A New Edit Distance for Fuzzy Hashing Applications

V. Gayoso Martínez[1], **F. Hernández Álvarez**[1], **L. Hernández Encinas**[1], **and C. Sánchez Ávila**[2]
[1]Information Processing and Cryptography (TIC), Institute of Physical and Information Technologies (ITEFI)
Spanish National Research Council (CSIC), Madrid, Spain
[2]Telecommunication Engineering School (ETSIT), Polytechnic University of Madrid (UPM), Madrid, Spain

Abstract— *Similarity preserving hashing applications, also known as fuzzy hashing functions, help to analyse the content of digital devices by performing a resemblance comparison between different files. In practice, the similarity matching procedure is a two-step process, where first a signature associated to the files under comparison is generated, and then a comparison of the signatures themselves is performed.*

Even though `ssdeep` *is the best-known application in this field, the edit distance algorithm that* `ssdeep` *uses for performing the signature comparison is not well-suited for certain scenarios. In this contribution we present a new edit distance algorithm that better reflects the similarity of two strings, and that can be used by fuzzy hashing applications in order to improve their results.*

Keywords: Edit distance, fuzzy hashing, similarity preserving hashing

1. Introduction

Similarity Preserving Hashing (SPH) functions, also known as fuzzy hashing algorithms, try to detect the resemblance between two files [1]. There are basically four types of SPH functions [2]: Block-Based Hashing (BBH) functions, Context-Triggered Piecewise Hashing (CTPH) functions, Statistically-Improbable Features (SIF) functions, and Block-Based Rebuilding (BBR) functions. In any fuzzy hashing application, files are processed and, as a result of the analysis performed, a code linked to the content of the file is generated, so files can be later compared based on their codes. In this context, the file's code is indistinctly referred to as its digest, hash or signature.

In CTPH functions, the length and content of the signature is determined by the existence of certain special points, called trigger points or distinguished 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 previously specified range. Once a number of trigger points large enough is detected, CTPH applications generate the signature associated to the file by processing the data portions located between consecutive trigger points.

Since its first release, `ssdeep` [3] has been one of the best known fuzzy hashing applications. When comparing files, `ssdeep` generates a matching score after analysing the similarity of the signatures. In order to do that, `ssdeep` implements an edit distance algorithm based on the Damerau-Levenshtein distance between two strings [4], [5]. That edit distance function compares the two strings and counts the minimum number of operations needed to transform one into the other, where the allowed operations are insertions, deletions, and substitutions of a single character, and transpositions of two adjacent characters [6], [7].

Even though the success of `ssdeep` is quite remarkable, its edit distance implementation has important limitations that prevent `ssdeep` from generating a score that reflects the percentage of the bigger file that is also present in the smaller file, which is the definition of similarity better adapted for some real-world scenarios. With the goal to improve the quality of fuzzy hashing applications, in this contribution we present a new edit distance algorithm that can be used as a replacement of `ssdeep`'s edit distance or in new implementations.

The rest of this paper is organized as follows: Section 2 reviews `ssdeep`'s edit distance. In Section 3, we provide a complete description of our proposed algorithm. Section 4 includes a comparison of both algorithms when working with some special signatures. Finally, Section 5 summarizes our conclusions about this topic.

2. Edit distance in `ssdeep`

In 2006, Jesse Kornblum released `ssdeep` [8], one of the first programs for computing context triggered piecewise hashes, and that soon became very popular. Since that initial release, new versions and updates have not ceased to appear, and the project is still active (at the time of preparing this contribution, the latest version is 2.12, which was released in October 2014 [3]). The core of `ssdeep` is derived from `rsync` [9] and `spamsum` [10], both of them tools developed by Andrew Trigdell.

As mentioned in the previous section, the similarity measurement that `ssdeep` uses is an edit distance algorithm based on the Damerau-Levenshtein distance [4], [5], [6], [7]. In the original Damerau-Levenshtein algorithm, all the operation costs are initially 1, though the substitutions and transpositions decrease their weight to 0 when certain conditions are met [11]. In comparison, `ssdeep` defines the weight of insertions and deletions as 1, the weight of substitutions as 3, and the weight of transpositions as 5.

