THE JOURNAL OF DIGITAL FORENSICS, SECURITY AND LAW

Journal of Digital Forensics, Security and Law

Volume 11 | Number 2 Article 2

2016

Security Analysis of MVhash-B Similarity Hashing

Donghoon Chang
Indraprastha Institute of Information Technology Delhi

Somitra Sanadhya
Indraprastha Institute of Information Technology Delhi

Monika Singh
Indraprastha Institute of Information Technology Delhi

Recommended Citation

Chang, Donghoon; Sanadhya, Somitra; and Singh, Monika (2016) "Security Analysis of MVhash-B Similarity Hashing," *Journal of Digital Forensics, Security and Law*: Vol. 11 : No. 2 , Article 2.
DOI: https://doi.org/10.15394/jdfsl.2016.1376
Available at: https://commons.erau.edu/jdfsl/vol11/iss2/2

SECURITY ANALYSIS OF MVHASH-B SIMILARITY HASHING

Donghoon Chang, Somitra Kr. Sanadhya, Monika Singh
Indraprastha Institute of Information Technology Delhi (IIIT-D), India
{donghoon,somitra,monikas}@iiitd.ac.in

ABSTRACT

In the era of big data, the volume of digital data is increasing rapidly, causing new challenges for investigators to examine the same in a reasonable amount of time. A major requirement of modern forensic investigation is the ability to perform automatic filtering of correlated data, and thereby reducing and focusing the manual effort of the investigator. Approximate matching is a technique to find "closeness" between two digital artifacts. mvHash-B is a well-known approximate matching scheme used for finding similarity between two digital objects and produces a 'score of similarity' on a scale of 0 to 100; however, no security analysis of mvHash-B is available in the literature. In this work, we perform the first academic security analysis of this algorithm and show that it is possible for an attacker to "fool" it by causing the similarity score to be close to zero even when the objects are very similar. By similarity of the objects, we mean semantic similarity for text and visual match for images.

The designers of mvHash-B had claimed that the scheme is secure against 'active manipulation.' We contest this claim in this work. We propose an algorithm that starts with a given document and produces another one of the same size without influencing its semantic and visual meaning (for text and image files, respectively) but which has low similarity score as measured by mvHash-B. In our experiments, we show that the similarity score can be reduced from 100 to less than 6 for text and image documents. We performed experiments with 50 text files and 200 images and the average similarity score between the original file and the file produced by our algorithm was found to be 4 for text files and 6 for image files. In fact, if the original file size is small then the similarity score between the two files was close to 0, almost always.

To improve the security of mvHash-B against active adversaries, we propose a modification in the scheme. We show that the modification prevents the attack we describe in this work.

Keywords: Fuzzy hashing, Approximate Matching, Blacklisting, Whitelisting, Anti-Blacklisting, Similarity digest

1. INTRODUCTION

In today's digital world, everything has been turning digital: conventional books have been replaced by ebooks, letters have been replaced by emails, paper photographs have been replaced by digital image and compact audio and video cassettes have been replaced by mp3 and mp4 CD/DVDs. This transformation has influenced the capacity of storage media that has changed from a few megabytes to some terabytes, which is an enormous volume of data for a forensic investigator to manually examine in a reasonable period of time. The essential requirement of modern forensic investigations is the ability to perform automatic filtering of the relevant data that an investigator needs to examine manually.

Due to the decreasing costs per Gigabytes and ever increasing capacity of storage media, on a crime scene, an investigator is confronted with several terabytes of suspected data. Manual forensic investigation of the enormous volume of data is hard to complete in a practicable amount of time. Therefore, it may be helpful for an investigator to reduce the data under investigation by eliminating similar files from the suspect's hard disk. On the other hand, in some situations, the investigator might be interested in looking only at files similar to a given file in order to find a suspected malicious file or investigate modifications to that file.

Nowadays, most of the forensic toolkits provide the functionality of automatic filtering by finding 'similarity' between files. Automatic filtering is normally done by measuring the amount of correlation between files. However, correlation method does not work well if the adversary deliberately modifies the file without influencing the semantics of file in such a manner that the correlation value becomes very low. For example, a C program can be modified by changing the names of variables, adding comments, writing looping constructs in a different way etc. Ideally, an investigator needs to find all the files which are similar to the desired (malicious) file along with their corresponding percentage similarity to the desired file. The traditional cryptographic hash functions do not work in this situation as even a single bit change in the file content is expected to modify the entire digest randomly. Hence, Cryptographic Hash Functions can be used to find exact duplicates, not similar files.

'Approximate Matching' is a generic technique for finding similarity among given files, typically by assigning a 'similarity score.' An approximate matching technique can be characterized into one of the following categories: Bytewise Matching, Syntactic Matching, and Semantic Matching (Breitinger, Guttman, McCarrin, & Roussev, 2014). Bytewise Matching relies on the byte sequence of the digital object without considering the internal structure of the data object. These techniques are known as fuzzy hashing or similarity hashing. Syntactic Matching relies on the internal structure of the data object. Semantic Matching relies on the contextual attributes of the digital objects which are more closely related to human perception. It is also called Perceptual Hashing or Robust Hashing.

Nowadays, approximate matching algorithms are used to filter in/out the files based on a reference data set. The investigator can generate the similarity digest of all the files on the target device and compare it with reference data set. Based on the dataset, files can be filtered in or out from the process of investigation. The reference datasets are broadly categorized in following two categories: 1) Known-to-be-good 2) Known-to-be-bad. Based on the reference dataset filtering process is of following two types:

- **Blacklisting:** In this process, files are filtered by matching them with the set of already Known-to-be-bad files. The resultant files after this process are the ones which an investigator needs to examine closely.

