State of the Art in Similarity Preserving Hashing Functions

V. Gayoso Martínez, F. Hernández Álvarez, and L. Hernández Encinas
Information Processing and Cryptography (TIC), Institute of Physical and Information Technologies (ITEFI)
Spanish National Research Council (CSIC), Madrid, Spain

Abstract— *One of the goals of digital forensics is to analyse the content of digital devices by reducing its size and complexity. Similarity preserving hashing functions help to accomplish that mission through a resemblance comparison between different files. Some of the best-known functions of this type are the context-triggered piecewise hashing functions, which create a signature formed by several hashes of the initial file. In this contribution, we present the state of the art of the most important similarity preserving hashing functions, analysing their main features. We conclude our work listing the most relevant properties that such type of functions should satisfy in order to improve their efficiency.*

Keywords: Forensics, Hash Functions, Similarity Preserving

1. Introduction

In modern society, the amount of information has increased in an incommensurable way and therefore the management of big quantities of data represents a major challenge. Digital forensics is the branch of Computer Science which, through investigation and analysis techniques, gathers evidence from the content of a particular electronic device in a way that is suitable for presentation in a court of law, for example. When inspecting the content of a computer, digital forensics experts need to reduce the large amount of data available to them to information that can be analysed in an easier way.

An initial approach to reach that reduction is using cryptographic hash functions. Hashing algorithms like MD5 [22] and the family SHA [17], [19], [20], among others, have been traditionally used in computer forensics to determine if two files were the same. Given the importance of this topic, NIST (National Institute of Standards and Technology) developed a database, called NSRL (National Software Reference Library), which contains hash values of files of several trusted operating systems [18]. With this public service, NIST contributes to reduce the search time of known files and to detect content forgery on the devices. However, the main limitation of cryptographic hash functions when comparing files is that, if one of the files is modified, the outcome of the comparison is negative, even if the two files are identical except in one byte.

In contrast to cryptographic hash functions, Similarity Preserving Hashing Functions (SPHF), also known as Piecewise Hashing Functions (PHF) or fuzzy hashing functions, aim to detect the resemblance between two files by mapping similar inputs to similar hash values. These functions, which compare files at byte level, are useful in order to compare a broader range of input data and detect not only similar texts, but also embedded objects (e.g. a JPEG image in a Word document) or binary fragments (e.g. a data packet in a network connection or a virus inside an executable file).

The technique behind SPHF was originally devised by Harbour [12], and consists in creating a signature formed by several hashes of the initial file, instead of only one. In this way, even if part of the content is modified, only the hashes related to that updated parts would change, allowing to detect if the rest of the file is related or similar to the original one. There are four types of SPHF:

- *Block-Based Hashing (BBH)*: functions that produce a hash after a fixed amount of bytes have been handled, so the number of hashes depends directly on the data object size and the length of the hash input.
- *Context-Triggered Piecewise Hashing (CTPH)*: functions where the number of hashes is determined by the existence of special points, called *trigger points*, within the data object. A point is considered to be a trigger point if it matches a certain property, defined in a way so that the number of expected trigger points falls within a range.
- *Statistically-Improbable Features (SIF)*: the basic aim of this functions is to identify a set of features (sequence of bits) which are least likely to occur in each of the data objects by chance and then compare the features themselves to obtain the similarity level.
- *Block-Based Rebuilding (BBR)*: these functions make use of external auxiliary data, such as binary blocks, to compare the bytes of the original file and calculate the differences between them (e.g. using the Hamming distance). Then, these differences are used as a base to find possible similar data objects.

In this paper, we present a study about the most important similarity preserving hashing functions, including a study of their main properties, and we conclude our work listing the main properties that such type of functions should satisfy in order to improve their efficiency. The rest of this paper is organized as follows: Section 2 summarizes the block-based hashing functions, whereas in Section 3 context triggered piecewise hashing functions are presented. In Section 4 functions based on statistically-improbable features are analyzed. Block-based rebuilding functions are studied in Section 5. Finally, Section 6 summarizes our conclusions in this topic.

