A COLLISION ATTACK ON SDHASH SIMILARITY HASHING

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

ABSTRACT

Digital forensic investigators can take advantage of tools and techniques that have the capability of finding similar files out of thousands of files up for investigation in a particular case. Finding similar files could significantly reduce the volume of data that needs to be investigated. Sdhash is a well-known fuzzy hashing scheme used for finding similarity among files. This digest produces a 'score of similarity' on a scale of 0 to 100. In a prior analysis of sdhash, Breitinger et al. claimed that 20% contents of a file can be modified without influencing the final sdhash digest of that file. They suggested that the file can be modified in certain regions, termed 'gaps', and yet the sdhash digest will remain unchanged. In this work, we show that their claim is not entirely correct. In particular, we show that even if 2% of the file contents in the gaps are changed randomly, then the sdhash gets changed with probability close to 1. We then provide an algorithm to modify the file contents within the gaps such that the sdhash remains unchanged even when the modifications are about 12% of the gap size. On the attack side, the proposed algorithm can deterministically produce collisions by generating many different files corresponding to a given file with maximal similarity score of 100.

Keywords: Fuzzy hashing, similarity digest, collision, anti-forensics.

1. INTRODUCTION

The modern world has been turning increasingly digital: conventional books have been replaced by ebooks, letters have been replaced by emails, paper photographs have been replaced by digital image and compact audio and video cassettes have been replaced by mp3 and mp4 CD/DVD's. Due to the reducing costs of storage devices and their ever increasing size, people tend to store several (maybe, slightly different) versions of a file. In case a person is suspected of some illegal activity, security agencies typically seize their digital devices for investigation. Manual forensic investigation of enormous volume of data is hard to complete in a reasonable 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 investigate modifications to that file.

Most forensic software packages contain tools which check for '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 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, writing looping constructs in a different way, adding comments etc. Ideally, an investigator would like to efficiently know the percentage change in two versions of a file so that he can concentrate on files which are slightly different from a desired file. Using Cryptographic Hash Function (CHF) as a digest of the file does not work in this situation as even a single bit change in the file content

is expected to modify the entire digest randomly by the application of a CHF.

'Approximate Matching' is a 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. It is also called Perceptual Hashing or Robust Hashing. Semantic Matching relies on the contextual attributes of the digital objects. Sdhash, proposed by Roussev (Roussev, 2010a) in 2010, is one of the most widely used fuzzy hashing schemes. It is used as a third party module in the popular forensic toolkit 'Autospy/Slueuth-kit' [1] and in another toolkit 'BitCurator' [2].

Breitinger et al. analyzed sdhash in (Breitinger, Baier, & Beckingham, 2012; Breitinger & Baier, 2012) and commented that "approximately 20% of the input bytes do not influence the similarity digest. Thus it is possible to do undiscovered modifications within gaps". In this work, we show that this claim is not entirely correct. We show that if data between the 'gaps' is randomly modified then the digest changes even when the modifications are only about 2% of the 'gap size'. After that we propose an algorithm which can generate multiple files having sdhash similarity score of 100 corresponding to a given file, by modifying upto 12% of the 'gap size'. The proposed algorithm can also be used to carry out an anti-forensic mechanism that defeats the purpose of digital forensic investigation by filtering out similar files from a given storage media. An attacker could generate multiple dissimilar files corresponding to a particular file with 100% matching sdhash digest using our technique.

[1] http://wiki.sleuthkit.org/index.php?title=Autopsy_3rd_Party_Modules

[2] http://wiki.bitcurator.net/?title=Software

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 sdhash scheme is explained in § 4 and existing analysis of the scheme is presented in § 5. § 6 contains our analysis and attack on sdhash, followed by our proposed algorithm . Finally, we conclude the paper in § 7 and § 8 by proposing solutions to mitigate our attack on sdhash.

2. RELATED WORK

The first fuzzy hashing technique, Context Triggered Piecewise Hashing (CTPH) was proposed by Kornblum (Kornblum, 2006) in his tool named **ssdeep**. 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.

Chen et al. (Chen & Wang, 2008) and Seo et al. (Seo, Lim, Choi, Chang, & Lee, 2009) proposed some modifications to ssdeep to improve its efficiency and security. Baier et al. (Baier & Breitinger, 2011) presented thorough security analysis of ssdeep and showed that it does not withstand an active adversary for blacklisting and whitelisting.

Roussev et al. (Roussev, 2009, 2010a) proposed a new fuzzy hashing scheme called sdhash. The basic idea of **sdhash** scheme is to identify statistically improbable features based on the entropy of consecutive 64 byte sequence of file data (which is called a 'feature') in order to generate the final hash digest of the file. Breitinger et al. (Breitinger & Baier, 2012) showed some weaknesses in sdhash and presented improvements to the scheme. Detailed security and implementation analysis of sdhash was done in (Breitinger et al., 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).