- **Whitelisting:** In this process, filtering is done by matching the files with a database of already Known-to-be-good files. The files passing this process need not be examined by the investigator.

mvHash-B proposed by Breitinger et al. (Breitinger, Astebol, Baier, & Busch, 2013) in 2013, is one of the most well known fuzzy hashing schemes. The runtime complexity of the mvHash-B scheme is almost equivalent to cryptographic hash function SHA-1, which makes it fastest among the existing approximate matching schemes. Moreover, the length of the mvHash-B similarity digest is just 0.5% of the input length. Both of the above desirable features make it one of the most prominent approximate matching scheme.

(Breitinger et al., 2013) also claims that mvHash-B is sufficiently robust against active manipulations. In this work, we propose an anti-forensic attack on mvHash-B similarity hashing. We develop an algorithm that can be used to circumvent the blacklisting based filtering of the mvHash-B scheme, i.e. it is possible to hide a malicious file from the blacklisting process of mvHash-B similarity hashing. This work shows that less than 0.03 % deliberate modifications can take down the similarity score of a file from 100 to less than 6 without influencing the file semantically and visually.

Our attack can also be used to carry out an anti-forensic mechanism that defeats the very purpose of the approximate matching scheme by hiding a malicious file from the filtering process. An attacker could modify the desired file without changing its semantics and visual meaning. When an investigator tries to filter the desired malicious file using mvHash-B similarity preserving hashing from the hard disk of a suspect, it will not appear in the filtered output.

Finally, we also propose an improvement to the mvHash-B construction in order to prevent our attack. The minor tweak we propose to the scheme ensures the security of the modified mvHash against an active adversary.

The rest of the paper is organized as follows: We discuss related literature in § 2. Notations and definitions used in the paper are provided in § 3. The mvHash-B scheme is explained in § 4 . § 5 contains our analysis and attack on mvHash-B, followed by our proposed attack against the scheme. Experimental results on text and image files validating our attack are presented in § 6. Finally, we conclude the paper in § 7 and § 8 by proposing solutions to mitigate our attack on mvHash-B.

2. RELATED WORK

The first approximate matching technique was proposed in the year 2002 by Nicholas Harbour (Harbour, 2002) called *dcfldd*. It divides an input into fixed-size blocks, hashes each block separately and concatenates all hash values. dcfldd is also known as block based hashing. Security of this scheme can easily bypass by inserting / removing one byte in the beginning (Divakaran, 2008). Context Triggered Piecewise Hashing (CTPH) was proposed by Kornblum (Kornblum, 2006) in his tool named *ssdeep*. CTPH can be considered as an improvement of dcfldd. It overcomes the weakness of dcfldd. The CTPH scheme is based on the spamsum algorithm proposed by Andrew et al.(Tridgell, 2002) for spam email detection. The ssdeep tool computes a digest of the given file by first dividing the file into several chunks and then by concatenating the least significant 6-bits of the hash value of each chunk. A hash function named FNV is used to compute the hash of each chunk. Some improvement to ssdeep scheme was proposed by Chen et al.(Chen & Wang, 2008) and Seo et al.(Seo, Lim, Choi, Chang, & Lee, 2009) in terms of efficiency and security. Thorough security analysis of ssdeep is done by Baier et al.(Baier & Breitinger, 2011) and showed that ssdeep sceme does not withstand an active adversary for blacklisting and whitelisting.

In year 2010, Roussev et al.(Roussev, 2009, 2010) proposed a new approximate matching scheme called *sdhash*. The basic idea of sdhash scheme is to identify statistically improbable features of document based on the entropy analysis of consecutive 64 byte sequence of file data (which is called a 'feature'), which then be used generate the final hash digest of the file. Breitinger et al.(Breitinger & Baier, 2012b) showed some weaknesses in sdhash and presented improvements to the scheme. Detailed security and implementation analysis of sdhash are done in (Breitinger, Baier, & Beckingham, 2012) by the same authors. This work uncovered several implementation bugs and showed that it is possible to beat the similarity score by tampering a given file without changing the perceptual behavior of this file (e.g. image files look almost same despite the tampering). Then claims of (Breitinger et al., 2012) is again verified by chang et al. in (Chang, Sanadhya, Singh, & Verma, 2015). They also proposed an attack method which can mislead the investigator with many forged similar files.

In year 2012 Breitinger et al. proposed a new scheme called *bbHash* (Breitinger & Baier, 2012a). bbHash aims two properties of the approximate matching algorithm: 1) Coverage: every byte of input expected to be involved in final hash digest generation. Thus, every byte should influence the hash digest. 2) Variable sized length: unlike the traditional hash bbHash ensures that the hash digest length is proportional to the size of the input. The run time of bbHash is too high so it is practically not usable.

mvHash-B similarity preserving hashing was proposed by Breitinger et al. in year 2013 (Breitinger et al., 2013). The basic idea of mvHash-B algorithm is to first compress the input data using majority voting and run length encoding then represent the final fingerprint into Bloom filters. 'B' in mvHash-B symbolizes the bloom filter representation of similarity digest.

3. NOTATIONS

- BS denotes input byte sequence i.e. BS= $B_0B_1B_2.....B_{L-1}$ where B_i represents the ith byte of input and L denotes the length of the input file in bytes.
- $N_{k,n}$ denotes the n-neighborhood of input byte B_k.
  $N_{k,n} = B_{k-\frac{n}{2}}B_{k-\frac{n}{2}+1}....B_{k-1}B_kB_{k+1}.....B_{k+\frac{n}{2}-1}B_{k+\frac{n}{2}}$ where n denotes size of neighborhood and n is always even.
- bitcount($N_{k,n}$) denotes the function that outputs number of bits set to 1 in binary representation of $N_{k,n}$
- t denotes the threshold
- ib denotes average number of influencing bits for one byte.