2. Block-Based Hashing

The most basic scheme that can be used for determining similarity of binary data is Block-Based Hashing (BBH). In short, using this method cryptographic hashes are generated and stored for every block of a chosen fixed size (e.g. 512 bytes). Later, the block-level hashes from two different sources can be compared and, by counting the number of blocks in common, a measure of similarity can be determined.

An example of this kind of similarity hashing functions was performed by Harbour, who developed a program called `dcfldd` [12]. This software splits the input data into sectors or blocks of a fixed length and computes the corresponding cryptographic hash value for each of these blocks.

The main advantage of this scheme is that it is already supported by existing hashing tools and it is computationally efficient. The disadvantages become fairly obvious when block-level hashing is applied to files: success heavily depends on the intrinsic layout of the files being very similar. For example, if we search for versions of a given text document, a simple character insertion/deletion towards the beginning of the file could render all block hashes different. This means that `dcfldd` is not alignment robust.

Similarly, block-based hashes will not tell us if an object, such as a JPEG image, is embedded in a compound document, such as a Microsoft Word document. In short, the scheme is too fragile and a negative result does not reveal any useful information.

3. Context Triggered Piecewise Hashing

The second type of piecewise hashing functions, which are usually known as Context Triggered Piecewise Hashing (CTPH) functions, were originally proposed by Tridgell [30]. Later, Tridgell developed a context triggered piecewise hashing based algorithm to identify mails which are similar to known spam mails. He called his software `spamsum` [31]. The basic idea is to identify content markers, called *contexts*, within a binary data object and to store the sequences of hashes for each of the pieces, also called *chunks*, in between contexts. In other words, the boundaries of the chunk hashes are not determined by an arbitrary fixed block size but are based on the content of the object.

Nowadays, `ssdeep` is the best known CTPH application, but another algorithms based in the same concepts or improvements to the original `ssdeep` algorithm have been proposed: `FKSum`, `SimFD`, `MRSH`, etc.

3.1 `ssdeep`

In 2006, Kornblum released `ssdeep` [15], one of the first programs for computing context triggered piecewise hashes. In this algorithm, blocks (chunks or segments) are not determined by an arbitrary fixed block size but are based on the content of the object.

The algorithm's core is a rolling hash very similar to the rolling hash used in `rsync` [30] and `spamsum` [31]. The rolling hash is used to identify a set of reset points (also known as distinguished points or triggered points) in the plaintext that depend on the content of a sliding window of seven bytes. The algorithm reaches a reset point whenever the rolling hash (which is based on the *Adler32* function) generates a value which meets a predefined criteria. Let

$$BS_p = B_{p-s+1}B_{p-s+2}\ldots B_p$$

denote the byte sequence in the current window of size s, which is 7 by default, at position p within the file, and let *PRF*(BS_p) be the corresponding rolling hash value. If *PRF*(BS_p) hits a certain value, the end of the current chunk is identified. So, the byte B_p is a trigger point and the current byte sequence BS_p a trigger sequence. The subsequent chunk starts at byte B_{p+1} and ends at the next trigger point or the end of the file. As there are only low-level operations, Kornblum's *PRF* is very fast in practice.

In order to define a hit for *PRF*(BS_p), Kornblum introduced a modulus, b, called block size, which determines the reset frequency. The byte B_p is a trigger point if and only if *PRF*$(BS_p) \equiv -1 \mod b$. If *PRF* outputs are equally distributed values, then the probability of a hit is reciprocally proportional to b. Thus if b is too small, we have too many trigger points and vice versa.

As Kornblum aims at having 64 chunks, the block size depends on the file size as given in Eq. (1), where b_{min} is the minimum block size with a default value of 3, S is the desired number of chunks with a default value of 64, and N is the file size in bytes (for a complete explanation of the formula and the chosen default values, please see [15]).

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

Given that $b \approx N/S$, the procedure generates as a result around S chunks. Once a chunk is identified, a second hash based on the *FNV* algorithm is then used to produce hash values of the content between two consecutive trigger points. Using its last 6 bits, each of those hash values is translated into a Base64 character, so the resulting signature is the concatenation of the single characters generated at all the trigger points (with a maximum of 64 characters per signature). In this way, if a new version of the object is created by localized insertions and deletions, some of the original chunk hashes will be modified, reordered, or deleted, but enough will remain in the new composite hash to identify the similarity.

