using

$$f = \log_2\left(\frac{1}{p}\right) + \log_2(2 \cdot b) \text{ (bits)}. \quad (A.12)$$

Here, p is the false positive probability for a single feature and b the number of entries of each bucket in the hash table.

Then, we need to estimate the average of bits per item C. According to Fan et al. [34], each entry in the hash table stores one fingerprint, but not all of them are occupied. This way, there must be some slack in the table to the avoid failures when inserting new items, making each item cost more than a fingerprint. This value can be calculate by

$$C = \frac{f}{\alpha} \text{ (bits/item)}. \quad (A.13)$$

α, in this case, is the load factor ($0 \leq \alpha \leq 1$) used to express the percentage of the filter currently used.

Finally, we can estimate the amount of memory required by MRSH-CF using

$$m_{MRSH\text{-}CF} = z \cdot C \text{ (bits)}, \quad (A.14)$$

where z is the number of features extracted from the reference list (see (A.2)) and C the average bit per item (see (A.13)).

B. Parameters

For our experiments, we have adopted the parameters presented in Table 3.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is partially supported by CAPES FORTE Project (23038.007604/2014-69).

References

[1] D. Quick and K.-K. R. Choo, "Impacts of increasing volume of digital forensic data: a survey and future research challenges," *Digital Investigation*, vol. 11, no. 4, pp. 273–294, 2014.

[2] F. Breitinger and H. Baier, *Performance Issues about Context-Triggered Piecewise Hashing*, Springer, Berlin, Heidelberg, Germany, 2012.

[3] NIST, National software reference library, 2016, http://www.nsrl.nist.gov/.

[4] F. Breitinger, C. Winter, Y. Yannikos, T. Fink, and M. Seefried, *Using Approximate Matching to Reduce the Volume of Digital Data*, Springer, Berlin, Heidelberg, Germany, 2014.

[5] J. Wang, H. T. Shen, J. Song, and J. Ji, "Hashing for similarity search: a survey," 2014, https://arxiv.org/abs/1408.2927.

[6] V. Moia and M. A. A. Henriques, *A Comparative Analysis about Similarity Search Strategies for Digital Forensics Investigations*, XXXV Simpósio Brasileiro de Telecomunicações e Processamento de Sinais, Sao Pedro, Barzil, 2017.

[7] F. Breitinger, B. Guttman, M. McCarrin, V. Roussev, and D. White, "Approximate matching: definition and terminology," *National Institute of Standards and Technology*, vol. 800, article 168, 2014.

[8] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings of the 1997 International Conference on Compression and Complexity of Sequences*, pp. 21–29, June 1997.

[9] C. F. Dorneles, R. Gonçalves, and R. dos Santos Mello, "Approximate data instance matching: a survey," *Knowledge and Information Systems*, vol. 27, no. 1, pp. 1–21, 2011.

[10] V. Roussev, "An evaluation of forensic similarity hashes," *Digital Investigation*, vol. 8, pp. S34–S41, 2011.

[11] V. S. Harichandran, F. Breitinger, and I. Baggili, "Bytewise approximate matching: the good, the bad, and the unknown," *Journal of Digital Forensics, Security and Law*, vol. 11, no. 2, article 59, 2016.

[12] Y. Li, S. C. Sundaramurthy, A. G. Bardas, X. Ou, D. Caragea, X. Hu et al., "Experimental study of fuzzy hashing in malware clustering analysis," in *Proceedings of the 8th workshop on cyber security experimentation and test (CSET '15)*, vol. 5, p. 52, USENIX Association.

[13] P. C. Bjelland, K. Franke, and A. Årnes, "Practical use of approximate hash based matching in digital investigations," *Digital Investigation*, vol. 11, no. 1, pp. s18–s26, 2014.

[14] N. Harbour, "Dcfldd, Defense Computer Forensics Lab," *Net*, vol. 5, no. 5.2, article 1, p. 5, 2002.

[15] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, pp. 91–97, 2006.

[16] L. Chen and G. Wang, "An efficient piecewise hashing method for computer forensics," in *Proceedings of the 1st International Workshop on Knowledge Discovery and Data Mining, WKDD '08*, pp. 635–638, January 2008.

[17] H. Baier and F. Breitinger, "Security aspects of piecewise hashing in computer forensics," in *Proceedings of the 6th International Conference on IT Security Incident Management and IT Forensics, IMF 2011*, pp. 21–36, deu, May 2011.

[18] V. Roussev, "Data fingerprinting with similarity digests," *IFIP Advances in Information and Communication Technology*, vol. 337, pp. 207–226, 2010.