3. NOTATIONS

Following notations are used throughout this work:

- D denotes the input data object of N bytes, D = $B_0B_1B_2.....B_N$, where B_i is the i^{th} byte of D .
- f_k is a L byte subsequence of consecutive bytes of data object D. It is termed the kth 'feature' of the data object. In the sdhash implementation, L = 64, f_k = $B_{k+0}B_{k+1}B_{k+2}.....B_{k+63}$ where 0≤ k< n and n is the total number of features of the data object D. Thus n = N - L + 1.
- H(X) represents the entropy of random variable X.
- H_{max}(X) represents the maximum entropy of random variable X.
- H_{min}(X) represents the minimum entropy of random variable X.
- H_{norm}(X) denotes the normalized entropy of random variable X.
- nbf_k denotes the next byte of feature f_k of data object D.
- $R_{prec,D}(f_k)$ denotes the precedence rank of feature f_k of data object D.
- $R_{pop,D}(f_k)$ denotes the popularity score of feature f_k of data object D.
- bf denotes the bloom filter of 256 bytes.
- $\overline{bf}$ represents the number of features within bloom filter bf.
- $|bf|$ denotes number of bits set to one within the bloom filter bf.
- t denotes some threshold (sdhash uses t = 16).
- $SF_{score}(bf_1,bf_2)$ represents the similarity score of bloom filter bf_1 and bf_2.

4. DESCRIPTION OF SDHASH

We now describe the working of sdash using the notation defined in § 3. Given a data object D of length N bytes ($B_0B_1B_2.....B_{N-1}$), a feature f_k is a subset of L (= 64) consecutive bytes of D, that is f_k:$B_{k+0}B_{k+1}B_{k+2}...B_{k+63}$ where 0≤k<n and n = N-L+1. In order to generate sdhash fingerprint, the first step is to calculate the normalized entropy of each feature. Entropy of a random variable X with probability P_X and alphabet α is defined as

$$H(X) = \sum_{x \in \alpha} (P[X = x] log_2 P[X = x])$$

The entropy of X attains its maximum value if $P[X = x] = \frac{1}{|\alpha|}, \forall x \in \alpha$; that is, if all possibilities for X are equiprobable. This maximum value of entropy H_{max}(X) is $\log_2|\alpha|$. Similarly, the entropy is minimum if $\exists x \in \alpha : P[X = x]$ = 1; hence H_{min}(X) = 0. Entropy of a random variable ranges between 0 to $\log_2|\alpha|$. Normalized entropy of a random variable X is defined as H_{norm}(X)= $\frac{H(X)}{H_{max}(X)}$. The normalized entropy ranges between 0 to 1. Random variable in the context of a feature (f_k) is the next byte of the feature (f_k), represented as nbf_k. In sdhash implementation, α is the set of all possible 256 values of x. The probability distribution of nbf_k is defined as:

$$\forall x \in \alpha \; P[nbf_k = x] = \frac{|\{j | B_{k+j} = x, 0 \leq j < 64\}|}{64}$$

where f_k = $B_{k+0}B_{k+1}B_{k+2}.....B_{k+63}$. The Entropy of nbf_k is:

$$H(nbf_k) = \sum_{x \in \alpha} (P[nbf_k = x] log_2 P[nbf_k = x])$$

$H_{max}(nbf_k)$= $\log_2|\alpha|$= 8 and $H_{min}(nbf_k)$=0. Normalized entropy of nbf_k is $\frac{H(nbf_k)}{H_{max}(nbf_k)}$ = $\frac{H(nbf_k)}{8}$. Range of normalized entropy of nbf_k is 0 to 1. It is being scaled up to the range 0 to 1000 and represented by $H_{norm}(nbf_k)$:

$$H_{norm}(nbf_k) = \left\lfloor 1000 * \frac{H(nbf_k)}{8} \right\rfloor$$

After calculating the normalized entropy of each feature, a precedence rank is assigned to the respective feature of the data object D based on the empirical observation of probability density function for normalized entropy of experimental data set.

Let Q is the experimental data set of q data objects $D^1D^2D^3.....D^q$ of same type and same size. Here the random variable is normalized entropy of next data object's nbf_k of set Q, represented as nenfd_Q. Let A is a set of integers from 0 to 1000 i.e. 0,1,2,....,1000.

$$\text{For a}\in\text{A } P[nenfd\_Q = a] = \frac{|\{(i,k)|H_{norm}(nbf_k^i)=a, 0\leq k<n, 0\leq i<q\}|}{qn}$$

where q is number of data object in set Q, nbf_k^i is next byte of feature f_k of D^i data object, $0\leq i<q$, $0\leq k<n$. $H_{norm}(nbf_k^i)$ is normalized entropy of nbf_k^i. Each D^i consists n features. A characteristic probability distribution of each type of data object (i.e. doc, html, gz etc.) can be found.

