An operational costs analysis of similarity digest search strategies using approximate matching tools

Vitor Hugo Galhardo Moia, Marco Aurélio Amaral Henriques

[1]School of Electrical and Computer Engineering (FEEC)
University of Campinas (UNICAMP)
Campinas, SP, Brasil 13083-852

[vhgmoia,marco]@dca.fee.unicamp.br

Abstract. *Approximate matching functions are suitable tools for forensic investigators to detect similarity between two digital objects. With the rapid increase in data storage capacity, these functions appear as candidates to perform Known File Filtering (KFF) efficiently, separating relevant from irrelevant information. However, comparing sets of approximate matching digests can be overwhelming, since the usual approach is by brute force (all-against-all). In this paper, we evaluate some strategies to better perform KFF using approximate matching tools. A detailed analysis of their operational costs when performing over large data sets is done. Our results show significant improvements over brute force and how the strategies scale for different database sizes.*

1. Introduction

With the vast amount of data available nowadays, stored in a variety of different devices (computers, smartphones, tablets, etc.), forensic examiners face a new challenge: Process all these data in a relatively short time. Even ordinary investigations are becoming a problem since users tend to acquire multiple devices with high storage capabilities. The increased usage of cloud storage services is also an issue. Now, users have more space to store content and, in most cases, data is replicated among all devices (redundancy). Also, data is increasing in size due to high-resolution picture and videos, more sophisticated software and operating systems etc., requiring more time to be processed.

One of the methods used to handle this huge amount of data efficiently is the Known File Filter (KFF), a way to separate relevant from irrelevant information by the use of white or black lists of reference data. Forensic investigators can either eliminate data from the analysis, such as operating system files, known software, and other inoffensive objects (white list), or separate information that is considered illegal or suspicious (black list). In both cases, the examiner holds databases of known interest objects which are used in the filtering processes [Breitinger and Baier 2012].

Cryptographic hash functions (SHA-1, SHA-2, etc.) appear as suitable candidates to perform the KFF in a very efficient manner. To cooperate with forensic experts, NIST (National Institute of Standards and Technology) provided a white list database (NSRL - National Software Reference Library [NIST 2016]) in the form of hashes that can be used to perform this filtering process. However, some limitations arise with the use of such functions. By design, hashes of two objects with the same content will be the same; but changing a single bit in one of the objects, make its hash completely different. This way, only binary answers are provided with hashes: The objects are identical or not. This

 © 2017 Sociedade Brasileira de Computação

limitation force examiners to keep databases with hashes of every possible version of the interest object, since attackers could make random changes on objects to bypass detection. Also, some objects get new versions as software or operating systems are updated, and keeping track of each object changing is an impossible task.

To mitigate the limitations presented above, researchers have come up with new functions capable of detecting similarity between objects, called approximate matching or similarity hashes. They have the same idea of cryptographic hashes: The creation of small and compact representations for data (digest). Using a generation function, digests are created in a way that similar objects will produce similar representations, and by the utilization of a particular digest comparison function, a score related to the amount of the content shared between the corresponding objects is produced.

The downside of using approximate matching functions are the high costs associated with both processes: Digest generation and comparison. This fact becomes evident when comparing them to cryptographic hashes, although they have distinctive goals. However, hashes are the baseline, and new approximate matching functions always try to achieve similar time costs as traditional hashes.

Investigations involving the use approximate matching functions to perform KFF are very time-consuming. In the first stage, called preparation phase, digests are created and stored for every object in the reference list of interest objects. Then, in the operational phase, the target system under analysis has digests created for each object in this media and compared to the reference list. In case a certain pre-defined threshold is achieved, a match is found, and the corresponding object is separated for further analysis. However, the major bottleneck on this task is not the digest generation process, but the comparison one. Usually, the latter is done by brute force, where every digest in the target system is compared to every digest in the reference list. Since these two data sets tend to be very large, the whole comparison process could take days or even weeks to be accomplished.

In this work, we evaluate alternatives to the expensive brute force approach. These new methods referred as similarity digest search strategies, aim at performing digest lookups very efficiently for KFF. This paper extends the work done in [Moia and Henriques 2017], where the authors performed the analysis of three different strategies (F2S2, MRSH-NET, and BF-based Tree) regarding three main aspects: precision, memory requirement, and lookup complexity. We show how the same strategies carry out the search and develop equations to estimate their operational costs, allowing a more precise time comparison. Our results indicate that these strategies are a better choice than the simple brute force, and can reduce significantly the time consumed in KFF investigations. We also show how they scale when performing over different data set sizes. To the best of our knowledge, this is the first work to compare these strategies regarding operational costs.

2. Approximate matching tools and similarity digest search

According to NIST [Breitinger et al. 2014b], approximate matching functions are designed to identify the similarity between objects, in the level of resemblance (objects that resemble each other) or containment (objects contained in other, e.g., images inside a document). Also, these sort of functions can operate in three different levels when creating the digests: bytewise, syntactic or semantic [Breitinger et al. 2014b]. We restrict

 © 2017 Sociedade Brasileira de Computação

our research to only those functions working at the byte level since they are more time efficient and do not depend on the object format; moreover, they do not try to interpret the object either to consider any data structure.