4. DESCRIPTION OF MVHASH-B

mvHash-B works in following three phases:

1. **Majority Votes:** The idea of this phase is to convert each input byte into 0x00 or 0xFF in order to compress the input in subsequent phases. mvHash-B counts number of bits set to one for n-neighboring of each input byte i.e. bitcount($N_{k,n}$) for $0 \leqslant k \leqslant (L-1)$. If bitcount($N_{k,n}$)$\leqslant$ t(threshold), then the value majority vote of the byte B_k is set to 0xFF else 0x00. The value threshold t is calculated as follows:

   $$t = \frac{(n+1)\cdot ib}{2}$$

   where ib denotes average number of influencing bits for one byte($0 \leq ib \leq 8$), default value of ib=8.

2. **Encoding the majority vote bit sequence with RLE:** Run length encoding(RLE) is a data compression algorithm. Number of the identical subsequent byte is called 'Sun-count'. The output of run length encoding is the sequence of run-counts as shown in Fig 1 and denoted as RLE. mvHash-B assumes that each RLE starts with number of identical 0x00 bytes, therefore, if the majority vote of input starts with 0xFF then run-length-encoding keeps 0 in the beginning.

3. **Fingerprint generation using Bloom filters:** mvHash-B stores the resultant RLE sequence in bloom filter. The reason behind using bloom filter is its efficient comparison capability. Bloom filter is an array of m elements with all elements initialized to 0. In mvHash-B implementation m=2018(2^{11}). Select first 11 bytes of RLE sequence to built a group and perform mod 2 of each RLE element of this group. Mod 2 operation transforms the RLE sequence into 11 bit sequence of 1 or 0 i.e. $b_0b_1b_2. \ . \ . \ b_{10}$ where $b_i\epsilon$ 0,1 and $0 \leq i \leq 10$. This is further divided into two parts:

   - $v_1 = b_{10}b_9b_8b_7b_6b_5b_4b_3$ is used to identify the byte within the Bloom filter and
   - $v_2 = b_2b_1b_0$ is used to identify the bit within the byte.

   Identified bit position of bloom filter is set to 1. Next group can not be consecutive in order to ensure alignment robustness. Next group starts will the alternate element of RLE sequence. This process is explained in detailed in Fig 1. Same process is applied to till the last element of RLE sequence.

   In order to find similarity between two bloom filters, a distance score is calculated which is represented as di_{score}. di_{score} computation uses the hamming distance. Let bf_1 and bf_2 are two bloom filters, the value of di_{score} is calculated as follows:

$$di_{score} = \frac{hd(bf_1, bf_2)}{|bf_1| + |bf_1|} \cdot 100$$

   where $hd(bf_1, bf_2)$ denotes the hamming distance between bf_1 & bf_2 and $|bf_i|$ denotes the number of bits set to 1 in bloom filter bf_i. Value of di_{score} ranges from 0 to 100, where 0 indicates the exact similarity(100% similarity) and 100 indicates zero similarity.

5. ANTI-BLACKLISTING ATTACK ON MVHASH-B

We present an anti-forensic attack based on mvHash-B blacklisting i.e. we have shown that it is possible to circumvent mvHash-B blacklist filtering. We have provided an algorithm/tool that can be used to avoid the automated detection of the blacklisted or malicious files by the blacklisting process of mvHash-B similarity hashing. Blacklisting is the process of filtering out the files by matching it with the set of already Known-to-be-bad files. The resultant file of the blacklisting process is similar to known-to-be-bad or malicious files and need to be examined manually. The proposed algorithm modifies the target file without changing the semantics of the file in a way so that it does not appear in the list of blacklisted files. These kind of attacks are termed as an 'Anti-Blacklisting attack' in literature by baier et al. (Baier & Breitinger, 2011).

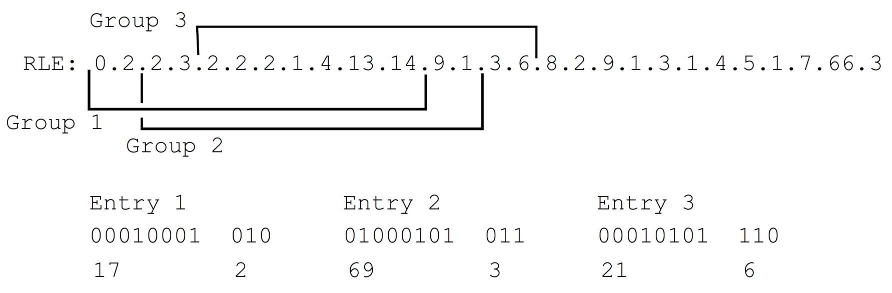

Figure 1. Processing of RLE encoding by mvHash-B from (Breitinger et al., 2013) .

The proposed Anti-Blacklisting attack shows that mvHash-B scheme does not withstand an active adversary against a blacklist. This paper follows the definition of anti-blacklisting provided by Baier et al. in their paper (Baier & Breitinger, 2011). Proposed attack generates false negatives for mvHash-B similarity results. Let F_1 is a malicious file. The malicious user or attacker can use the proposed algorithm and generate file F_2, which is semantically and perceptually same as F_1. However, mvHash-B similarity hashing results F_2 as non-similar to F_1, thus a false negative.