As the frequency of the trigger points strongly determines how many characters will appear in the signature, at the beginning of its execution the algorithm estimates the value of the block size, which would theoretically produce a signature of around 64 characters. Once the signature is produced, if its length is less than 32 characters `ssdeep` adjusts the block size ($b \leftarrow b/2$) and the algorithm is executed one more time, which generates a new signature. This procedure continues until a signature of at least 32 characters is produced.

In the comparison process, `ssdeep` computes how similar are two files based on their signatures. The similarity measurement that `ssdeep` uses is an edit distance algorithm based on the Damerau-Levenshtein distance [11], [16], which compares the two strings and counts the minimum number of operations needed to transform one string into the other, where the allowed operations are insertions, deletions, and substitutions of a single character, and transpositions of two adjacent characters [13], [33].

In `ssdeep`, insertions and deletions are given a weight of 1, while substitutions are given a weight of 3, and transpositions a weight of 5. As an example, using `ssdeep`'s algorithm the distance between the strings “Saturday” and “Sundays” is 5, as it can be checked with the following steps and the computations of Table 1.

S**a**turday $\xrightarrow{\textit{del}}$ S**t**urday $\xrightarrow{\textit{del}}$ Su**r**day $\xrightarrow{\textit{del}}$ Suday
$\xrightarrow{\textit{ins}}$ Su**n**day $\xrightarrow{\textit{ins}}$ Sunday**s**

Table 1: `ssdeep` edit distance example.

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	2	3	2	3	4	5	6
n	3	2	3	4	3	4	5	6	7
d	4	3	4	5	4	5	4	5	6
a	5	4	3	4	5	6	5	4	5
y	6	5	4	5	6	7	6	5	4
s	7	6	5	6	7	8	7	6	**5**

A consequence of assigning the weights 3 and 5 to the substitution and transposition operations is that, in practice, the edit distance computed by `ssdeep` only takes into consideration insertions and deletions. In this way, a substitution has a cost of 2 (a deletion plus an insertion) instead of 3, and a transposition has also a weight of 2 (again an insertion and a deletion) instead of 5.

One of the limitations derived from this design is that, given a string, a rotated version of the initial string is credited with many insertion and deletion operations, when in its nature it is basically the same string (i.e., the content is the same, although the order of the substrings is different). Consider for example the strings “1234abcd” and “abcd1234”.

The resulting distance is then scaled to produce a score in the range 0-100, where a value of 100 indicates a perfect match and a score of 0 indicates a complete mismatch. There are two conditions that have to be taken into consideration at this point: if the two signatures have a different block size, then the score is automatically set to 0 without performing any additional calculation. Besides, if the minimum length of the longest common substring in the comparison is less than the windows size (7), then `ssdeep` provides a score of 0.

In retrospect, `ssdeep` represented a cornerstone in similarity detection techniques. Its source code is freely available, and there are implementations for Windows and Linux [21], [32]. The latest version of `ssdeep` is 2.10, which was released in July 2013. Even though `ssdeep` is not a multi-threaded program, the author states that the library on which its based can be used in multi-threaded applications [14].

Despite the benefits brought by the release of this program, during the last years some limitations have been brought to attention by different researchers, proposing improvements or even different theoretical approaches (for example, see [1], [2], [4], [5], [8], [10], [24]).

3.2 `FKSum`

A improvement to Kornblum's algorithm was proposed by Chen *et al.* [10]. In their algorithm, `FKSum`, they showed that it is possible to improve the efficiency of `ssdeep`, since until the very last step it does not examine the signature and, if it is too short (i.e., shorter than 32 characters), the file has to be processed again using an adapted block size $b \leftarrow b/2$.