Approximate matching tools usually have two primary functions: Digest generation and comparison. In the first one, a digest is created for an object as a small and compact representation of it. Its size can be fixed or not, depending on the chosen tool. For instance, ssdeep produces representations of at most 96 bytes (no metadata included), while sdhash digests are proportional to the object size ($\approx 2.6\%$). The second function of such tools is intended to compare two objects, where the output is a confidence measure about their similarity, usually in the scale of 0 (dissimilar) to 100 (very similar), but this changes according to the tool.

The applications of approximate matching are vast, encompassing the detection of new versions of documents, libraries or software, embedded objects, code reuse, malware clustering, among others. A broader view on approximate matching tools can be found in [Harichandran et al. 2016, Martínez et al. 2014].

The process of comparing two data sets of digests to find similar ones is referred to similarity digest search. The forensics examiner needs to check if any of the objects in the media under analysis is similar to any reference list object. Usually, this process is done efficiently using ordinary databases, but in such cases, the search aims at finding exact matches. Looking for objects that resemble each other or are contained one in another is a much harder task, and cannot be done using traditional ways but only by the simple brute force approach. The similarity detection is done by a particular function of the approximate matching tool chosen, which needs to be taken into consideration in the design of an efficient structure to store digests and allow similarity lookups. The strategies evaluated in this paper (F2S2, MRSH-NET, and BF-based Tree) follow this idea and are most specific for a particular kind of tool.

3. Strategies to perform known file filtering

The most time-consuming part of an investigation involving the KFF method and approximate matching functions is the comparison step, usually performed by the brute force approach. However, to avoid this expensive solution to perform comparisons, researchers came up with strategies to better find similar objects on large data sets. In this section, we present some of these strategies and show how they perform the similarity digest search.

3.1. Brute force

The simple method of brute force consists in comparing all digests from the reference list (usually pre-computed) with all digests generated for the target system. The best matching for each object is selected in case the value returned by the comparison function is above a certain threshold. The comparison using traditional hash functions (SHA-1, SHA-2, etc.) is fast since it only requires to verify whether two fixed size strings (digests) are equal or not. However, when using approximate matching tools, this operation becomes more expensive. Checking the similarity of two digests, in this case, requires interpreting the digests, using a special function designed for the chosen particular tool.

© 2017 Sociedade Brasileira de Computação

3.2. F2S2 strategy

The Fast Forensic Similarity Search (F2S2) strategy, proposed by Winter, C. et al. [Winter et al. 2013], is an alternative to brute force when performing the KFF method. This approach uses ssdeep as its approximate matching tool, but it is not limited to it. It requires that similar objects produce similar digests, a requirement fulfilled by ssdeep when looking at its comparison function: edit distance (number of operations to transform one string into another). Roughly speaking, ssdeep divides the object input into variable size pieces and codifies them into two digests of at most 96 base64 characters, in a way that similar objects will produce similar digests when compared [Kornblum 2006].

F2S2 is an index strategy which uses n-grams (n consecutive bytes) extracted from the digests and store them in a chained hash table. The n-grams are extracted using a fixed window size that moves through the digest byte-to-byte. They are used as lookup keys in the search for similarity, pointing out possible candidates for being similar to the queried object. At the end of the search, a comparison using ssdeep is made to only those objects sharing the same n-grams as the queried item. According to Winter's experiments [Winter et al. 2013], a speedup above 2000 times was achieved in comparison to the brute force approach using ssdeep.

The hash table used by F2S2 is composed of a central array with buckets of variable size, which means that more than one n-gram can be stored in each bucket (multiple entries), in a chaining fashion. Each n-gram is made of two parts: e-key and bucket address. The former identifies the sequence, while the latter is used to find the position in the hash table where it will be stored by a mapping function. Also, an ID is assigned to each digest meant to be inserted in the table and stored along with the n-gram in the bucket, as a way of linking it with the digest and allowing later identification [Winter et al. 2013].

The first step before using F2S2 is to build the index structure. By using ssdeep, the examiner creates digests for each reference list object and then inserts it in the index. The first step of this process is to assign IDs to each digest, followed by the n-grams extraction. Then, all n-grams are inserted in the hash table along with their corresponding ID. Here, collisions in the hash table buckets are resolved by using a chaining technique, where each bucket can store multiple n-grams, one followed by the other (linked list).

3.3. MRSH-NET: A Bloom filter approach

Another strategy for performing the KFF approach is the MRSH-NET, proposed by Breitinger, F. et al. [Breitinger et al. 2014a]. The motivation behind this strategy is to improve the lookup procedure of sdhash and other approximate matching tools which encode their digests using Bloom filters (BF). The F2S2 strategy presented previously cannot work with this sort of technology, since there is no effective way to index bloom filters.