As discussed in § 4, mvHash-B works in three phases 1)Majority Votes 2)Encoding the majority vote bit sequence with RLE 3) Fingerprint generation using Bloom filters, where the first phase majority votes converts each input byte into 0x00 or 0xFF, then the next phase Run length encoding transforms the input into run-count of the identical subsequent bytes (0x00 or 0xFF) and finally the last phase performs a modulo 2 operation on the resultant RLE sequence and stores the resultant RLE sequence in bloom filter as shown in Fig. 1. Each consecutive 11 elements of RLE-sequence forms a group. Each group sets one bit in bloom filter. Fig.2 shows an example.

The key idea of the attack is by modifying one element in each group, we can change the position of all the set bits in bloom filter. Since the bloom filter represents the final mvHash-B digest hence the entire digest is modified. Each element of a group is modulo 2 representation of corresponding RLE elements, therefore, it is either 1 or 0. We need to flip any one bit among all 11 bits of a group. Which requires to increase or decrease any of the corresponding 11 RLE-elements just by 1. Any modification in the RLE sequence requires replacement of the corresponding input byte with the byte containing more or fewer number of 1s (in its binary representation). Now the important question is, how to modify RLE-encoding without influencing the semantics of the file. Algorithm 1 presents a way of making such modifications. Algorithm1 works in following steps:

1. Go through each byte of the input file. Check if the number of bits set to 1 in the n-neighborhood(bitcount($N_{k,n}$)) of the current byte(B_k) is equal to t or (t-1), where k is the current byte index and t is the threshold. If yes, proceed with following steps:
   (a) Modify the B_k with semantically similar character (defined later in this section). Let B'_k denotes modification on B_k and similarly $N'_{k,n}$ denotes modification on $N_{k,n}$. Check for following condition:
      i. bitcount($N'_{k,n}$)<bitcount($N_{k,n}$) and
      bitcount($N_{k,n}$)=t

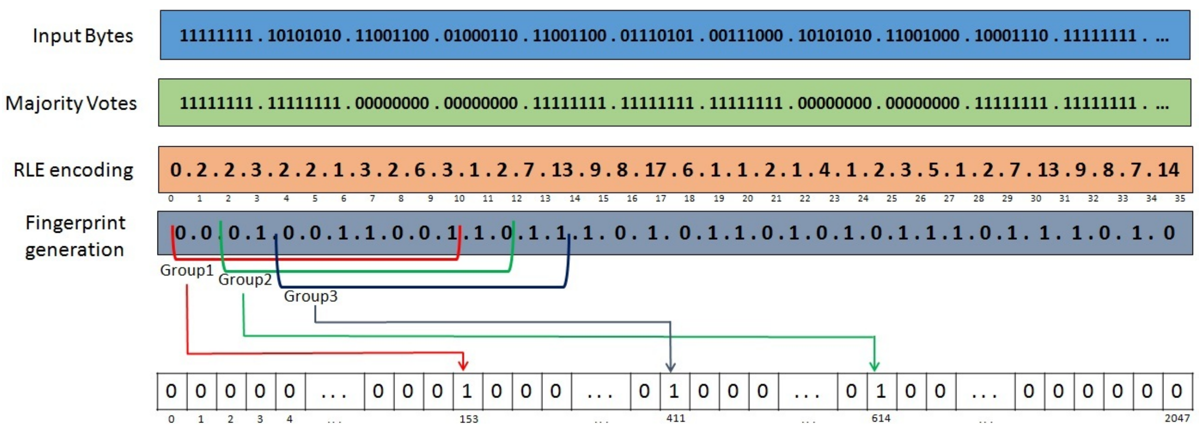

Figure 2. mvHash-B similarity digest generation considering n=2

ii. bitcount($N'_{k,n}$)>bitcount($N_{k,n}$) and bitcount($N_{k,n}$)=(t-1)

If any of the above condition satisfies then accept the modification else revert the changes.

(b) If the modification happens at step (a) it will change the majority vote of the corresponding byte from 0x00 to 0xFF or vice versa.

(c) Any change in majority vote will reflect modification in corresponding RLE element since it is the count of consecutive 0x00 or 0xFF.

(d) Any alteration in RLE sequence will modify the index of bloom filter element addressed by the group containing the modified RLE element. Fig. 3 shows an example of one byte modification. One RLE modification may effect several groups.

2. Perform step 1 after the last byte of the last modified group.

3. Repeat all the above steps till the last byte of the input file.

The semantically and perceptually similar alteration can be performed as follows:

- For text documents:

  1. Lower case to upper case conversion or vice-versa
  2. Space to tab or tab to space, etc

- For the image documents:
  The modification can be performed by doing a minor change in the RGB value of a pixel. For example, in bmp (RGB32) format each pixel in an indexed color image is described by 3 bytes, representing its RGB (Red-Green-Blue) value. This RGB value is the index of single color described by the color table. Altering the least significant bit of any of these three bytes does not produce a visual change in the image.

- For a program file:

  1. changing the names of variables
  2. writing looping constructs in a different way
  3. adding comments, etc.