As this fact is very likely to happen (they showed that it happens in 38% of the cases), the goal of their modification to the original algorithm was to generate intermediate hashes using numbers in the geometric progression with factor 4 as block size. If the current block size is b, they perform the same algorithm and compute the hashes with block sizes b and $4b$, counting the trigger points for block sizes b, $2b$, $4b$, and $8b$. As the authors used *FNV* as a homomorphic hash function, it is possible to create the hashes for $2b$ by using the hashes of b and hence the process runs more efficiently. The drawback of this approach is that the combination of the hashes might be only possible with the *FNV* hash. If we use any cryptographic hash function such as MD5, we would have to do more computations and the efficiency advantage would no longer be available.

3.3 New version of `ssdeep`

Breitinger and Baier [5] discussed the efficiency of `ssdeep`, presenting some enhancements that, in their opinion, would increase the performance of his algorithm by 55% if applied to a real life scenario:

- Each file should be processed only once. As it was proposed by Chen *et al.* [10], they use four different block size values: $2b$, b, $b/2$, and $b/4$.

- Implementation should be flexible in order to be able to change the *PRF* and chunk functions.
- It should be able to determine untypical behaviour of trigger sequences (which may be caused by an active adversary), in order to mitigate the security concerns regarding adversary attacks detected in [2].

Their main idea is to process the file once and count the trigger sequences for all reasonable block sizes (according to Kornblum approach). In the next step, the file is read again and the block size b is set to the largest value that yields at least 32 signature characters. This fact is the main disadvantage of this proposal, since the file has to be read twice.

An important point related to security is the restriction of the signature length of `ssdeep`. Kornblum forces the resulting hash length to be between 32 and 64, but he does not give any justification for those limits. Even though Breitinger and Baier considered that the upper boundary was a weakness, and as such it was exploited in [2], they maintained both limits.

3.4 `SimFD`

Seo *et al.* [29] developed the `SimFD` algorithm by combining `ssdeep` with other statistical analysis and improved the false positive rate, but at the cost of efficiency.

Statistical analysis uses byte frequency analysis to detect if a file is similar to the original one. This method generates a result through byte frequency analysis for the original file. As a result of it, the process is performed before any similar file detection task, and some *reference values* are established from the original file for comparing other files. These reference values consist of three types, where each type has a different purpose for detection.

The first reference values are computed for comparing similar files, and they are determined from features of the original file through numerical values obtained from data distribution. The second reference values are computed using metadata, such as file signature, header/footer, null values etc, which was eliminated during the computation of the first reference values. Finally, the third reference values are calculated as a clustering value for all binaries. The clustering scheme is divided into increase, decrease and stagnation for accumulated frequency. The clustering results have the advantage of grasping the distribution type for a file in a character string.

`SimFD` consists of four modules. First the input module, used by selecting the original copy and the target object. Then the analysis modules, that consists of the *CTPH* analysis module (mainly `ssdeep`) and the statistical analysis module. And finally, the detection result module, that judges final similarity by checking the results of *CTPH* and statistical analysis through the reference values.

3.5 `md5bloom`

Roussev *et al.* [26] proposed a new tool, `md5bloom`, which uses Bloom filters as an efficient tool for fast comparisons. Bloom filters are a space-efficient probabilistic data structure, first introduced by Bloom in [3], and widely used in areas such as network routing and traffic filtering. They allow to test whether an element is a member of a set.

A Bloom filter B is a representation of a set of n elements, $S = \{s_1, \ldots, s_n\}$, taken from a universe U. The filter consists of an array of m bits, initially all set to 0. To represent the set of elements, the filter uses k independent hash functions, $h_0, \ldots, h_{k-1}$, that produce values in the range of 0 to $m-1$. All hashes are assumed to be independent and to map elements from U uniformly over the range of the function.

To insert an element x from S, each hash function is applied to it, which gives k values. For each value, $h_1(x), \ldots, h_k(x)$, the bit with the corresponding number to one is set (setting a bit twice has the same effect as setting it once). To verify if an element x is in S, we must hash it with all the hash functions and check the corresponding bits: if all of them are set to one, we return *yes*; otherwise, *no*. The filter will never return a false negative; that is, if the element was inserted, the answer will always be *yes*. However, we could have a false positive for an element that has never been inserted but whose bits have been set by chance by other element insertions. False positives are the price we pay for the compression gains.