The main idea of MRSH-NET is to create a single, huge bloom filter to represent all reference list objects instead of one or more BFs separately for each item. However, due to the characteristics of BFs, MRSH-NET is limited to only membership queries. It can answer if object *A* is contained in the set or not, but it can not point out what is/are the one(s) sharing similarity with it. According to Breitinger, F. et al. [Breitinger et al. 2014a], this could be enough for a blacklist case. However, we argue that it could generate lots of false positives that could not be confirmed since the strategy does not tell us which were the similar objects.

 © 2017 Sociedade Brasileira de Computação

To use MRSH-NET for KFF, the examiner needs to create the bloom filter structure and then insert all reference list objects on it. This process is done by first using the chosen approximate matching tool (sdhash in our work) to extract the features from each object in the list and inserting them into the single, huge Bloom filter.

3.4. BF-based tree

A possible solution to overcome MRSH-NET limitation regarding membership queries and yet to work with sdhash and related tools is presented by Breitinger, F. et al. [Breitinger et al. 2014c]. This new similarity digest search strategy is based on the divide and conquer paradigm, where a Bloom filter-based tree structure is used to store all reference list objects, allowing similarity detection and also object identification. However, this approach only exists in theory and lacks a working prototype. Also, it demands an enormous amount of memory to operate in comparison to other strategies [Moia and Henriques 2017].

Holding a reference list of objects of interest *S* with *i* items, the examiner must build the BF tree structure and fill it up with all the elements. For this purpose, a single bloom filter is created as the root node, and all items are inserted into the filter, by first using the chosen approximate matching tool to extract the object's features and then inserting them in the filter. The next step involves dividing the set *S* in half, where each part has its features extracted and added in new Bloom filters, child of the root node. These two parts are broken in half again, and four new filters are created with the features of the items belonging to each subset. Each filter is a child of the previous node. The process repeats until the division of the set results in one object. Finally, the parameters *FI* (File Identifier) and *FIC* (File Identifier Counter) are created at the end of the tree, as leaves. The former variable is used to link the corresponding BF to a database containing the digest of the item, while the latter is initially set to 0 (zero) and incremented when the corresponding *FI* is traced in a lookup procedure.

Lookups are expected to be very fast using this strategy since only a subset of nodes needs to be checked. The first step requires the feature extraction of the queried item, followed by a lookup procedure which is done for each extracted feature, starting from the root of the tree. In the case of a non-match in the root, we know for sure that this feature is not in the tree structure and we can proceed to the next object in the target system. Otherwise, the search continues through the rest of the tree until it reaches a leaf. The *FIC* of the corresponding leaf is incremented. After comparing all features from the queried item, the *FI* of the highest *FIC* is selected and compared to a threshold *t*, which is the minimum number of consecutive features that need to be found in the filter. When the value is higher than the threshold, the item is said to belong to the tree. Finally, a comparison of the selected item and the queried one is done in the digest level, using the chosen approximate matching tool with the purpose of verifying similarity. According to Breitinger, F. et al. [Breitinger et al. 2014c], most of the comparisons yield a non-match, especially for blacklist cases, leading the search to stop in the root node. They argue that the number of bad or illegal objects are usually much smaller than the total number of objects in the target system, making the BF-based tree strategy very time efficient.

© 2017 Sociedade Brasileira de Computação

4. Steps in a similarity digest search procedure

In this section, we evaluate the necessary steps in the operational phase of each strategy and present equations that we developed to estimate the time needed for each of them. We performed this analysis because most strategies do not have their source codes or a compiled version available to calculate their costs and compare them.

4.1. Brute force

The procedure involving the brute force can be summarized in the following steps:

1. For each item in the target system, perform:
   (a) digest generation;
   (b) For each item in the reference list, perform:
      i. digests comparison.
   (c) Return the digest with higher value if above a threshold *t*.

The steps aforementioned are part of the operation phase of a simple brute force approach. We can estimate the time to perform such task using Eq.1.

$$T_{op} = i \cdot (T_{digCalc} + (r \cdot T_{compFunc})) \quad (1)$$

Here i is the number of objects in the target system, $T_{digCalc}$ is the time for calculating a single digest using the chosen approximate matching tool (ssdeep, sdhash, etc.) or hash function (SHA-1, SHA-2, etc.), r the number of objects in the reference list and $T_{compFunc}$ is the time to compare two entries using the same tool.

Brute force can be used with any similarity tool. In our work, we chose ssdeep [Kornblum 2006] and sdhash [Roussev 2010] to perform the brute force approach and hence compare them with other strategies regarding time performance. We chose SHA-1 hash function as a benchmark since most approximate matching functions aim at achieving times close to it. We emphasize that SHA-1 vulnerabilities regarding collisions are not an issue here, as this hash function is not being used with a security role. Even simpler (and also compromised) functions as MD-5 would be useful here.

4.2. F2S2

Upon an investigation process where the forensic examiner wants to perform the KFF approach in a target system, the following steps must be done using F2S2:

1. Load the index structure into main memory;
2. For each item in the target system, perform:
   (a) ssdeep digest generation;
   (b) n-gram extraction;
   (c) n-gram lookup (*L* digests that share the same n-grams as the ones of the queried item will be selected as candidates).
   (d) ssdeep comparison of the queried item with the *L* selected digest.
   (e) return the digests with value above a threshold *t*.