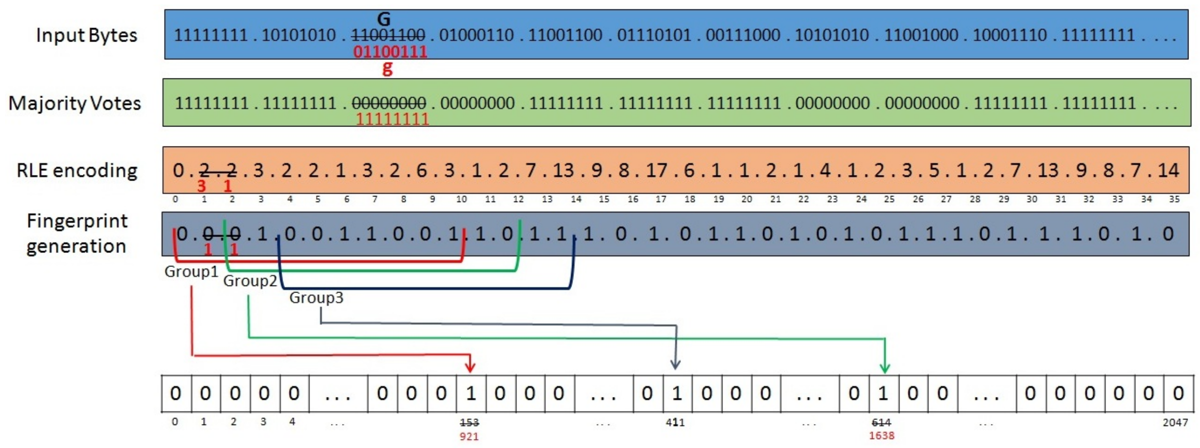

Figure 3. Example of one byte modification

One byte modification in a group is enough, since a modulo 2 operation is performed on the RLE elements, so addition or subtraction of one byte will change the position of the addressed bit in bloom filter. Each group differs from its neighboring group only by two elements; therefore, one modification influences several groups. Fig. 4 contains an example which shows that for a 174 byte long document, just 2 input byte modifications are enough to change the entire mvHash-B digest of the document. The value of n considered in the example explained in Fig. 4 is 2, where as recommended value of n by design of mvHash-B scheme is 50 or 20 depending on the file type. If the value of neighborhood(n) is higher then number of modifications required are smaller. For example if n is 50, then one byte modification will impact majority vote calculation of 50 neighboring bytes.

6. RESULTS

We performed two experiments: one on the textual data(text documents) and other on visual data (images).

6.1 Experiment 1

Our first experiment was performed on a dataset of 50 text files of variable sizes from the T5-corpus dataset[1]. As recommended in (Breitinger et al., 2013), we took parameters n and ib to be 50 and 7, respectively. The results obtained from the experiment show that merely 3% deliberate modification in the file takes down the similarity score from 100 to 4 (on an average), whereas the modified file is semantically similar to the original file. Table 1 shows the experimental results of our proposed anti-blacklisting attack for a small sample of 10 text files.

6.2 Experiment 2

We performed a second set of experiments on a dataset of 200 bitmap images of variable sizes. We took images from various publicly available datasets such as Microsoft Windows Bitmap Sample Files [2], CVonline: Image Databases [3] and Yokogawa Y-Link [4]). The value of input parameter n was chosen to be 50 and ib was taken to be 8. This is as per the suggestion in (Breitinger et al., 2013).

[1]http://roussev.net/t5/t5.html

[2]http://www.fileformat.info/format/bmp/sample/

[3]http://homepages.inf.ed.ac.uk/rbf/CVonline/Imagedbase.htm#segmentatio

[4]https://y-link.yokogawa.com/YL008/?V_ope_type=Show&LANG=EN&Language_id=EN&Login_type=2&Download_id=DL00002164

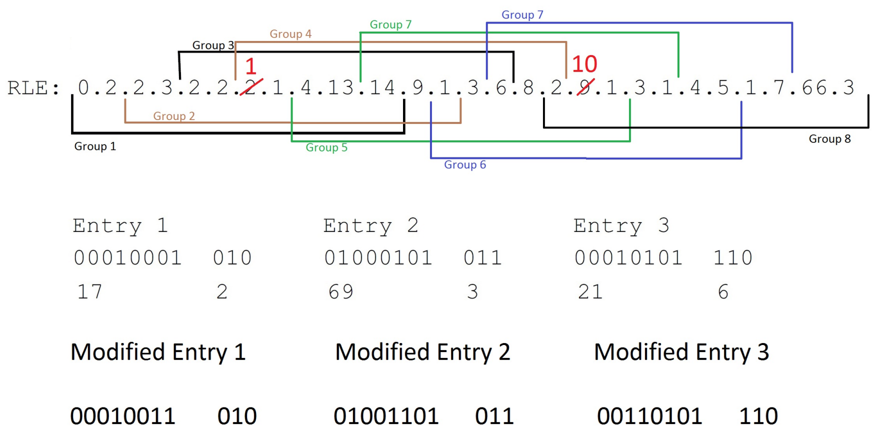
Group 7
Group 7
Group 4
Group 3
1
10
RLE: 0.2.2.3.2.2.2.1.4.13.14.9.1.3.6.8.2.9.1.3.1.4.5.1.7.66.3
Group 2
Group 1
Group 5
Group 6
Group 8
Entry 1 00010001 010 17 2
Entry 2 01000101 011 69 3
Entry 3 00010101 110 21 6
Modified Entry 1 00010011 010
Modified Entry 2 01001101 011
Modified Entry 3 00110101 110

Figure 4. Example illustrating number of Deliberate Modifications required in order perform anti-blacklisting attack

Table 2 illustrates the experimental results of our proposed attack on a sample set of 10 bitmap images. Observed results demonstrate that by varying as little as 0.3% of the original image bytes, the similarity score of mvHash-B digest for the image reduces from 100 to 6 on an average. Moreover, the resultant modified images are visually same as the original image. Fig. 5 shows an image and a similar image obtained by our attack algorithm. The left image is the unmodified source RAY.BMP, taken from Microsoft Windows Bitmap Sample Files [5] while the image on the right is the modified image generated by our algorithm. Visual similarity between these two files can be seen to be very high. However, the mvhash-B similarity score for these images is 0.