As it turns out, the routine use of cryptographic hashes in digital forensics makes it easy to introduce Bloom filters into the process. Instead of computing k separate hashes, we can take an object's cryptographic hash, split it into several nonoverlapping subhashes, and use them as if different hash functions had produced them. This is the way the `md5bloom` application works: the MD5 function returns 128 bits, any individual bit of the hash value can be viewed as an independent random variable and, by extension, any subset of the 128 hash bits can be selected to produce a value within a desired range. Let the bits in h^{md5} be numbered $0{:}127$ (we use the notation $h_{d_1:d_2}$ and the term subhash to denote the selection of bits numbered d_1 through d_2, inclusively), thus, $h^{md5} = h_{0:127}$ and can also be expressed as the concatenation of subhashes, for example:

$$h^{md5} = h_{0:15}h_{16:31}h_{32:47}h_{48:63}h_{64:79}h_{80:95}h_{96:111}h_{112:127}$$

3.6 `MRSH`–Multi-Resolution Similarity Hashing

In [27] Roussev *et al.* applied three main changes to the `ssdeep` algorithm:

1) First, they stated that it is not necessary to use a cryptographic hash function for the *PRF*. Therefore, instead of using the *PRF* based on *Adler32*, they used the following polynomial hash function, `djb2`:

$$h_0 = 5381; h_{k+1} = 33h_k + c_k \mod 232;\ \text{for } k \geq 0,$$

where c_k denotes the k^{th} character of the input.
Given that djb2 has the disadvantage that each window has to be processed from scratch, this change influences negatively the efficiency.

2) Second, they changed the hash function for processing each chunk. Instead of the *FNV* hash, they used MD5. Then, the least significant 11 bits of the MD5 output are used as input for a Bloom filter to represent the final signature.
3) The next step in the design process is to determine whether the composite hash will be of fixed or variable size. Fixed-size hashes have an obvious appeal (minimum storage requirements and simple management). However, they also have some scalability issues as they limit the ability of the hashing scheme to compare files of varying sizes. md5bloom, on its own, has a very similar problem if the attempt is to produce a composite hash which consists of a single filter. Moreover, to compare two filters, they must be of the same size and use the same hash functions.
This analysis points out the need to devise a variable-sized hashing scheme that scales with the object size but also maintains a low relative overhead. In this sense, to enable universal comparison of filters Roussev standardized a set of Bloom filters of 256 bytes, 8 bits per element, using four hash functions. To obtain the four hashes, they take the MD5 chunk hash, split it into four 32-bit numbers and take the least significant 11 bits from each part.

With all these design changes, the process of this new algorithm works in the following way:

1) A 32-bit djb2 hash is computed on a sliding window of size 7. At each step, the least significant t bits of the hash (the trigger) are examined, and if they are all set to 1, a context discovery is declared; t is the essential parameter that distinguishes the different levels of resolution. For the lowest level 0, the default value is 8.
2) Context discovery triggers the computation of the MD5 chunk hash between the previous context and the current one.
3) The chunk hash is split into four pieces and four corresponding 11-bit hashes are obtained and inserted into the current Bloom filter. If the number of elements in the current filter reaches the maximum allowed (256), a new filter is added at the end of the list and becomes the current one.
4) The hash consists of the concatenation of all the Bloom filters, preceded by their total count.

Even though this modification slows down ssdeep, it increases the security aspects, therefore this change is considered to be very useful.

3.7 MRSH v2–Multi-Resolution Similarity Hashing, version 2

In [8] Breitinger *et al.* reviewed in terms of efficiency and performance the parameters used in the Multi-Resolution Similarity Hashing function (MRSH) proposed by Roussev (see §3.6) and developed a new version, MRSH v2, which recovers some of the original ssdeep parameters, such as *Adler32* and *FNV*.