 © 2017 Sociedade Brasileira de Computação

To estimate the time to perform the operational phase of F2S2, we can use Eq.2.

$$T_{op} = T_{indexLoad} + i \cdot (T_{genSsdeep} + T_{ngramExtr} + T_{ngramLookup} + (L \cdot T_{compSsdeep})), \quad (2)$$

where $T_{indexLoad}$ represents the time to load the index into memory (step 1), i the number of objects in the target system, $T_{genSsdeep}$ the time to compute a single ssdeep digest (step 2.a), and $T_{ngramExtr}$ the time to extract all n-grams from the corresponding object (step 2.b). $T_{ngramLookup}$ refers to the time to perform a lookup procedure in the index (step 2.c), calculated by Eq. 3. L represents the number of candidates sharing the same n-grams as the queried item, returned by the lookup process, and $T_{compSsdeep}$ denotes the time to calculate the Edit Distance for two digests (ssdeep comparison function)(step 2.d).

To calculate the time to perform the lookup procedure, we can use Eq. 3.

$$T_{ngramLookup} = g \cdot (T_{hashI} + (T_{compStrE} \cdot b)). \quad (3)$$

Here g is the number of n-grams extracted from the digest ($g = l_{dig} - n + 1$, where l_{dig} is the digest size (bytes) and n the n-gram sequence size), T_{hashI} the time to hash a string (n-gram index), $T_{compStrE}$ the time to compare two strings (n-gram e-key) and b the average number of different n-grams in each bucket (section 5.2 shows how to compute it).

4.3. MRSH-NET

The operational process using MRSH-NET involves the following steps:

1. Load the Bloom filter into main memory;
2. For each item in the target system, perform:
   (a) feature extraction (with sdhash), resulting in z features;
   (b) z lookups.

To estimate the time to perform this task, we first need to calculate the number of features (z) presented in the target system (average). To this end, we can use Eq. 4, derived from Breitinger, F. et al [Breitinger et al. 2014c] statement: *"sdhash maps 160 features into a Bloom filter for every approximately 10 KiB of input file"*.

$$z = (\mu \cdot 2^{20} \cdot 160)/(10 \cdot 2^{10}) = 2^{14} \cdot \mu, \quad (4)$$

where μ is the size of all objects in the reference list (MiB) and both factor, 2^{20} and 2^{10}, are used to change from MiB and KiB to bytes, respectively.

Then, we can use Eq. 5 to estimate the required time of MRSH-NET, as follows:

$$T_{op} = T_{bfLoad} + i \cdot (T_{featureExtr} + (z \cdot T_{bfLookup})). \quad (5)$$

where T_{bfLoad} is the time to load the bloom filter data structure into memory (step 1), i the number of objects in the target system, $T_{featureExtr}$ the time to extract the features from an object (step 2.a), and z the number of features extracted from it (Eq. 4). The $T_{bfLookup}$ is the time to perform a lookup procedure in the filter (step 2.b), calculated by eq. 6.

$$T_{bfLookup} = T_{hashF} + (k \cdot T_{compStrF}) \quad (6)$$

Here T_{hashF} denotes the time to hash each feature of β bytes and k the number of sub-hashes. MRSH-NET inserts a feature in the bloom filter by first hashing it and breaking the hash into k parts (sub-hashes). The resulting pieces are used to set the bloom filter. $T_{compStrF}$ is the time to compare two strings of F bytes each (feature size divided by k).

© 2017 Sociedade Brasileira de Computação

4.4. BF-based tree

We can sum up the required steps using BF-based tree by the following ones:

1. Load the Bloom filter-based tree structure into main memory;
2. For each item in the target system, perform:
   (a) feature extraction (with sdhash), resulting in z features;
   (b) z lookups in the tree.
   (c) If any object feature *FIC* is higher than the threshold *t*,
      i. compare the corresponding *FI* digest with the queried one (with sdhash).

To estimate the operational phase time of BF-based tree, we can use Eq.7.

$$T_{op} = T_{bfTreeLoad} + i \cdot (T_{featureExtr} + (z \cdot (T_{bfLookup} \cdot h)) + T_{compFunc}). \tag{7}$$

where $T_{bfTreeLoad}$ corresponds to the time to load the bloom filter-based tree structure into memory (step 1), i is the number of objects in the target system, $T_{featureExtr}$ the time to extract the features from a single object (2.a), z the number of features extracted from the object (calculated by Eq. 4), and $T_{bfLookup}$ the time to lookup each feature in the tree (Eq. 6) (step 2.b). The h parameter denotes the number of steps required to reach an object in the tree (see section 5.3), while $T_{compFunc}$ is the time to compare two digests using the chosen approximate matching tool (step 2.c).