An attacker can apply the proposed algorithm on a malicious image and can generate a modified image, which is visually same as the original malicious image but can not be detectable from mvhash-B similarity hashing. Therefore an attacker can easily hide the malicious image/text file from the mvhash-B filtering process defeating the very purpose of approximate matching algorithms.

7. COUNTERMEASURES

In order to prevent the proposed attack, we suggest following improvement in the design of mvHash-B construction. The root cause of the attack is that attacker has the ability to identify the position of input byte that he can modify with maximum influence over the mvHash digest. We want to restrict this liberty by adding two secret input parameter called ‘trigger’ and ‘x’ to the scheme. The investigator can choose any value of these two parameter while mvHash digest calculation.

Let T be the trigger value chosen by the investigator. Fig. 6 explains our suggested improvement to mvHash-B scheme. Before Majority vote calculation rolling hash over the input byte is calculated. The rolling hash calculation is be done in same way as explained by Kornblum in Context Triggered Piecewise Hashing (Kornblum, 2006). The value of rolling hash depends only on last s bytes of the input file. Let R_i denotes the rolling hash of ith input byte.

R_i=Rolling-Hash(B_i,B_{i-1},B_{i-2},. . .,B_{i-s})

[5]http://www.fileformat.info/format/bmp/sample/1d71eff930af4773a836a32229fde106/view.htm

Table 1. Experimental results obtained from the proposed attack technique on Text file

S.No.	File Name	File size (In Kilo Bytes)	Number of Modification (in Kilo Bytes)	Similarity Score
1†	Test_02.text	0.47	0.01	0
2	Test_4955.text	14	0.40	0
3	Test_4950.text	23	0.61	0
4	Test_4954.text	27	0.74	0
5	Test_4956.text	30	0.91	0
6	Test_3518.text	34	0.41	0
7	Test_4960.text	47	1.20	7
8	Test_4953.text	76	2.00	14
9	Test_4953.text	189	6.00	9
10	Test_4953.text	190	4.00	10
On an avg.			2.59%	4

† This file is not from T5-corpus database

Table 2. Experimental results obtained from the proposed attack technique on bitmap images

S.No.	File Name	File size (In Kilo Bytes)	Number of Modification (in Kilo Bytes)	Similarity Score
1	Air Conditioner S.bmp	6	0.01	0
2	Pressure Transmitter 03 M.bmp	20	0.45	0
3	Control Valve L.bmp	55	0.028	0
4	DadWood.bmp	265	0.480	0
5	Edison.bmp	500	1.838	8
6	carsgraz_287.bmp	901	2.383	10
7	Ray.bmp	1407	2.14	0
8	Alex bit.bmp	3984	16.682	13
9	MARBLES.bmp	4165	13.424	16
10	12x12 Women sample 400 dpi.bmp	22502	42.244	16
On an avg.			0.228%	6.3

† This file is not from T5-corpus database

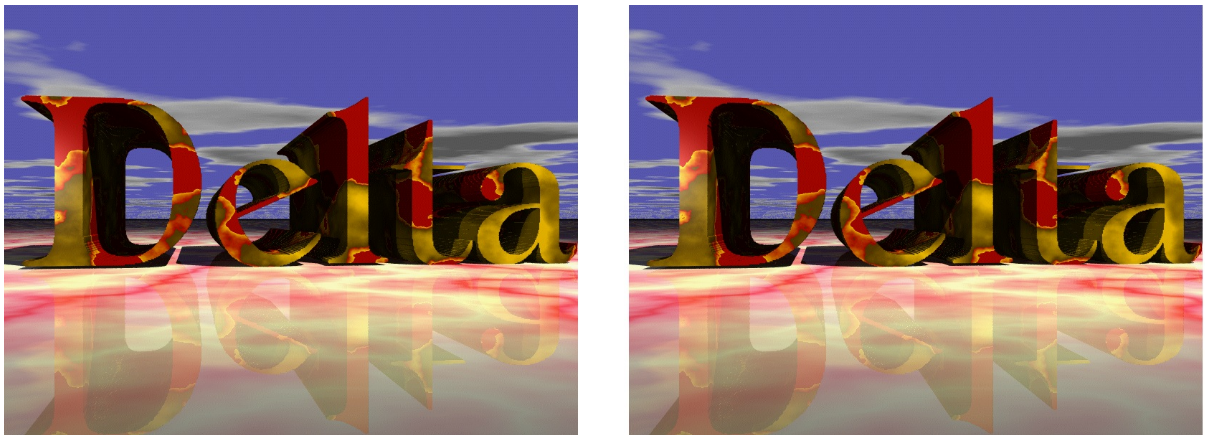
Figure 5. Example: First image from the left is original image taken from Microsoft Windows Bitmap Sample Files[6] and the other image is the generated modified image from the proposed algorithm; mvHash-B similarity score of above images is 0