In order to be more efficient, they decided to use again the original rolling hash (*Adler32*) instead of djb2, since it computes the hash value over the 7-byte window in an easier way just by removing the last byte and adding the new one instead of doing seven loops per window. Moreover, as collision resistance is not necessary, the new version makes use of *FNV*, as the original ssdeep, instead of MD5. For performance reasons they stated that the minimum block size should be $b/4$, which is in line with FKSum (see §3.2). Finally, the maximum number of elements has been changed to 160 and 5 subhashes are used. The maximum is therefore 800 bits, so one Bloom filter could represent approximately 40,960 bytes. In order to insert the chunk hash value into a Bloom filter, they used the least significant $k \cdot \log_2(m)$ bits (MRSH divides the chunk hash values).

Additionally, they demonstrated that the algorithm is compliant with the five properties that in their opinion a SPHF should have, namely: compression, ease of computation, similarity score, coverage, and obfuscation resistance.

4. Statistically-Improbable Features

This approach is based on the idea that finding similarities between two objects can be understood as identifying a set of features in each of the objects and then comparing the features themselves. A feature in this context is simply a sequence of consecutive bits selected by some criterion from the object.

Roussev [23] uses entropy as the way of finding statistically-improbable features and measures the false positive range for different kind of files (doc, xls, txt, html, pdf, etc.). With this idea, he proposed a new algorithm, called sdhash, whose goal is to pick object features that are least likely to occur in other data objects by chance.

Instead of dividing an input into pieces, sdhash identifies statistically-improbable features using an entropy calculation. These characteristic features, forming a sequence of length 64 bytes, are then hashed using the cryptographic hash function SHA-1 and inserted into a Bloom filter. Hence, files are similar if they share identical features.

A security analysis was performed by Breitinger *et al.* in [9], finding some bugs in the implementation as well as showing some possible attacks to circumvent the algorithm and analyzing the resistance of the parameters designed. Another analysis [7] in terms of measuring the compression, ease of computation, coverage and similarity score showed different weaknesses of `sdhash`, e.g. there is no full coverage (a change up to 20% of the input does not alter the fingerprint) and that the chosen design of the comparison function is made for fragment detection but not for comparing two files.

Moreover, in [24] Roussev performs a comparison between `sdhash` and `ssdeep`, analyzing two different experiments (random files and real files) with three different scenarios (embedded object detection, single-common-block file correlation, and multiple-common-blocks file correlation) concluding that `sdhash`'s accuracy and scalability outperforms `ssdeep`.

Finally, in [25] the `sdhash` basic algorithm was made scalable by parallelizing it. The new modification was called `sdhash-dd` and to reach this objective, some chain dependencies among the Bloom filter component filters were moved away in order to allow concurrent generation. The idea was to split the target into blocks of fixed size and run the signature generation in block-parallel fashion.

5. Block-Based Rebuilding

There are mainly three algorithms, `SimHash`, `mvHash-B`, and `bbHash`, which make use of external auxiliary data or blocks that can be chosen randomly, uniformly or as a fixed base, in order to rebuild a file. The process compares the bytes of the original file to the auxiliary data and calculates the differences between them (e.g., using the Hamming distance).

5.1 `SimHash`

Sadowski *et al.* presented an algorithm, called `SimHash` [28], which preselects 16 blocks of 8 bits each in order to find matches by scanning and comparing the original file to these blocks. When a match is found, it is stored in a sum table and then the hash key is computed as a function of the sum entries. Another function, called *SimFind*, identifies the files with key values within a certain threshold of a particular file, then performs a pairwise comparison among the sum table entries to return a filtered selection of similar files.

They performed some experiments using a Uniform Key which has all the 16 blocks weighted equally and a Skew Key which has uneven weights in 4 of the blocks.

5.2 `mvhash-B`

The `mvhash-B` function was described by Breitinger *et al.* [4], having three phases to create the fingerprint:

1) First, majority voting is used to map every byte of the input file to either 0x00 or 0xFF. Majority voting in this case means counting the amount of 0s/1s in the neighborhood of the currently processed input byte. If the neighborhood is crowded by 1s, the majority vote yields an output 0xFF and vice versa.
2) Next, Run Length Encoding (RLE) compresses these sequences of 0x00s or 0xFFs bytes.
3) Finally, the RLE sequence is inserted into Bloom filters to represent the actual fingerprint.