5. Strategies operational costs

The similarity digest search strategies costs were estimated using the formulas developed in the previous section. We have designed pieces of code simulating the operations required by the strategies and then measured the necessary time for performing them. However, some parameters required a deeper analysis to understand their behavior and to estimate their values. To this end, we created a database of real data objects to perform the measures. In this section, we discuss details about this database, the singularities of some particular parameters, and finally, our results.

In this work, the analysis is focused on one of the most important parameters in forensics investigations: The database size. Other variations in the strategies parameters to find their best-operating conditions will be addressed in future works.

5.1. Test database details

To better understand the behavior of some operations and then estimate their values more precisely, we created a database from real data. This set encompasses over one million objects extracted from two Linux operating systems (Elementary OS client - Ubuntu 16.04-based - and Ubuntu 16.04 server), Microsoft Windows 10 Home, and also from personal data, which includes photos, documents, videos, applications, etc. Our database has 1,256,356 objects, corresponding to about 233.32 GiB of data. There were several typical applications installed in each operating systems, including Latex, LibreOffice, Microsoft Office 2013, Foxit Reader, NetBeans IDE, Internet Explorer, Google Chrome, Firefox, and default operating system applications. The type of the objects varies, including pdf, jpg, png, bmp, txt, doc, docx, odf, mkv, avi, py, mp3, wma, html, jar, rar, c, bin, among others. Since this work focus on the best performance of the approximate matching tools in their best conditions and our experience shows that ssdeep do not produce reliable results with large objects, we limited the object size in our database to 200 MiB. Future works will address scenarios using larger objects.

 © 2017 Sociedade Brasileira de Computação

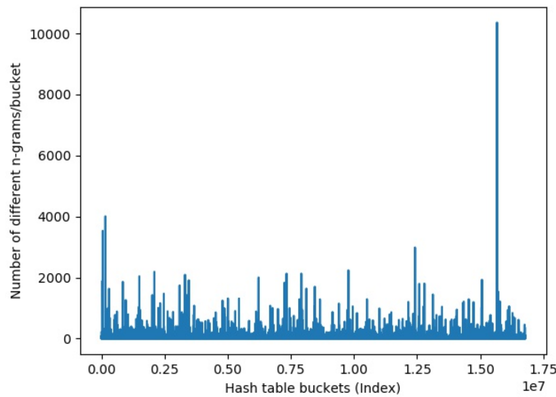

Figure 1. Different n-grams per bucket

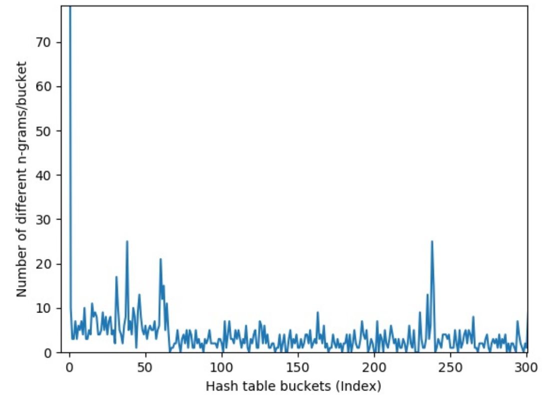

Figure 2. Number of different n-grams per bucket (amplified): index 0 to 300.

5.2. The F2S2 case analysis: Using different database sizes

To estimate some parameters of the F2S2 strategy, we implemented a simplified version of this approach, aiming to observe how data is spread over the hash table. Then, we estimated the values of variables b (number of different n-grams per bucket) and L (number of objects sharing the same n-gram as the queried item), necessary to the calculus of F2S2 operational phase. To this end, we generated ssdeep hashes for every object in our database (encompassing all four data sets) and inserted them in the index structure. The chosen hash table size was 2^{24} (4 6-bit base64 characters from the n-gram).

After inserting all n-grams obtained from our database objects in the index structure, a total of 75,549,716 n-grams were counted. However, 2,218,267 of 16,777,216 (13.22%) buckets were still empty, which means that lots of buckets contained multiple entries. Fig 1 shows the distribution of the number of different n-grams per bucket (b), while Fig. 2 is an amplified version of it, presenting only the b value for the 300 first buckets. Estimating b can be done using Eq. 8, as follows:

$$b = d_{ngrams}/n_{buckets}, \tag{8}$$

where d_{ngrams} is the number of different n-grams inserted in the hash table and $n_{buckets}$ the hash table size (number of buckets). The statistic information about the distribution of b in the table follows: standard deviation = 5.22, median = 2, and mode = 2. To find the average number of different n-grams in each bucket, we applied Eq. 8 for the data inserted in the F2S2 structure, obtaining a result of 2.27. Although we may find a few buckets with lots of n-grams, a significant portion of the table is empty and will lead to empty buckets, as we can see by Fig. 2. There is a peak in one bucket with a significant number of n-grams occurrences and others, but as expressed by the mode and median got from the analysis, most buckets have two or fewer elements. This fact led us to choose the average of the number of different n-grams as the b value (Eq. 8) since this seems the better choice when comparing it to the other statistic components got from the analysis.