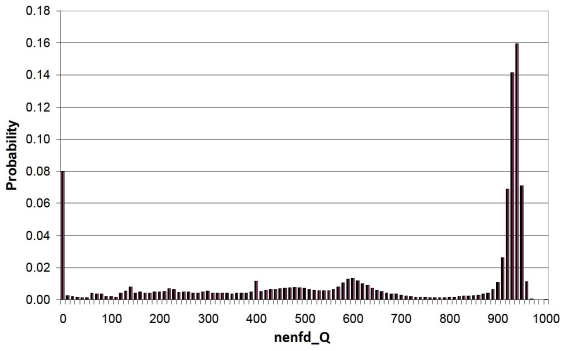

Figure 1: Empirical probability density function for experimental data set of doc files taken from (Roussev, 2009, 2010a)

Based on the probability distribution, each element of set A (all possible outcomes) is now assigned a rank. Let t_a = Pr[nenfd_Q=a] ∀ a∈A, where A is the set of integers {0,1,2,3,. . . .,1000}.

t_0= Pr[nenfd_Q=0],
t_1= Pr[nenfd_Q=1],
.
.
.
t_{1000}= Pr[nenfd_Q=1000].

We assign a rank r_i to each t_i as follows: r_i=1000 if t_i is the largest, and r_i=0 if t_i is the smallest. Now each feature f_k of D is assigned a precedence rank $R_{prec,D}(f_k)$ as follows:

$$\forall\ f_k \text{ of D, } R_{prec,D}(f_k) = r_i, \text{ where } \Pr[\text{nenfd\_Q}=H_{norm}(nbf_k)]=t_i$$

where D is the given data object, n is number of features of data object D and $0\leq k<n$. Data type of data object D and data objects $D^i(0\leq i<q)$ of set Q is the same. $H_{norm}(nbf_k)$ is normalized entropy of next byte of feature f_k of data object D. Essentially, the least common f_k gets the lowest rank whereas the most common one is assigned the highest rank.

Now based on the precedence rank, each feature(f_k) is assigned a popularity score denoted by $R_{pop,D}(f_k)$. The non-zero popularity score of a feature f_k of a data object D shows that there are ($R_{pop,D}(f_k)$+W-1) W-neighboring features of a feature f_k such that the precedence rank of its left neighboring feature f_i is greater than the precedence rank of f_k; and precedence rank of its right neighboring features f_j is greater than or equal to precedence rank of f_k where i<k, j>k; and number of f_i + number of f_j = ($R_{pop,D}(f_k)$+W-1). W-neighboring features of feature f_k: A feature f_{nb} is called a W-neighboring feature of feature f_k if |k-W| <nb<k or k<nb< |k+W|. If |k-W| < nb < k: feature f_{nb} is called left neighboring feature of feature f_k. If k < nb < |k+W|: feature f_{nb} is called right neighboring feature of feature f_k. $R_{pop,D}(f_k)$ for $0\leq k<n$ is calculated as follows:

- Initialize $R_{pop,D}(f_k) = 0$ for each $0\leq k<n$.
- Consider a window of size W (64).
- For every sliding window leftmost feature f_k with lowest $R_{prec,D}(f_k)$ is taken and value of $R_{prec,D}(f_k)$ is incremented by 1.
- Slide window by one and same steps are repeated (n-W) times,.

Fig.2 shows an example of the $R_{pop,D}(f_k)$ calculation of a data object D where n=18.

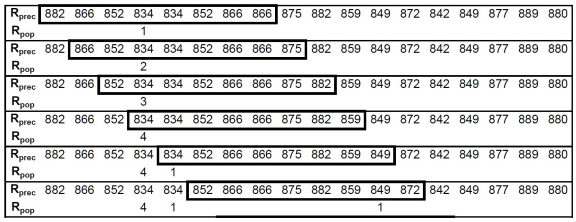

Figure 2: Popularity Rank Calculation from (Roussev, 2009, 2010a)