By design, `mvHash-B` aims at having a fingerprint length of 0.5% of the input length, but a drawback of this implementation is the dependence on the file type: each file type requires its own configuration (no standard configuration works for all file types). In other words, although `mvhash-B` works on the byte level, it needs different configurations.

5.3 `bbHash`

Another example of this way of finding similar files is the `bbHash` function, designed by Breitinger *et al.* in [6]. Their new fuzzy hashing technique is based on two concepts:

- *Deduplication*: is a backup scheme for saving files efficiently because instead of saving it completely, it makes use of small pieces. If two files share a common piece, it is only saved once, but referenced for both files.
- *Eigenfaces*: they are used in biometrics for face recognition, for example by representing any face as a combination of a set of N eigenfaces previously selected.

In this algorithm, they use a fixed set of N random byte sequences called building blocks of 128 bytes. The process is to slide through the file byte-by-byte and compute the Hamming distance of all building blocks against the current input sequence. If the building block with the smallest Hamming distance is smaller than a certain threshold, its index contributes to the files's hashing result. The disadvantage of this algorithm is that its runtime is high.

6. Conclusions

Breitinger and Baier presented a list of four general properties for SPHF [7], which they later extended to the following five general characteristics [8]:

1) *Compression*: the output must be much smaller than the input for space-saving and performance reasons.
2) *Ease of computation*: generating the hash value of a given file and making comparisons between files must be a fast procedure.
3) *Similarity score*: the comparison function must provide a number which represents a matching percentage value.
4) *Coverage*: every byte of the input must be used to calculate the hash value.
5) *Obfuscation resistance*: it must be difficult to obtain a false negative/false positive result, even after manipulating the input data.

Nevertheless, after analysing the characteristics of these functions, we have been able to identify a list of additional specific features that any SPHF should provide, either by improving a feature already existing or implementing it for the first time. These additional features are:

- *Generate more realistic results consistent with the content of the files compared*: this requirement implies the existence of a simple and clear definition of the concept of similarity, and how to express it as a number.
- *Detect content rotation*: by rotation we mean moving some part of the end of the document to the beginning (or vice versa). A visual examination of a pair of such rotated files would provide a result close to 100, as both files have the same content.
- *Detect content swapping*: by content swapping we mean taking some portion of the document and moving it into a different location, so graphically it could be seen as moving data blocks inside the document. For a file where several swaps have been made, the result should be close to 100, as again the content of both files is basically the same.
- *Compare files of different sizes without limit*: in some cases, it is necessary to compare files of very dissimilar sizes (e.g., considering a book, this could be seen as comparing one chapter with ten chapters in order to detect plagiarism).
- *Avoid insertion attacks*: there are two ways by which a user could alter the comparison results. He could repeatedly insert a specific byte string at the beginning of the file, or could insert a specific byte string scattered along the document (not necessarily at the beginning). Any new piecewise hashing application should try to provide countermeasures to ameliorate the effects of this type of attacks, at least to some extent.

Acknowledgements

This work has been partially supported by the Ministerio de Ciencia e Innovación (España) under the project TIN2011-22668.

References

[1] K. Astebøl, "mvHash–A new approach for fuzzy hashing," Master's thesis, Gjøvik University College, 2012.

[2] H. Baier and F. Breitinger, "Security aspects of piecewise hashing in computer forensics," in *Sixth International Conference on IT Security Incident Management and IT Forensics (IMF 2001)*, 2011, pp. 21–36.

[3] B. H. Bloom, "Space time tradeoffs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, 1970.

[4] F. Breitinger, K. Astebøl, H. Baier, and C. Busch, "mvhash-b - A new approach for similarity preserving hashing," in *Seventh International Conference on IT Security Incident Management and IT Forensics (IMF 2013)*, 2013, pp. 33–44.

[5] F. Breitinger and H. Baier, "Performance issues about context-triggered piecewise hashing," in *Proc. of 3rd ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*, vol. 3, 2011.

[6] ——, "A fuzzy hashing approach based on random sequences and hamming distance," in *7th annual Conference on Digital Forensics, Security and Law (ADFSL 2012)*, 2012.