Concerning to the number of objects sharing the same n-gram L, we have used

Eq. 9 to calculate this value.

$$L = n_{obj}/d_{ngrams}. \tag{9}$$

Here, the new variable n_{obj} denotes the number of objects in our database. We cannot use the average as we did in the previous case because digests may share multiple n-grams and we are only interested in finding the number of different digests sharing at least one n-gram with the queried item, not the number of similar n-grams. When we apply Eq. 9 to the data of the F2S2 hash table, we get a result of 0.033, which express our expectation for the average number of different objects sharing the same n-gram.

The values calculated so far are concerning to our data set only. Other sets of similar or different sizes may present different values for b and L since these parameters depend exclusively on the data being handled, which could have a different n-gram distribution. To extend our results to other sets, one needs to compute the b and L values for them. To this end, equations 8 and 9 can be applied. However, if by on hand we can define our hash table size ($n_{buckets}$) and count the number of objects in our database (n_{obj}) quickly, on the other hand, it is hard to know the number of different n-grams (d_{ngrams}) without inserting all ssdeep digests in the hash table and counting them.

To estimate the value of different n-grams (d_{ngrams}) for a data set with different type and size, required to the calculus of the parameters b and L, we need to find an expression that give us such value based on the total number of n-gram in this set. First, we consider the different data sources that forms the database (Linux client, Linux server, Windows 10, and personal data) to simulate different systems and get a more general idea on how the n-grams are spread across them. Then, we insert each set separately in the F2S2 index structure and count the number of different n-grams. We also use the data got from our first analysis where the entire database (all former sets together) was inserted in the structure since it can represent a different and larger set.

To find the expression that gives us the value of d_{ngrams}, we use regression analysis techniques. This way, we can determine the relationship between the values of different n-grams (unknown) and the total number of n-grams in the database (known). We chose the least squares method to this end. Applying this technique to our results (Tab. 1), we came up with the following expression:

$$d_{ngrams} = 2237231.04 + (0.4816 \cdot n_{ngrams}), \tag{10}$$

where n_{ngrams} is the number of n-grams of the database. We can obtain this number by multiplying the number of objects for the average number of n-grams in a single digest.

Table 1. Number of different n-grams in each data source

Source	Total number of n-grams	Number of different n-grams
Linux client	22,937,595	12,828,701
Linux server	28,071,532	18,405,835
Microsoft Windows	11,376,395	7,357,119
Personal data	13,164,194	7,316,768
All sources	75,549,716	38,053,476

The expression above allows us to estimate the number of different n-grams in any database size and hence calculate both b and L values.

 © 2017 Sociedade Brasileira de Computação

5.3. BF-based tree case analysis: Average steps in a search

Estimating the time needed for the BF-based tree strategy demands the knowledge of the number of times we will need to go through the tree structure to identify a given element. In a lookup procedure, there will be cases where the search stops in the root when the given element is not present in the structure, but in others, we may have to go through the entire structure to find it. Knowing the number of steps performed by element (h) will allow us to estimate the operational cost of the BF-based tree strategy (Eq. 7). To this end, one can develop an algorithm to count the average number of steps considering two scenarios: when the input is present in the tree and when it is not. In the first case, we go through the tree structure checking each node (bloom filter) for the presence of the given object until we find it, stopping the procedure in a leaf. For the second one, the search ends in the root tree, but there is also the possibility of occurring false positives, leading the process to go through the tree until all nodes of any level return a negative result. In the end, a weighted average between the number of steps found in each scenario and the number of elements must be returned, where the percentage of the two events occur are used as weight.

5.4. Parameters definitions and measurements

Using the proposed formulas we can estimate the costs for performing the searches with different database sizes. The values adopted in our experiments are presented in Table 2. We performed the tests using the following machine: Elementary OS 0.4.1 Loki 64-bit (built on Ubuntu 16.04.2 LTS), i7-5500U CPU @2.40GHz processor, 8 GB of memory, and NVIDIA GeForce 920M. We measured the time for each operation using the clock library from the C language, except for ssdeep and sdhash times (generation and comparison functions) which were measured using the time command (sys + user times) available on Linux distributions, since we used the compiled version of these tools. The times to compute the ssdeep, sdhash, and SHA-1 (both hashes generation and comparison) were calculated over the average object size, presented in the table. We repeated all experiments 20 times and took the average, taking care of clearing the cache each time to prevent previous results influencing new ones.

Before calculating the operational costs, we determined the strategies structure size. To this end, we used the formulas and parameters presented in another work [Moia and Henriques 2017], adapting only the database and average object sizes. We measured the time to load an object of 1 GiB from disk to memory (buffer of s_{buf} bytes) and adjusted this value according to the strategies structure size to simulate the loading.

5.5. Operational costs evaluation

We have measured the time for the three presented strategies (F2S2, MRSH-NET, and BF-based Tree) and also for the brute force one when using SHA-1, ssdeep and sdhash. In our experiments, we first considered the database presented in section 5.1 as our reference list and then measured the time for performing over it using different target system sizes, varying them from 1 GiB to 10 TiB. It is important to mention that our database encompasses objects from different fonts gathered in a single set. Using the equations from Section 3 along with the values of section 5.4, we can estimate the strategies operational costs, shown in Fig. 3 (a).