As an example, using `ssdeep`'s algorithm the distance between the strings "emerald" and "overall" is 6, as it can be checked with the following steps and the computations of Table 1.

$$\textbf{e}\text{merald} \xrightarrow{del} \textbf{m}\text{erald} \xrightarrow{del} \text{erald} \xrightarrow{ins} \underline{\text{v}}\text{erald} \xrightarrow{ins}$$
$$\underline{\text{o}}\text{veral}\textbf{d} \xrightarrow{del} \text{overal} \xrightarrow{ins} \text{overal}\underline{\text{l}}$$

		e	m	e	r	a	l	d
	0	1	2	3	4	5	6	7
o	1	2	3	4	5	6	7	8
v	2	3	4	5	6	7	8	9
e	3	2	3	4	5	6	7	8
r	4	3	4	5	4	5	6	7
a	5	4	5	6	5	4	5	6
l	6	5	6	7	6	5	4	5
l	7	6	7	8	7	6	5	**6**

Table 1: `ssdeep` edit distance example.

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".

As the signature comparison algorithm implemented by `ssdeep` is not available in a descriptive way, Algorithm 1 shows our interpretation (made upon inspection of the source code of `ssdeep` [3]) of that functionality, where `A` and `B` are one-dimensional arrays containing, respectively, the m characters of `string1` and the n characters of `string2`, and where `D` is a $(m+1) \times (n+1)$ matrix used in the computations with all its positions initially set to 0. During the set-up phase, the first row (respectively, the first column) of `D` is initialized with the number corresponding to the column (respectively, the row) of the position being processed. The rest of the positions are processed based on the content of the nearby elements and the characters being compared. Once the comparison procedure is finished, the algorithm generates a similarity score in the range 0-100.

The meaning of the functions included in the algorithm is the following:

- `length(string)`: calculates the number of characters of the string.
- `longestCommonSuString(string1,string2)`: provides the longest common substring of two strings.
- `min(param1,param2,param3)`: identifies the minimum value given by the numbers or expressions passed to the function as parameters.
- `floor(value)`: returns the bigger integer whose value is equal to or lower than `value`.

3. Our proposed edit distance

Our edit distance algorithm compares two signature strings, `string1` and `string2`, and produces a similarity score in the range 0-100. Algorithm 2 describes all the steps that must be performed in order to evaluate the similarity of the strings `string1` and `string2`, where the first step consists in identifying as `string1` the shortest string and as `string2` the longest string, swapping the strings if necessary. During the procedure, the algorithm manipulates modified versions of the input strings, using their longest common substring for deciding which modification to perform next and increasing a counter with the differences found so far. The procedure is repeated until there are no more common substrings for the modified versions of the input elements. In the final step, the algorithm compares the resulting strings character by character in order to add to the counter the number of difference elements found for the same positions. It is important to point out that, unlike `ssdeep`, our algorithm does not impose a minimum length for the longest common substring, which allows to compare a wider range of strings.

The meaning of the functions included in Algorithm 2 and not presented in the previous section is the following:

- `longestCommonSuStringNoHyphen(string1, string2)`: returns the longest common substring which does not contain the hyphen (-) character.
- `hyphenString(size)`: creates a new string of length `size` containing only the hyphen character.
- `indexOf(string,substring)`: returns the position where the first character of `substring` is located inside `string`.
- `replace(string,index,size,substring)`: replaces in the element `string` the existing substring of `size` characters starting at `index` with the characters of `substring`.
- `abs(number)`: provides the absolute value of the input number.
- `charAt(string,index)`: returns the character located at position `index` in the element `string`.

In order to illustrate the comparison process performed by Algorithm 2, Table 2 provides an example using two ad-hoc strings, denoted as `string1` and `string2`. In the first row of the table, we have included the two initial strings (renamed as `string1temp` and `string2temp`), the template for the modified version of `string2` (called `string2mod`), and the score, which initially equals 0. Starting with the step 1, the element `substring` identifies the longest common substring of `string1temp` and `string2temp`, which are then updated to show the removal of that substring. Then, we have inserted the common

Algorithm 1 ssdeep edit distance algorithm.

```
if (length(longestCommonSuString(string1,string2)) $< 7$) then
    return 0
end if
$\lambda_{del} \leftarrow 1$
$\lambda_{ins} \leftarrow 1$
$\lambda_{sub} \leftarrow 3$
i $\leftarrow 0$
for all $i \leq m$ do
    D[i,0] $\leftarrow$ i
end for
j $\leftarrow 0$
for all $j \leq n$ do
    D[0,j] $\leftarrow$ j
end for
i $\leftarrow 1$
j $\leftarrow 1$
for all $i \leq m$ do
    for all $j \leq n$ do
        if (A[i] = B[j]) then
            $\lambda_{sub} \leftarrow 0$
        else
            $\lambda_{sub} \leftarrow 3$
        end if
        D[i, j] = min(D[i-1,j] + $\lambda_{ins}$, D[i,j-1] + $\lambda_{del}$, D[i-1,j-1] + $\lambda_{sub}$)
    end for
end for
score $\leftarrow$ D[m,n]
score $\leftarrow$ floor$\left(\frac{\text{score} \cdot 100}{\text{length(string1)} + \text{length(string2)}}\right)$
if (score $> 100$ ) then
    score $\leftarrow 0$
else
    score $\leftarrow$ 100 - score
end if
return score
```

longest substring obtained in that step into string2mod, so the position of that substring in string2mod is the same that it occupies in string1.