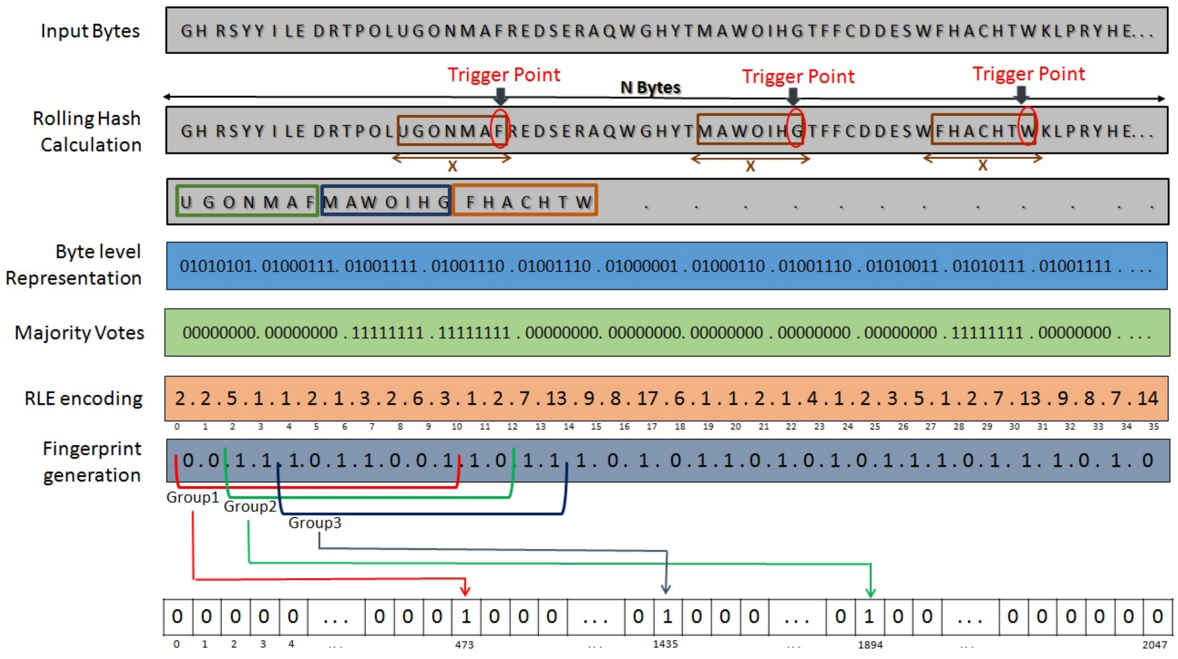

Figure 6. Counter measure for the proposed attack

where B_i represents the i^{th} input byte. If R_i== -1 mod T for s≤i< n where n denotes input size. i^{th} byte position is called trigger point. Select last x bytes from the trigger point for further digest calculation as shown in Fig. 6. The number of trigger points is inversely proportional to T (Baier & Breitinger, 2011). Thus, for the higher value of T number of trigger points will be smaller or vice-versa. An investigator can choose the value of T and x based on input file size and these values are unknown to the attacker.

The attacker does not know selected input byte. Therefore, cannot perform the deliberate intelligent modification. Random modification also does not impact the mvHash-B digest much because every input byte is not taking part in final digest calculation. The attacker is unaware of trigger positions, and thus cannot perform random insertion/deletion as well.

8. CONCLUSION

This work explores the weakness of mvHash-B similarity digest scheme, which can be exploited by an active adversary to defeat the purpose of the approximate matching scheme. We show an Anti-Blacklisting attack on mvHash-B similarity digest and practically prove that mvHash-B does not withstand an active adversary against blacklist. We provide an anti-forensic tool that can be used by an adversary to bypass the blacklist filtering process of mvHash-B similarity digest. Additionally, we suggested an improvement to mvHash design to conquer the proposed attack. Furthermore, the proposed improvement ensures the security of a scheme against an active adversary.

9. ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful suggestions and feedback. We thank our colleagues Surabhi Garg, Robin Verma, Sweta Mistra, and Jyoti Leeka, who provided insight and fruitful suggestions during the course of this work. Monika Singh was supported by TCS Research and Development PhD fellowship.

REFERENCES

Baier, H., & Breitinger, F. (2011). Security aspects of piecewise hashing in computer forensics. In H. Morgenstern et al. (Eds.), *Sixth international conference on IT security incident management and IT forensics, IMF 2011, stuttgart, germany, may 10-12, 2011* (pp. 21–36). IEEE Computer Society. Retrieved from `http://dx.doi.org/10.1109/IMF.2011.16` doi: 10.1109/IMF.2011.16

Breitinger, F., Astebol, K. P., Baier, H., & Busch, C. (2013). mvhash-b - A new approach for similarity preserving hashing. In *Seventh international conference on IT security incident management and IT forensics, IMF 2013, nuremberg, germany, march 12-14, 2013* (pp. 33–44).

Breitinger, F., & Baier, H. (2012a). A fuzzy hashing approach based on random sequences and hamming distance. In *Proceedings of the conference on digital forensics, security and law* (pp. 89–100).

Breitinger, F., & Baier, H. (2012b). Properties of a similarity preserving hash function and their realization in sdhash. In *2012 information security for south africa, balalaika hotel, sandton, johannesburg, south africa, august 15-17, 2012* (pp. 1–8). Retrieved from `http://dx.doi.org/10.1109/ISSA.2012.6320445` doi: 10.1109/ISSA.2012.6320445

Breitinger, F., Baier, H., & Beckingham, J. (2012). Security and implementation analysis of the similarity digest sdhash. In *First international baltic conference on network security & forensics (nesefo)*.

Breitinger, F., Guttman, B., McCarrin, M., & Roussev, V. (2014). Approximate matching: definition and terminology. *URL http://csrc. nist. gov/publications/drafts/800-*

168/sp800_168_draft. pdf.

Chang, D., Sanadhya, S. K., Singh, M., & Verma, R. (2015). A collision attack on sdhash similarity hashing. In *Proceedings of 10th intl. conference on systematic approaches to digital forensic engineering* (pp. 36–46).