Now features with $R_{pop,D}(f_k) \geq$ t(threshold) are selected (in sdhash implementation t=16). Selected features are the least likely features to occur in any data object. These features are called "Statistically Improbable Features". These Statistically Improbable Features will be used to generate fingerprint of the data object D. Let $\{f_{s_0}, f_{s_1}, \ldots, f_{s_x}\}$ are the selected features, where $0 < x <$n, and n is the total number of features of data object D. SHA-1 hash of each selected feature is calculated. Then the resultant 160 bit hash is split into 5 chunks of 32 bits and least significant 11 bits of each chunk are used to address a bit in the bloom filter array. sdhash implementation uses 256 byte (2^{11} bit) bloom filter with maximum 128 element per filter (i.e. 5 bits per feature hence 640 bits per bloom filter). Similarity between two different sdhash digests is defined as the number of overlapping bits of the corresponding bloom filters. Let bf_1 and bf_2 are two bloom filters. Then the similarity between two bloom filter is defined as follows and represented as $SF_{score}(bf_1, bf_2)$:

$$SF_{score}(bf_1, bf_2) = \begin{cases} 0, & \text{if } e \leq C \\ [100\frac{e-C}{E_{max}-C}], & \text{otherwise} \end{cases}$$

where $e = |bf_1 \cap bf_2|$ (number of overlapping bits), C is the cuttoff of minimum and maximum number of possible overlapping bits by chance, defined as $C = \alpha(E_{max} - E_{min}) + E_{min}$ and E_{min}, E_{min} are minimum and maximum number of overlapping bits by chance respectively.

5. EXISTING RESULTS:

Implementation and security analysis of sdhash has been done by Breitinger et al. (Breitinger et al., 2012). Two of the implementation bugs, 'Window size bug' and 'left most bug' mentioned in (Breitinger et al., 2012) still exist in the latest version 3.4 of sdhash implementation.

Listing 1 shows the implementation of above stated bugs. At line number 13, there is an error in first condition that causes incorrect identification of minimum precedence rank ($R_{prec,D}(f_k)$), referred to as the 'Window size bug'. This error can be removed by replacing the first condition of while loop with 'chunk_ranks[i+pop_win-1] $\geq$ min_rank'. There is another error in the if condition at line number 14-15 and 26-27, that has been referred to as the 'Left most bug'.

If two features (f_i, f_j) have equal precedence rank ($R_{prec,D}(f_i) = R_{prec,D}(f_j)$) and are lowest within a popularity window, then this condition will cause the selection of right most feature that contradicts the proposed sdhash scheme. According to the proposed sdhash scheme (Roussev, 2010b), the left most feature with lowest precedence rank should get selected. In order to mitigate this bug, line numbers 14-15 & 26-27 should be removed from the code. We corrected the above mentioned bugs in the provided sdhash version 3.4[3], and the same corrected code is used to carry out experiments stated in § 6.

In the same work (Breitinger et al., 2012), the authors have indicated that undiscovered modifications to the input of sdhash are possible. However, details of how to achieve this are not provided. An undiscovered modification means that input can be modified without influencing the final sdhash digest. Therefore two or more files can generate same sdhash digest, which is called collision in terms of cryptographic hash functions and such files are called **'colliding files'** in rest of this paper.

Collision detection violates one of the basic properties of similarity preserving hash functions called 'Coverage' (Breitinger & Baier, 2012). Every byte of input is expected to influence the final sdhash digest. Breitinger et al. (Breitinger & Baier, 2012) have statistically shown that 20% of the input bytes are not part of any selected fea-

[3]http://roussev.net/sdhash/sdhash.html

Table 1: Different statistic on sdhash from (Breitinger & Baiber, 2012)

	Average	improved	original
1	filesize*	428,912	428,912
2	gaps count	2888	2889
3	min_gap*	1.09	1.076
4	max_gap*	1834	1834
5	avg_gap*	33.46	34.27
6	ratio to file size	20.65%	21.21%

ture. So, these bytes are not expected to influence the final sdhash digest, and are referred to as **'gap'**. Table 1 shows statistics of both original sdhash code and improved code (after correcting the bugs discussed above).

```
1  void
2  sdbf::gen_chunk_scores( const uint16_t *
       chunk_ranks, const uint64_t
       chunk_size, uint16_t *chunk_scores,
       int32_t *score_histo) {
3      uint64_t i, j;
4      uint32_t pop_win = config->
           pop_win_size;
5      uint64_t min_pos = 0;
6      uint16_t min_rank = chunk_ranks[
           min_pos];
7
8      memset( chunk_scores, 0, chunk_size*
           sizeof( uint16_t));
9      if (chunk_size > pop_win) {
10         for( i=0; i<chunk_size-pop_win; i
               ++) {
11             // try sliding on the cheap
12             if( i>0 && min_rank>0) {
13                 while( chunk_ranks[i+
                       pop_win] >= min_rank
                       && i<min_pos && i<
                       chunk_size-pop_win+1)
                        {
14                     if( chunk_ranks[i+
                           pop_win] ==
                           min_rank)
15                         min_pos = i+
                               pop_win;
16                     chunk_scores[min_pos
                           ]++;
17                     i++;
18                 }
19             }
20             min_pos = i;
21             min_rank = chunk_ranks[
                   min_pos];
22             for( j=i+1; j<i+pop_win; j++)
                    {
23                 if( chunk_ranks[j] <
                       min_rank &&
                       chunk_ranks[j]) {
24                     min_rank =
                           chunk_ranks[j];
25                     min_pos = j;
26                 } else if( min_pos == j-1
                       && chunk_ranks[j] ==
                       min_rank) {
27                     min_pos = j;
28                 }
29             }
30             if( chunk_ranks[min_pos] > 0)
                    {
31                 chunk_scores[min_pos]++;
32             }
33         }
34         // Generate score histogram (for
               b-sdbf signatures)
35         if( score_histo) {
36             for( i=0; i<chunk_size-
                   pop_win; i++)
37                 score_histo[chunk_scores[
                       i]]++;
38         }
39     }
40 }
```