As described in Algorithm 2, we only increase the score if the difference between the initial and final positions of the substring in string2mod is greater than the length difference of string1 and string2. With this rule we avoid to penalize the change of positions derived from the different length of the strings under comparison (e.g. this difference could have been produced by the insertion of some characters at the beginning of the string, which would displace the rest of the characters that compose the original string a given number of positions).

The score is increased in one unit if the longest common substring has more than one character, which means that common substrings of different sizes would receive the same penalty (i.e., a penalty of 1.0, but only if they are separated a number of positions bigger than the difference of the string lengths). In this sense, what we penalize is the movement of the string, not its size.

Besides, when the longest common substring has exactly one character, the quantity to be added to the score is 0.5. The reason for doing this is not to penalize in excess the displacement of a unique character. If we do not impose this rule, the displacement of a single character would receive a score of 1.0, which would be the same penalty produced by the substitution of a character by a completely different character. A topic open for future study is the modification of this value in order to obtain better results.

After the rearrangement phase, a pair by pair comparison of the characters elements is performed in the last step of the procedure. As there are eight different characters

Algorithm 2 Our proposed edit distance algorithm.

```
if (length(string1) $>$ length(string2)) then
  string1 $\leftrightarrow$ string2
end if
string1temp $\leftarrow$ string1
string2temp $\leftarrow$ string2
common $\leftarrow$ longestCommonSuStringNoHyphen(string1temp,string2temp)
string2mod $\leftarrow$ hyphenString(length(string2))
diff $\leftarrow 0$
while (length(common) $> 0$) do
  pos1 $\leftarrow$ indexOf(string1temp,common)
  pos2 $\leftarrow$ indexOf(string2temp,common)
  string2mod $\leftarrow$ replace(string2mod,pos1,length(common),common)
  if (abs(pos1-pos2) $>$ abs(length(string1)-length(string2))) then
    if (length(common) $> 1$) then
      diff $\leftarrow$ diff + 1
    else
      diff $\leftarrow$ diff + 0.5
    end if
  end if
  string1temp $\leftarrow$ replace(string1temp,pos1,length(common),
                          hyphenString(length(common)))
  string2temp $\leftarrow$ replace(string2temp,pos2,length(common),
                          hyphenString(length(common)))
  common $\leftarrow$ longestCommonSuStringNoHyphen(string1temp,string2temp)
end while
for all $i$ such that $0 \leq i \leq$ length(string2temp) do
  char $\leftarrow$ charAt(string2temp,$i$)
  if char $\neq$ "-" then
    pos2 $\leftarrow$ indexOf(string2mod,"-")
    string2mod $\leftarrow$ replace(string2mod,pos2,1,char)
  end if
end for
for all $i$ such that $0 \leq i \leq$ length(string2temp) do
  if (($i \geq$ length(string1)) or (charAt(string1,$i$) $\neq$ charAt(string2mod,$i$)) then
    diff $\leftarrow$ diff + 1
  end if
end for
return floor$\left(100 - \frac{\texttt{diff} \cdot 100}{\texttt{length(string2)}}\right)$
```

in string1 and string2mod, the score is increased in eight units from 4.5 up to 12.5. In order to facilitate the identification of the dissimilar characters, Table 2 displays in bold font the dissimilar elements of the two strings.

Taking into account that the length of the longest string (string2) is 27, the comparison between string1 and string2 provides the following output:

$$\text{Result} = 100 - \left\lfloor \frac{12.5 \cdot 100}{27} \right\rfloor = 100 - 46 = 54.$$

A score of 54 implies that 54% of the longest string, string2, is also contained in the shorter string, string1.

4. Special signatures

When designing this test, our goal was to check the behaviour of ssdeep's algorithm and our proposed algorithm when using some special strings, whose pattern could appear in certain real-world scenarios (for example, when obtaining the signature of files containing lists of elements such as names, file paths, etc.).

Even though we are aware that the tests included below represent extreme cases with ad-hoc strings, we believe it is worthwhile to test both algorithms in this scenario, as it represents different degrees of content rotation and modification.