Table 2. Similarity digest search strategies experiments - Parameters

Parameter	Description	Value	STD
s_{obj}	Average object size	195 (KiB)	-
$s_{strategy}$	Strategies' structure size	see section 5.4	-
s_{buf}	Buffer size	4096 (bytes)	-
T_{read}	Time to read 1 GiB from disk to memory	429.7565 (ms)	8.1452 (ms)
T_{genSH}	SHA-1 digest generation	0.6581 (ms)	0.0949 (ms)
T_{compSH}	SHA-1 comparison function	0.0102 (ms)	0.0037 (ms)
s_{ngram}	n-gram size	7 (bytes)	-
s_{index}	index size	4 (bytes)	-
s_{ekey}	e-key size	3 (bytes)	-
T_{genSS}	ssdeep digest generation	5.6000 (ms)	2.3324 (ms)
T_{compSS}	ssdeep comparison function	1.0000 (ms)	1.7320 (ms)
$T_{indexLoad}$	Time to load the index structure	$s_{strategy} \cdot T_{read}$ (ms)	-
$T_{ngramExtr}$	Time to extract n-grams from ssdeep digest	0.0207 (ms)	0.0062 (ms)
b	N-grams in each bucket	see section 5.2	-
L	Candidates sharing the same n-gram	see section 5.2	-
T_{hashI}	Time to hash s_{index} bytes	0.0813 (ms)	0.0115
$T_{compStrE}$	Time to compare two strings (s_{ekey} bytes)	0.0010 (ms)	0.0001 (ms)
T_{genSD}	sdhash digest generation	19.2000 (ms)	5.3066 (ms)
T_{compSD}	sdhash comparison function	19.4000 (ms)	4.7791 (ms)
T_{bfLoad}	Time to load the Bloom filter structure	$s_{strategy} \cdot T_{read}$ (ms)	-
$T_{featureExtr}$	Time to extract features from an object	12.2975 (ms)	4.4667 (ms)
β	Feature size	64 (bytes)	-
T_{hashF}	Time to hash a feature of β bytes	0.0929 (ms)	0.0295 (ms)
$T_{compStrF}$	Time to compare two strings (β bytes)	0.0010 (ms)	0.0001 (ms)
k	Number of hash functions for the BF	5	-
$T_{bfTreeLoad}$	Time to load the BF tree structure	$s_{strategy} \cdot T_{read}$ (ms)	-
h	Average steps in a lookup procedure	see section 5.3	-
x	Degree of the tree	2	-

According to our results, the strategies presented a linear grow as the target system size increased. As expected, the brute force approach had the worst results, with sdhash as the most expensive one, followed by ssdeep and SHA-1. The latter one had similar results as the BF-based tree but was a bit more costly. The best strategy regarding the operational time was F2S2, presenting, on average, a speedup of 1,933,723.39, 99,676.83, and 996.82 times compared to the brute force approaches sdhash, ssdeep, and SHA-1, respectively, and 778.89 and 25.24 times better than BF-tree and MRSH-NET strategies, respectively.

Fig. 3 (b) shows how the strategies perform over a reference list size variation (10 GiB to 250 GiB) and using a fixed target system size database (1 TiB). We limited our experiments to this particular range since it is the one covered by our data set. Estimating the behavior of b and L parameters in the F2S2 equation for database sizes far beyond ours could lead to wrong results once the data distribution in such cases could be very different. For larger database sizes, additional studies are required to estimate these variables and hence the strategies operational costs.

 © 2017 Sociedade Brasileira de Computação

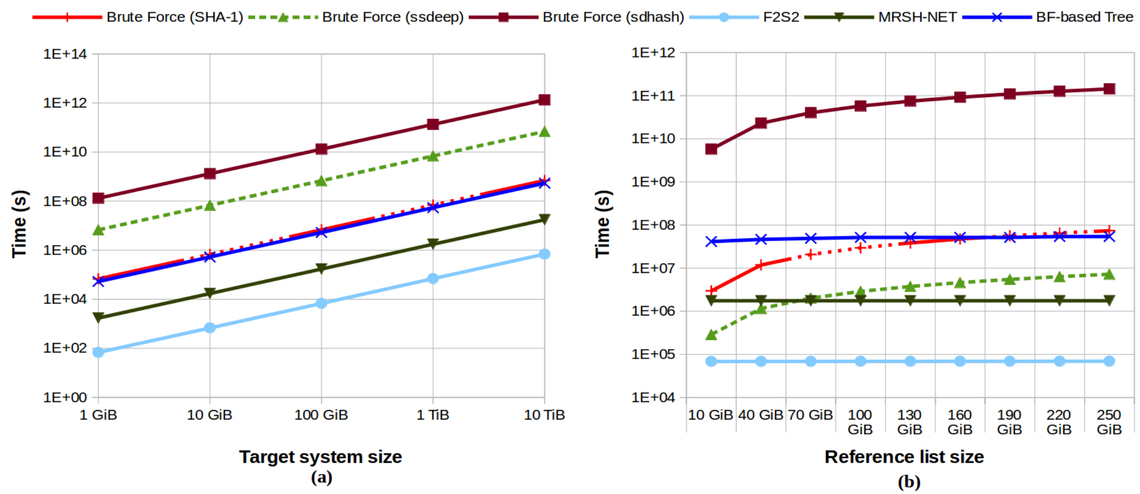

Figure 3. Strategies operational costs:
(a) Target system size variation, considering a fixed size ref. list (233.32 GiB)
(b) Ref. list size variation, considering a fixed size target system (1 TiB)