Listing 1: sdfb_core.cc from sdhash-3.4

6. OUR CONTRIBUTION

The purpose of fuzzy hashing or similarity hashing schemes is to filter similar or correlated files corresponding to a given file that an investigator needs to examine. These schemes reduce the search space and corresponding manual effort of analysis for the investigator. The process of filtering the files by matching them with a set of already known to be bad files is called **Blacklisting**.

We propose a scheme that can generate multiple similar files corresponding to a given file with a similarity score of 100 for the sdhash similarity hashing. The scheme shows a weakness of the sdhash algorithm that an attacker could exploit to confuse and delay the investigative process. An example attack scenario is explained in the following paragraph.

Let us suppose a scenario where a suspected person 'X' has accessed and downloaded some proprietary images from a commercial website 'A' while she is logged in as a registered user. X, as an anonymous owner, runs a parallel website 'B' that hosts content from the original website available for free from a hosting location in some different part of the world. She intends to popularize her website 'B' to get a large viewership

that might attract web advertisers to put their ads on the website. A consistent viewership over a period would result in high chances of advertisement hits and consequently monetary returns for X. She would recover the membership cost gradually while the rest of the revenue is profit.

The original website 'A' eventually comes to know about the existence of website 'B' which is hosting their proprietary content. Since the owner of the domain name is registered as anonymous on records, the only way to track her is her IP address. Fortunately, the country where the website is hosted follows anti-piracy and Intellectual property protection laws. The physical location of systems on which the data of website 'B' is stored can be determined. X uploads content downloaded from original website after putting a watermark of his own website on each image. The use of cryptographic hash functions is ruled out in that case and investigators would need a similarity digest algorithm, possibly sdhash to find the files.

Here, in this condition if X has any time to prepare herself for such an investigation, she could use our tool to generate multiple similar files, with same metadata, corresponding to each file. The approach is definitely heavy on storage but can help X in increasing the effort of the investigation by forcing the investigators to analyze the files manually. Secondly, the investigation process could also be confused as by X's claim that she is innocent and it is a work of someone else who has access to her system or even a malware. In both the cases, investigation effort is increased many folds. Moreover, the primary purpose of a similarity digest to help investigators quickly filter out files of interest is defeated.

Breitinger et al. in (Breitinger et al., 2012) mentioned that 20% of the input data can be modify without influencing the final sdhash digest. We used two approaches to verify the number of undiscovered modification within gaps. These are (1) Random modification and (2) Deliberate Modification.

In the random modification approach, gap bytes are filled with randomly chosen ASCII characters. Our experiments on text files show that random modification of only 2% of the gap bytes influences the sdhash digest with probability close to 1. In the second approach of 'Deliberate modification' we propose an algorithm for careful modifications in order to increase the available bytes for modification within gaps. Experimental analysis of the proposed algorithm shows that by using this algorithm, around 12% of the gap bytes can be modified with maximal similarity score of 100.

6.1 Random Modification

We randomly choose several byte positions within the gap and modify each with a randomly chosen ASCII character to find the maximum number of random modifications within the gap that do not influence the sdhash digest of the entire document. We performed experiments on a data set of 50 text files of variable size from the T5-corpus dataset. We found that even one byte of random modification within the gap would influence the sdhash digest with an average probability of 0.22, and the modification of all bytes in the 20% gap will impact the final hash digest with probability 1. So, we focused on finding the minimum number of modifications that would influence the final sdhash digest with probability 1.

We started with single byte modifications and generated more than 5000 files with only one byte tampering and evaluated its influence the hash digest.

We gradually increased number of modifications until the hashes for all 5000 files got influenced. It was found that with a random modification of only around 2% bytes of the gap there is an influence on the sdhash digest of each of the randomly generated file which is on an average 0.42% of the respective file size. Experimental results for a small sample of 8 files is given in table 2.