Step	Element	Content
0	string1temp	ABCDEFGHIJKLMNOPQRSTUVWXYZ
	string2temp	1XYZI2JKL3MNOPQ4BCDEFGH5678
	string2mod	---------------------------
	score	0.0
1	substring	BCDEFGH
	string1temp	A-------IJKLMNOPQRSTUVWXYZ
	string2temp	1XYZI2JKL3MNOPQ4-------5678
	string2mod	-BCDEFGH-------------------
	score	1.0
2	substring	MNOPQ
	string1temp	A-------IJKL-----RSTUVWXYZ
	string2temp	1XYZI2JKL3-----4-------5678
	string2mod	-BCDEFGH----MNOPQ----------
	score	2.0
3	substring	JKL
	string1temp	A-------I--------RSTUVWXYZ
	string2temp	1XYZI2---3-----4-------5678
	string2mod	-BCDEFGH-JKLMNOPQ----------
	score	3.0
4	substring	XYZ
	string1temp	A-------I--------RSTUVW---
	string2temp	1---I2---3-----4-------5678
	string2mod	-BCDEFGH-JKLMNOPQ------XYZ-
	score	4.0
5	substring	I
	string1temp	A----------------RSTUVW---
	string2temp	1----2---3-----4-------5678
	string2mod	-BCDEFGHIJKLMNOPQ------XYZ-
	score	4.5
6	string1	**A**BCDEFGHIJKLMNOPQ**RSTUVW**XYZ
	string2mod	**1**BCDEFGHIJKLMNOPQ**234567**XYZ**8**
	score	12.5

Table 2: String rearrangement example.

The strings included in this test are the following ones:

S01: `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijk lmnopqrstuvwxyz`
S02: `ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJK LMNOPQRSTUVWXYZ`
S03: `abcdefghijklmnopqrstuvwxyzABCDEFGHIJK LMNOPQRSTUVWXYZ`
S04: `12345678901234567890123456ABCDEFGHIJK LMNOPQRSTUVWXYZ`
S05: `BADCFEHGJILKNMPORQTSVUXWZYbadcfehgjil knmporqtsvuxwzy`
S06: `CDABGHEFKLIJOPMNSTQRWXUVabYZefcdijghm nklqropuvstyzwx`
S07: `EFGHABCDMNOPIJKLUVWXQRSTcdefYZabklmng hijstuvopqrwxyz`
S08: `IJKLMNOPABCDEFGHYZabcdefQRSTUVWXopqrs tuvghijklmnwxyz`
S09: `QRSTUVWXYZabcdefABCDEFGHIJKLMNOPwxyzg hijklmnopqrstuv`
S10: `ghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ RSTUVWXYZabcdef`

The first string, S01, can be considered the base element of the set. The second string replaces the second half of S01 with its own first half. String S03 swaps the two blocks that form S01. In addition to the previous change, string S04 replaces the first half of the string with digits. Strings S05 to S10 take as basis the first string and perform transpositions of blocks whose size is 1, 2, 4, 8, 16, and 32 characters, respectively.

The results generated when comparing these special signatures are included in Tables 3, 4, and 5. Table 3 shows the results obtained when using `ssdeep` with signature files whose content replies the strings of the tests. As the limitation imposed by `ssdeep` regarding the minimum length for the common substrings produces as a result that several comparisons are not effectively performed (`ssdeep` directly assigns a score of 0 in those cases), we have implemented the logic of `ssdeep`'s algorithm in Java Standard Edition [12] and have removed that limitation in our code. Thus, Table 4 shows the results that `ssdeep` would provide if it did not apply the aforementioned minimum length requirement. Finally, Table 5 displays the results obtained with our proposed algorithm once implemented as another Java application.

As it can be observed, our algorithm is able to provide meaningful results in all the comparisons, which is not the case in `ssdeep`. For example, the comparison between S01 and S07, which renders a score of 0 in `ssdeep`, is evaluated as having a similarity degree of 77% by our algorithm. Following that example, the modified version of `ssdeep` without the minimum length requirement generates a score of 55 which, even representing a better result, it still fails to properly reflect the fact that S01 and S07 share far more than half of their content.

When inspecting the tables, it can be stated that the results provided by our algorithm are more realistic according to the similarity definition given in the Introduction. For instance, when comparing S01 to S03 and S04, it is clear that S03 is almost the same string as S01, whilst S04 only shares with S01 half of its string. However, `ssdeep` is not able to detect that difference and assigns a value of 50% in both cases. In comparison, our algorithm computes the similarity degree as 97% and 49%, respectively.