Again, our results pointed out that sdhash brute force is the most time-consuming approach and, on average, F2S2 is the best one. However, despite the linear behavior of the brute force method, the other strategies presented a different time variation as the reference list size increased. When performing queries over these new database sizes, MRSH-NET showed constant time as expected due to the lookup procedure time be independent of the structure size. The time on the BF-based tree strategy was not constant but increased slowly due to the tree structure getting larger as the reference list expanded (number of elements grow), elevating the average number of steps in a lookup procedure and hence the time to go through the tree. The F2S2 strategy time had a very small increase due to the number of similar n-grams increase. For databases ranging from 10 GiB to 250 GiB, the F2S2 time increase was only 1.24% of the initial value.

Our findings showed the prohibitive costs of the brute force method when performing over large database sizes. For instance, an investigation of a 1 TiB target system using a reference list of 233.32 GiB and sdhash brute force approach could take more than 4 thousand years to be accomplished (around 6,917,958,887,350 comparisons between the data sets are necessary). The same search using the strategies MRSH-NET and the BF-based tree could take only 70 and 2145 days, respectively. Using the ssdeep brute force, the same process would require around 754 years, while F2S2 would only demand no more than 20 hours. It is important to mention that this enormous amount of time is due to the hardware used to perform the experiments. In a working station with powerful hardware and using parallelism techniques, these times are expected to decrease, but the relative differences found among the strategies will remain.

Another important fact to mention is that even though F2S2 presents higher lookup complexity than MRSH-NET and BF-based tree [Moia and Henriques 2017], it performed better in our experiments when comparing large data sets. As expected, MRSH-NET was better than the BF-based tree.

6. Conclusion

Approximate matching functions appear as candidates to speedup the process of digital forensic investigations and yet being accurate in the search for objects of interest in medias with high capacity. However, traditional search procedures are based on the inefficient brute force method, which is too time-consuming and even prohibitive for larger data sets. In this paper, we evaluated similarity digest search strategies aiming at reducing this overwhelming cost. A detailed analysis of the operational costs of these approaches was presented, showing significant improvements over the simple brute force ones. Our results point out how the strategies scale with different database sizes and show the ones that performed best. Future studies encompass extending our analysis to other similarity digest search strategies, along with the estimation of the preparation phase costs and variations of the strategies parameters to find the best-operating conditions of each approach. We also plan to find new methods to estimate the parameters b and L of F2S2 to scale with any database size as well as developing a new strategy to overcome current limitations.

References

Breitinger, F. and Baier, H. (2012). *Performance Issues About Context-Triggered Piecewise Hashing*, pages 141–155. Springer Berlin Heidelberg, Berlin, Heidelberg.

Breitinger, F., Baier, H., and White, D. (2014a). On the database lookup problem of approximate matching. *Digital Investigation*, 11:S1–S9.

Breitinger, F., Guttman, B., McCarrin, M., Roussev, V., and White, D. (2014b). Approximate matching: definition and terminology. *NIST Special Publication*, 800:168.

Breitinger, F., Rathgeb, C., and Baier, H. (2014c). An efficient similarity digests database lookup-a logarithmic divide & conquer approach. *The Journal of Digital Forensics, Security and Law: JDFSL*, 9(2):155.

Harichandran, V. S., Breitinger, F., and Baggili, I. (2016). Bytewise approximate matching: The good, the bad, and the unknown. *The Journal of Digital Forensics, Security and Law: JDFSL*, 11(2):59.

Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97.

Martínez, V. G., Álvarez, F. H., and Encinas, L. H. (2014). State of the art in similarity preserving hashing functions. In *Proceedings of the International Conference on Security and Management (SAM)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

Moia, V. H. G. and Henriques, M. A. A. (2017). A comparative analysis about similarity search strategies for digital forensics investigations. In *XXXV Simpósio Brasileiro de Telecomunicacões e Processamento de Sinais (SBrT 2017)*, São Pedro, Brazil.

NIST (2016). National software reference library. `http://www.nsrl.nist.gov/`. Accessed 2016 Set 13.

Roussev, V. (2010). Data fingerprinting with similarity digests. In *IFIP International Conf. on Digital Forensics*, pages 207–226. Springer.

Winter, C., Schneider, M., and Yannikos, Y. (2013). F2s2: Fast forensic similarity search through indexing piecewise hash signatures. *Digital Investigation*, 10(4):361–371.

 © 2017 Sociedade Brasileira de Computação