As described in § 3, only the selected features (statistically improbable features) participate in the generation of final similarity digest. Therefore gaps (the data bytes which are not part of any selected feature) are expected not to influence the final hash digest. However,

Table 2: Minimum number of random modification, that modifies final sdhash digest with probability 1.

S.No.	File size (In KBs)	Gap (In Bytes)	Random Modification		
			Bytes	Gap%	File%
1†	1.5	354	45	12.70%	3%
2	22.9	3948	50	1.26%	0.21%
3	50	8917	70	0.78%	0.14%
4	81	14084	60	0.42%	0.07%
5	307	46296	80	0.17%	0.03%
6	841	215894	20	0.01%	0.00%
7	1095	139038	50	0.04%	0.00%
8	1554	378636	35	0.01%	0.00%
On an avg.			51.25	1.92%	0.42%

† This file is not from T5-corpus database

as we showed in the experiments, these bytes do influence the sdhash digest. This happens since each feature in the sdhash construction is highly correlated to its neighbors. Each feature differs from its left and right neighbor by only one byte. For example, let D be a data object under investigation which has the following byte sequence and features.

$B_0B_1B_2B_3B_4B_5. . . . . B_{63}B_{64}B_{65}B_{66}B_{67}. . .B_N$

f_0 $\boxed{B_0B_1B_2B_3B_4B_5. . . . . B_{63}}B_{64}B_{65}B_{66}B_{67}. . .B_N$

f_1 $\boxed{B_1B_2B_3B_4B_5B_6 . . . . . B_{64}}B_{65}B_{66}B_{67}. . .B_N$

f_2 $\boxed{B_2B_3B_4B_5B_6B_7 . . . . . B_{65}}B_{66}B_{67}. . .B_N$

.

.

.

f_n $\boxed{B_{N-63}B_{N-62}. . . . . .B_{N-2}B_{N-1}B_N}$

where N is the number of bytes in the data object D, and n is the number of features in D (n=N-L+1). Each byte is part of atleast one and at-most L (i.e. 64) features. Each byte (B_k), except the first L-1 and the last L-1 bytes (L≤k≤N-L+1), is part of exactly L features. Change in any byte, B_k will reflect in a change in features f_k to f_{k-L+1}, which may lead to a change in the precedence ranks $R_{prec,D}(f_{k-L+1})$ to $R_{prec,D}(f_k)$. A change in the rank of any feature($R_{prec,D}(f_k)$) will reflect in a change in the popularity score of features of D, which may affect the list of selected features. Any modification in the list of selected features will lead to changes in the final hash digest.

6.2 Deliberate Modification

The experiment results from § 6.1 show that the entire 20% gap of any file cannot be modified by random modification. We now propose an algorithm that performs careful modifications in order to increase the number of changes within the gaps while still ensuring no change in the similarity digest.

6.2.1 Algorithm Description

As discussed in § 6.1, modification in any byte B_k will influence the rank of all features containing B_k. This might cause changes in the list of selected statically improbable features. In the sdhash construction, a feature with leftmost lowest rank gets selected in a popularity window. If the rank of a feature is leftmost lowest in t or more than t (threshold) popularity windows then it gets selected as a statistically improbable feature. These selected statistically improbable features participate in the computation of the final sdhash digest. Let D be a data object with f_{S_1} and f_{S_2} as two consecutive statistically improbable features.

f_0 f_1 f_2. . . (f_{S_1}) f_{S_1+1} . . . f_{S_1+63} f_{S_1+64}. . . f_{S_2-1} (f_{S_2}). . . f_n

$B_0B_1B_2$. . . B_{S_1} B_{S_1+1}. . . B_{S_1+63} $\boxed{B_{S_1+64}. . .B_{S_2-1}}$ B_{S_2}. . . B_n. . . B_N

where f_{s_1} : $B_{s_1}B_{s_1+1}B_{s_1+2}. . . . .B_{S_1+L+1}$

f_{s_2} : $B_{s_2}B_{s_2+1}B_{s_2+2}. . . . .B_{S_2+L+1}$

Data bytes B_{S_1+64} to B_{S_2-1} are not a part of any selected features. The aim is to modify these bytes in such a way that modified features never get selected over f_{S_1} and f_{S_2}. For every data byte B_k, where $S_1+L \leq k \leq S_2-1$, a specific value among all possible ASCII characters satisfying the following two conditions is chosen:

1. $R_{prec,D}(f_j') > R_{prec,D}(f_{S_2})$ AND $R_{prec,D}(f_j') \geq R_{prec,D}(f_{S_1})$

2. $R_{prec,D}(f_j') \geq R_{prec,D}(f_j)$

where (k-L+1) ≤ j ≤ k and (S_1+L) ≤ k ≤ S_2-1 and f_j' is modified feature f_j obtained as the result modification of byte B_k. The above two conditions ensure that all the modified features f_j' have rank $R_{prec,D}(f_j')$ greater than the rank of the right selected statistically improbable feature (j<S_2) i.e. $R_{prec,D}(f_{S_2})$. At the same time,

$R_{prec,D}(f_j')$ is greater than or equal to the rank of the left selected (j>S_1) statistically improbable feature, i.e. $R_{prec,D}(f_{S_1})$. It can be equal to this value because even if two features have equal rank, the left most feature always gets selected. Ultimately, no other feature gets selected over both the statistically improbable features.