Even though the modified version of `ssdeep` provides higher results than our algorithm in some instances (e.g., when comparing S02 and S05 or S04 and S06), those differences are small and do not imply a representative difference. However, when our algorithm provides higher results the difference in some instances is quite important (e.g., when processing S08 and S09). In fact, the average difference in the scores of the test when comparing different strings is 18.96 in favour of our method. We are aware that, in general, a higher value should not imply a better result; however, when comparing the test strings, which clearly share an important part of their contents, a higher result implies a better similarity detection capability.

	S01	S02	S03	S04	S05	S06	S07	S08	S09	S10
S01	100	50	50	50	0	0	0	55	63	63
S02	50	100	50	50	0	0	0	47	50	50
S03	50	50	100	50	0	0	0	36	44	90
S04	50	50	50	100	0	0	0	32	32	50
S05	0	0	0	0	100	0	0	0	0	0
S06	0	0	0	0	0	100	0	0	0	0
S07	0	0	0	0	0	0	100	0	0	0
S08	55	47	36	32	0	0	0	100	32	32
S09	63	50	44	32	0	0	0	32	100	32
S10	63	50	90	50	0	0	0	32	32	100

Table 3: Test results for special cases with `ssdeep`.

	S01	S02	S03	S04	S05	S06	S07	S08	S09	S10
S01	100	50	50	50	50	50	55	55	63	63
S02	50	100	50	50	29	32	36	47	50	50
S03	50	50	100	50	25	29	32	36	44	90
S04	50	50	50	100	25	29	29	32	32	50
S05	50	29	25	25	100	25	29	29	32	32
S06	50	32	29	29	25	100	29	29	32	32
S07	55	36	32	29	29	29	100	32	32	32
S08	55	47	36	32	29	29	32	100	32	32
S09	63	50	44	32	32	32	32	32	100	32
S10	63	50	90	50	32	32	32	32	32	100

Table 4: Test results for special cases with modified version of `ssdeep`.

	S01	S02	S03	S04	S05	S06	S07	S08	S09	S10
S01	100	50	97	49	50	50	77	89	93	97
S02	50	100	49	49	25	25	37	43	47	49
S03	97	49	100	50	50	50	74	85	91	97
S04	49	49	50	100	25	25	37	43	47	49
S05	50	25	50	25	100	50	50	50	50	50
S06	50	25	50	25	50	100	50	50	50	50
S07	77	37	74	37	50	50	100	77	79	75
S08	89	43	85	43	50	50	77	100	87	87
S09	93	47	91	47	50	50	79	87	100	93
S10	97	49	97	49	50	50	75	87	93	100

Table 5: Tests results for special cases with our algorithm.

5. Conclusions

In this contribution we have presented a new edit distance algorithm that can be used in fuzzy hashing applications. Our algorithm provides better results than `ssdeep`'s algorithm according to a definition of similarity useful in computer forensics when comparing two files, and that interprets similarity as the percentage of a file that is also present in another file. We have implemented both our algorithm and a modified version of `ssdeep` in Java, and have used those two applications together with version 2.12 of `ssdeep` in order to test some strings that could represent the signature of files including a list of elements.

The tests performed with the three applications allow us to state that our algorithm provides results better adapted to the aforementioned definition of similarity, so it can be considered as an alternative for the edit distance currently implemented in `ssdeep` and other fuzzy hashing applications.

Acknowledgment

This work has been partially supported by Comunidad de Madrid (Spain) under the project S2013/ICE-3095-CM (CIBERDINE) and by Ministerio de Economía y Competitividad (Spain) under the grant TIN2014-55325-C2-1-R (ProCriCiS).

References

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

[2] V. Gayoso Martínez, F. Hernández Álvarez, and L. Hernández Encinas, “State of the art in similarity preserving hashing functions,” in *Proc. of WorldComp 2014 - International Conference on Security & Management - SAM’14*, 2014, pp. 139–145.

[3] A. Tridgell. (2014) Fuzzy hashing and ssdeep. [Online]. Available: http://ssdeep.sourceforge.net/

[4] 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.

[5] 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

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

[7] 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.

[8] J. Kornblum, “Identifying almost identical files using context trigger piecewise hashing,” *Digital Investigation*, vol. 3(S1), pp. 91–97, 2006.

[9] A. Tridgell, “Efficient algorithms for sorting and synchronization,” Master’s thesis, The Australian National University. Department of Computer Science, Canberra, Australia, 1999.

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

[11] Wikipedia. (2014) Damerau-Levenshtein distance. [Online]. Available: http://en.wikipedia.org/wiki/Damerau-Levenshtein_distance

[12] Oracle. (2015) Java SE. [Online]. Available: http://www.oracle.com/technetwork/java/javase/overview/index.html