[7] ——, "Properties of a similarity preserving hash function and their realization in sdhash," in *Information Security for South Africa (ISSA 2012)*, 2012, pp. 1–8.

[8] ——, "Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2," in *Digital Forensics and Cyber Crime*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, M. Rogers and K. Seigfried-Spellar, Eds. Springer Berlin Heidelberg, 2013, vol. 114, pp. 167–182.

[9] F. Breitinger, H. Baier, and J. Beckingham, "Security and implementation analysis of the similarity digest sdhash," in *1st International Baltic Conference on Network Security & Forensics (NeSeFo 2012)*, 2012.

[10] L. Chen and G. Wang, "An efficient piecewise hashing method for computer forensic," in *Proc. of Workshop on knowlegde discovery and data mining*, IEEE, Ed., 2008, pp. 635–638.

[11] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Communications of the ACM*, vol. 7, no. 3, pp. 171–176, 1964.

[12] N. Harbour, "Dcfldd. defense computer forensics lab," 2002. [Online]. Available: http://dcfldd.sourceforge.net

[13] M. Karpinski, "On approximate string matching," *Lecture Notes in Computer Science*, vol. 158, pp. 487–495, 1983.

[14] J. Kornblum, "ssdeep 2.10 released." [Online]. Available: http://jessekornblum.livejournal.com/293679.html

[15] ——, "Identifying almost identical files using context trigger piecewise hashing," *Digital Investigation*, vol. 3(S1), pp. 91–97, 2006.

[16] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707 – 710, 1966. [Online]. Available: http://profs.sci.univr.it/~liptak/ALBioinfo/files/levenshtein66.pdf

[17] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of applied Cryptography*. Boca Raton, FL: CRC Press, 1997.

[18] NIST. National Software Reference Library. [Online]. Available: http://www.nsrl.nist.gov

[19] ——, "SHA-3 competition," National Institute of Standards and Technology, 2012, http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[20] ——, "The Keccak sponge function family," 2013, http://keccak.noekeon.org/specs_summary.html.

[21] Python Software Foundation. (2013) ssdeep python wrapper. [Online]. Available: https://pypi.python.org/pypi/ssdeep

[22] R. L. Rivest, "The MD5 message digest algorithm," 1992, request for comments (RFC 1321), Internet Activity Boards, Internet Privacy Task Force.

[23] V. Roussev, "Building a better similarity drap with statistically improbable features," in *Proc. of 42 Hawaii International Conference on System Science*, 2009, pp. 1–10.

[24] ——, "An evaluation of forensic similarity hashes," *Digital Investigation*, vol. 8, Supplement, no. 0, pp. 34 – 41, 2011.

[25] ——, "Scalable data correlation," in *Proc. of International Conference on Digital Forensics (IFIP WG 11.9)*, 2012.

[26] V. Roussev, Y. Chen, T. Bourg, and G. Richard, "Md5bloom: Forensic filesystem hashing revisited," *Digital Investigation*, vol. 3, pp. 82–90, 2006.

[27] V. Roussev, G. Richard, and L. Marziale, "Multi-resolution similarity hashing," *Digital Investigation*, vol. 4, Supplement, no. 0, pp. 105 – 113, 2007.

[28] C. Sadowsky and G. Levin, "Simhash: Hash-based similarity detection," Tech. Rep., 2007. [Online]. Available: http://simhash.googlecode.com/svn/trunk/paper/SimHashWithBib.pdf

[29] K. Seo, K. Lim, J. Choi, K. Chang, and S. Lee, "Detecting similar files based on hash and statistical analysis for digital forensic investigation," in *2nd International Conference on Computer Science and its Applications (CSA '09)*, 2009, pp. 1–6.

[30] A. Tridgell, "Efficient algorithms for sorting and synchronization," Master's thesis, The Australian National University. Department of Computer Science, Canberra, Australia, 1999.

[31] ——, "Spamsum readme," 1999. [Online]. Available: http://samba.org/ftp/unpacked/junkcode/spamsum/README

[32] ——. (2013) Getting started with ssdeep. [Online]. Available: http://ssdeep.sourceforge.net/usage.html

[33] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974.