The above mentioned conditions are not enough if (S_2-1)-(S_1+L) $\geq$ t, where L is the feature length and t is the threshold. Even if each modification satisfies both the conditions, still new features may get selected. The reason this happens is that if the distance between two selected features is more than L+t, then after modification, the rank of some modified features may become local minimum among their t or more neighbors. Since t is the threshold for a feature to get selected, it may get selected as a statistically improbable feature and hence may influence the final sdhash digest. In the case mentioned above, it needs to be verified that no modification causes any change in the list of selected statistically improbable features. To mitigate this problem, after modification of the gaps bytes being considered, the popularity score($R_{pop,D}$) of all the features of D is calculated. If any new feature, f_j' contains the popularity score $R_{pop,D}(f_j)$ > t then all the previous modifications are discarded. Similarly the gaps between each adjacent pair of selected improbable features are modified.

Algorithm 1 and 2 generate the multiple colliding files corresponding to a given data object with maximal similarity score. Each execution of algorithm 1 produces a different file with dissimilar modification and different number of modifications. Therefore, we can generate G^{256} different files with maximal similarity corresponding to a given file, where G denotes the total number of gap bytes in the data object. The attacker can easily confuse the investigator by generating a huge number of files corresponding to a malicious or desired file. Since our current implementation is focused on text files, so we have chosen the characters only from the set of 95 printable ASCII characters, starting from char 32 till char 126. The maximum number of files that can be generated are G^{95}, which is sufficiently large even for $G = 2$.

Table 3: Number of modification with maximal similarity score through proposed algorithm

S.No.	File size (In KBs)	Gap (In Bytes)	Random Modification		
			Bytes	Gap%	File%
1†	1.5	354	89	25.14%	5.99%
2	22.9	3948	552	13.98%	2.41%
3	50	8917	1065	11.94%	2.13%
4	81	14084	1273	9.03%	1.50%
5	307	46296	3357	7%	1.09%
6	841	215894	31371	14%	3.73%
7	1095	139038	6211	4%	0.56%
8	1554	378636	13787	3.60%	0.88%
On an avg.			7185.25	11.08%	2.28%

† This file is not from T5-corpus database

We ran the proposed algorithms for the same data set of 50 text files which were used for our earlier random experiment. We found that around 12% of the gap bytes can be modified with maximal similarity score of 100 using the proposed algorithm. This is a huge improvement over the random modification case when even 2% of the gap bytes cannot be modified without changing the final sdhash digest. Experimental results for a small sample of 8 files are presented in table 3.

7. COUNTER MEASURES

In order to reduce the amount of undiscovered modifications, we propose the following two mitigations.

7.1 Minimization of popularity score threshold

Decrease in the threshold of popularity score in selection of statistically improbable features will increase the number of selected features. This, in turn, will result in the reduction of gap bytes that could be modified without affecting the final sdhash digest.

7.2 Bit level feature formation

In the sdhash scheme, each feature differs from its neighboring features by one byte. Therefore, the attacker has 2^8 possible choices to modify the feature without influencing its neighboring feature. If each neighboring feature differs by