Chen, L., & Wang, G. (2008). An efficient piecewise hashing method for computer forensics. In *Proceedings of the international workshop on knowledge discovery and data mining, WKDD 2008, adelaide, australia, 23-24 january 2008* (pp. 635–638). IEEE Computer Society. Retrieved from `http://dx.doi.org/10.1109/WKDD.2008.80` doi: 10.1109/WKDD.2008.80

Divakaran, A. (2008). *Multimedia content analysis: Theory and applications* (1st ed.). Springer Publishing Company, Incorporated.

Harbour, N. (2002). *Dcfldd. defense computer forensics lab.*

Kornblum, J. D. (2006). Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, *3*(Supplement-1), 91–97. Retrieved from `http://dx.doi.org/10.1016/j.diin.2006.06.015` doi: 10.1016/j.diin.2006.06.015

Roussev, V. (2009). Building a better similarity trap with statistically improbable features. In *42st hawaii international international conference on systems science (HICSS-42 2009), proceedings (CD-ROM and online), 5-8 january 2009, waikoloa, big island, hi, USA* (pp. 1–10). IEEE Computer Society. Retrieved from `http://dx.doi.org/10.1109/HICSS.2009.97` doi: 10.1109/HICSS.2009.97

Roussev, V. (2010). Data fingerprinting with similarity digests. In K. Chow & S. Shenoi (Eds.), *Advances in digital forensics VI - sixth IFIP WG 11.9 international conference on digital forensics, hong kong, china, january 4-6, 2010, revised selected papers* (Vol. 337, pp. 207–226). Springer. Retrieved from `http://dx.doi.org/10.1007/978-3-642-15506-2_15` doi: 10.1007/978-3-642-15506-2_15

Seo, K., Lim, K., Choi, J., Chang, K., & Lee, S. (2009, 12). Detecting similar files based on hash and statistical analysis for digital forensic investigation. In *Proceedings of the 2009 2nd international conference on computer science and its applications, csa 2009.* doi: 10.1109/CSA.2009.5404198

Tridgell, A. (2002). *Spamsum readme.* Retrieved from `https://www.samba.org/ftp/unpacked/junkcode/spamsum/README`

Algorithm 1

```
*input    ▷ Pointer to the input file
input_size    ▷ Size of input file in Bytes
n    ▷ Size of neighhood (Default value for text files is 20)
ib    ▷ ib denotes average number of influencing bits for one byte($0 \leq$ib$\leq 8$), default value of ib=8.
bitcount_$n$(k)    ▷ Function that outputs number of bits set to 1 in n-neighborhood of kth byte of input
bits[255]    ▷ Array containing number of bits set to one in all ASCII characters
*output    ▷ Pointer to the modified resultant file
rle_index = 0;    ▷ Temporary variable containing the current index position of RLE sequence
input_index = 0;    ▷ Temporary variable containing the index position of current input byte
t = 0;    ▷ Threshold
modification = ‘Y’    ▷ Temporary variable initialised with ’Y’
tmp, tmp_rle    ▷ Temporary variables
reduce_bitcount(input[k])    ▷ Function that modifies input byte and write it to the resultant file with simantically
    similar character if that modification reduces the bitcount_$n$(kk) and returns ’Y’ else ’N’
increase_bitcount(input[k])    ▷ Function that modifies input byte and write it to the resultant file with simantically
    similar character if that modification increases the bitcount_$n$(kk) and returns ’Y’ else ’N’
$t = ((n + 1) * ib)/2$    ▷ Calculate the threshold value
for $kk \leftarrow 0$ to $input\_size - 1$ do    ▷ Run through each byte of input file
    if $tmp\_rle == t$ then
        tmp_rle ← 0
        modification ← Y
        tmp_rle ← bitcount_$N$(kk)
        modification ← reduce_bitcount(input[kk])    ▷ Try to reduce the bitcount of the byte
                                                       without changing semntic value
        if $modification ==$ ‘$Y'$ then    ▷ If modification of kk$^{th}$ byte is possible
            end_fl ← 0;
            while input_index ≤ kk do    ▷ Get the RLE index of that byte
                input_index = input_index + output[rle_index]; rle_index++;
            end while
            if input_index<input_size then end_fl = input_index;
            elseend_fl = input_size
            end if
            if (kk + 1) ≤ input_size then    ▷ Get the index of last byte of the group involved the modified
                                               byte
                for $i \leftarrow (kk + 1)$ to $end\_fl$ do fputc(input[i], output)
                end forkk = end_fl;    ▷ Next modification will be perform after end_fl$^{th}$ byte of input
            end if
        end if
    else if tmp_rle == (t - 1) then tmp = input[kk]; modification = increase_bitcount(input[kk])
        ▷ Try to increase the bitcount of the byte with changing semntic value
        if modification == ‘Y’ then
            while input_index ≤ kk do    ▷ Get the RLE index of that byte
                input_index = input_index + output[rle_index]; rle_index++;
            end while
            end_fl ← 0;
            if (input_index)<input_size then end_fl = input_index
            else end_fl = input_size
            end if
            if (kk + 1) ≤ input_size then    ▷ Get index of last byte of the group involved, the modified byte
                for $j \leftarrow (kk + 1)$ to $end\_fl$ do fputc(input[j], output)
                end forkk = end_fl;    ▷ Next modification will be perform after end_fl$^{th}$ byte of input
            end if
        end if ch_value = (char)tmp;
    else fputc(input[kk], output)
    end if
end for
```