[19] V. Roussev, "Building a better similarity trap with statistically improbable features," in *Proceedings of the 42nd Annual Hawaii International Conference on System Sciences, HICSS*, usa, January 2009.

[20] D. Chang, S. Sanadhya, and M. Singh, "A collision attack on sdhash similarity hashing," in *Proceedings of the Proceedings of 10th International Conference on Systematic Approaches to Digital Forensic Engineering*, pp. 36–46, 2015.

[21] J. Oliver, S. Forman, and C. Cheng, "Using randomization to attack similarity digests," in *International Conference on Applications and Techniques in Information Security*, vol. 490 of *Communications in Computer and Information Science*, pp. 199–210, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[22] B. Frank, H. Baier, and B. Jesse, "Security and implementation analysis of the similarity digest sdhash," in *Proceedings of the 1st International Baltic Conference on Network Security and Forensics (NESEFO)*, 2012.

[23] F. Breitinger and H. Baier, *Similarity Preserving Hashing: Eligible Properties and a New Algorithm MRSH-v2*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Springer, Berlin, Heidelberg, Germany, 2013.

[24] V. Roussev, G. G. Richard III, and L. Marziale, "Multi-resolution similarity hashing," *Digital Investigation*, vol. 4, pp. 105–113, 2007.

[25] J. Oliver, C. Cheng, and Y. Chen, "TLSH—A Locality Sensitive Hash," in *Proceedings of the 2013 4th Cybercrime and Trustworthy Computing Workshop, CTC 2013*, pp. 7–13, November 2013.

[26] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati, "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, August 2004.

[27] C. Winter, M. Schneider, and Y. Yannikos, "F2S2: fast forensic similarity search through indexing piecewise hash signatures," *Digital Investigation*, vol. 10, no. 4, pp. 361–371, 2013.

[28] S. S. Chawathe, "Fast fingerprinting for file-system forensics," in *Proceedings of the 2012 12th IEEE International Conference on Technologies for Homeland Security, HST 2012*, pp. 591–596, November 2012.

[29] J. Zhang, H. Lu, X. Lan, and D. Dong, "DHTnil: an approach to publish and lookup nilsimsa digests in DHT," in *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications, HPCC '08*, pp. 213–218, September 2008.

[30] Z. Jianzhong, P. Kai, Y. Yuntao, and X. Jingdong, *iCTPH: An Approach to Publish and Lookup CTPH Digests in Chord*, Springer, Berlin, Heidelberg, Germany, 2010.

[31] F. Breitinger, H. Baier, and D. White, "On the database lookup problem of approximate matching," *Digital Investigation*, vol. 11, pp. S1–S9, 2014.

[32] F. Breitinger, C. Rathgeb, and H. Baier, "An efficient similarity digests database lookup-a logarithmic divide and conquer approach," *The Journal of Digital Forensics, Security and Law: JDFSL*, vol. 9, no. 2, article 155, 2014.

[33] V. Gupta and F. Breitinger, "How cuckoo filter can improve existing approximate matching techniques," in *Proceedings of the International Conference on Digital Forensics and Cyber Crime*, pp. 39–52, Springer, Berlin, Germany, 2015.

[34] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: practically better than bloom," in *Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies, CoNEXT 2014*, pp. 75–87, ACM, New York, NY, USA, December 2014.

[35] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms. Cognition, Informatics and Logic*, vol. 51, no. 2, pp. 122–144, 2004.

[36] S. S. Chawathe, "Effective whitelisting for filesystem forensics," in *Proceedings of the 2009 IEEE International Conference on Intelligence and Security Informatics, ISI 2009*, pp. 131–136, June 2009.

[37] A. Lee and T. Atkison, "A comparison of fuzzy hashes: evaluation, guidelines, and future suggestions," in *Proceedings of the SouthEast Conference*, pp. 18–25, ACM, Kennesaw, GA, USA, April 2017.

[38] F. Breitinger, G. Stivaktakis, and H. Baier, "FRASH: a framework to test algorithms of similarity hashing," *Digital Investigation*, vol. 10, pp. S50–S58, 2013.

[39] F. Breitinger and V. Roussev, "Automated evaluation of approximate matching algorithms on real data," *Digital Investigation*, vol. 11, no. 1, pp. s10–s17, 2014.

[40] F. Breitinger, G. Ziroff, S. Lange, and H. Baier, *Similarity Hashing Based on Levenshtein Distances*, Springer, Berlin, Heidelberg, Germany, 2014.