Algorithm 2 Byte Modification algorithm

```
buffer                                                              ▷ Input Data object
indx                                                                ▷ index of the selected feature
lst_indx                                                            ▷ index of last selected feature
RANK(buffer,i)                                  ▷ function that returns rank of i^th feature of data object buffer
SCORE(buffer,i,j)  ▷ function that calculates popularity score of i^th feature to j^th feature of data object buffer and returns an array containing popularity scores
flag                                                                ▷ A boolean Variable
rank_indx                                                           ▷ Unsigned int variable for rank of selected feature
rank_lst_indx                                                       ▷ Unsigned int variable for rank of last selected feature
rank_k, rank_i                                                      ▷ Unsigned int; Temporary variable
procedure MODIFY_BYTES(buffer, indx, lst_indx, pop_win_size)
    buffer_copy ← buffer                                            ▷ Creating one copy of data object
    rank_indx ← RANK(buffer,indx)                                   ▷ Rank of selected feature of buffer
    rank_lst_indx ← RANK(buffer,lst_indx)                           ▷ Rank of last selected feature of buffer
    for i ← indx − 1 to lst_indx + 1 do   ▷ Run through all intermediate bytes between two selected features byte by byte
        ch ← buffer[i]                                              ▷ ch is a char variable
        rank_i ← RANK(buffer_copy,i)                                ▷ Rank of ith feature of unmodified buffer
        for j ← 0 to 255 do                ▷ Run through all ASCII value 0 to 255 until all conditions are satisfied.
            temp ← rand()% 256
            buffer[i] ← temp                          ▷ i^th byte will be replaced by randomly chosen ASCII char temp
            flag ← true
            if RANK(buffer, i) > RANK_indx AND RANK(buffer, i) > RANK_lst_indx AND RANK(buffer, i) ≥ rank_i then
                                          ▷ Rank of Modified features should be greater than rank of selected neighboring features
                for k ← i − (w − 1) to lst_indx do                  ▷ Run through features those consist i^th Byte
                    rank_k ← RANK(buffer_copy,k)                    ▷ Rank of k^th feature of unmodified buffer
                    if RANK(buffer, k) ≤ RANK_indx OR RANK(buffer, k) < RANK_lst_indx OR RANK(buffer, k) < RANK_k then
                        flag ← false
                        break                          ▷ Go out of the current loop and check for other values of j
                    end if
                end for
            else
                flag ← false
                break                  ▷ Change the i^th byte to ASCII character j, check for next byte
            end if
        end for
        if flag == false then
            buffer[j] ← ch;                                         ▷ reset the j^th charecter to its actual value
        end if
    end for
    score ← SCORE(buffer,lst_indx+1,indx-1)
    high ← false;
    for x ← 0 to (indx − lst_indx − 1) do
        if score[x] > 16 then
            high ← true break;
        end if
    end for
    if high == true then
        for z ← (indx − 1) to lst_indx do
            buffer[z] ← buffer_copy[z]                              ▷ Revert all the changes
        end for
    end if
end procedure
```

Algorithm 1

```
1: buffer                                   ▷ Data object
2: chunk_size                               ▷ Size of data object
3: chunk_score ▷ Array of score of each feature
   of the data object
4: pop_win_size    ▷ Window size: default is 64
5: t                                        ▷ Threshold: default is 16
6: indx            ▷ index of the selected feature
7: lst_indx        ▷ index of last selected feature:
   initialize with 0.
8: for i ← 0 to chunk_size − pop_win_size do
   ▷ Run through input byte by byte
9:     if chunk_scores[i] > t then    ▷ Selected
   features
10:        modify_bytes(buffer, indx, lst_indx,
   pop_win_size)
11:                ▷ Processing is in next algorithm
12:        lst_indx ← indx
13:
14:    end if
15: end for
```

only 1 bit (in place of the original one byte), it will reduce the number of possible choices with the attacker from 256 to 2. Hence the probability of modifying each bit without affecting the final hash will also get reduced substantially. However, it will increase the number of features and hence the selected features, thereby causing some loss in efficiency. Increase in the number of selected improbable features will not only increase the computation time, it will also cause an increase in the size of the final sdhash digest.

8. CONCLUSION

Currently sdhash is one of the most widely used byte-wise similarity hashing scheme. It is possible to do undiscovered modification to a file and yet obtain exactly the same sdhash digest. We have proposed a novel approach to do maximum number of byte modification with maximal similarity score of 100. We also provided a method to do an anti-forensic attack in order to confuse or delay the investigation process.

REFERENCES

Baier, H., & Breitinger, F. (2011). Security aspects of piecewise hashing in computer forensics. In *IT security incident management and IT forensics (IMF), 2011 sixth intl. conference on* (pp. 21–36).

Breitinger, F., & Baier, H. (2012). Properties of a similarity preserving hash function and their realization in sdhash. In *2012 Information Security for South Africa, johannesburg, 2012* (pp. 1–8).

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*.

Chen, L., & Wang, G. (2008). An efficient piecewise hashing method for computer forensics. In *Knowledge discovery and data mining, 2008. first intl. workshop on* (pp. 635–638).

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

Roussev, V. (2009). Building a better similarity trap with statistically improbable features. In *System sciences, 2009. 42nd hawaii intl. conference on* (pp. 1–10).

Roussev, V. (2010a). Data fingerprinting with similarity digests. In *Advances in digital forensics vi* (pp. 207–226).

Roussev, V. (2010b). Data fingerprinting with similarity digests. In *Advances in digital forensics vi* (pp. 207–226).

Seo, K., Lim, K., Choi, J., Chang, K., & Lee, S. (2009). Detecting similar files based on hash and statistical analysis for digital forensic investigation. In *2009 2nd international conference on computer science and its applications.*

Tridgell, A. (2002). *Spamsum readme.*