Security and Implementation Analysis of the Similarity Digest `sdhash`

Frank Breitinger & Harald Baier & Jesse Beckingham
Center for Advanced Security Research Darmstadt (CASED)
and Department of Computer Science, Hochschule Darmstadt,
Mornewegstr. 32, 64293 Darmstadt, Germany
email: {frank.breitinger,harald.baier}@h-da.de, jesse@beckingham.de

Abstract: Cryptographic hash functions are very sensitive – even if only one bit of the input is changed the output behaves pseudo randomly. Thus the similarity between two inputs cannot be identified. In 2010 Roussev presented a new similarity preserving fingerprinting technique based on statistical chosen features (the most unique ones) called `sdhash`. However, his proposal lacks a thorough implementation and security analysis.

This paper concludes after a thorough analysis that `sdhash` has the potential to be a robust similarity preserving digest algorithm. However, we present some inconsistencies between the specification of `sdhash` and its implementation, which leads to unexpected results of `sdhash`. Furthermore, we uncovered design errors within the fingerprint generation and its comparison. The security part shows that given a file it is easily possible to tamper the file with to come down to a similarity score of approximately 28, but that it is hard to overcome the matching algorithm completely.

1 Introduction

The crucial task during a computer forensic acquisition is to distinguish relevant from non-relevant files. As the amount of data an investigator has to deal with is growing rapidly, it is not possible to look at each file by hand. An automated pre-processing tries to filter out known-to-be-good and known-to-be-bad files by using cryptographic hash functions. The proceeding is quite simple: the investigator computes hash values of all files which he finds on a storage medium and performs look ups in a database, e.g., National Software Reference Library (NSRL, (NIS12)).

Cryptographic hash functions meet several security requirements and thus the hash value of a cryptographic hash function behaves pseudo-randomly if the input changes, i.e., if the input changes (e.g. by one bit), approximately 50% of the output bits change. Comparing the similarity of files using cryptographic hash functions is therefore not possible. This could be easily exploited by an active adversary.

(Rou10) came up with a new idea called Similarity Digests Hashing including a prototype called `sdhash`. Although this tool becomes very popular, e.g., NIST includes it into its NSRL, a thorough analysis with respect to correct implementation and security properties is missing. Thus, this paper is an analysis of `sdhash 1.2` and focuses on the specification and its implementation. We do not address any questions about the underlying design or the chosen parameters.

1.1 Contributions and Organization of this Paper

Our main contribution is to show that `sdhash` is a robust approach, but an active adversary can beat down the similarity score to approximately 28 while preserving the perceptual behaviour of a file (e.g., an image keeps it visual outcome to its observer). Furthermore we uncovered some implementation and design errors which lead to an inconsistency between the implementation and the description of the algorithm. For all cases we provide some solutions to improve `sdhash`.

The rest of the paper is organized as follows: In the subsequent Sec. 1.2 we introduce notation and terms, which we use throughout this paper. Then, in Sec. 2 we sketch the state of the art and discuss relevant literature. Next, we show in Sec. 3 the foundations of the similarity digest fingerprint `sdhash`, which is necessary to understand our improvements and attacks. The core of our paper is given in Sec. 4 and Sec. 5, where we analyze the implementation and present the security aspects, respectively. Sec. 6 concludes our paper.

1.2 Notation and Terms used in this Paper

In this paper, we make use of the the following notation and terms, which will be introduced and explained in the subsequent sections:

- h denotes a cryptographic hash function (e.g., SHA-1, MD5, RIPEMD-160).
- IN is a byte string of length L, $IN = B_0 B_1 B_2 \dots B_{L-1}$ called input.
- f_k is a sub byte string in IN starting at offset k with a length of l. We call f a feature. The implementation uses $l = 64$.
- H_{norm} denotes the normalized entropy score.
- R_{prec} is the precedence rank which has been obtained by preliminary statistical analysis.
- R_{pop} denotes the popularity score of a feature.
- F is a specific feature f whose R_{pop} is higher-equal than a given threshold. The implementation uses a threshold of 16.
- bf is a Bloom filter of 256 bytes containing a maximum of 128 features.
- $|bf|$ denotes the amount of bits set to one within bf.
- $\overline{bf}$ denotes the amount of features within bf.

2 Related Work

According to (MOV97) hash functions have two basic properties, *compression* and *ease of computation*. In this case compression means that regardless the length of the input, the output has a fixed length. This is why the term Fuzzy *Hashing* might be a little bit confusing and *similarity digest* is more appropriate. But as most of the similarity preserving algorithms do not output a fixed sized hash value, we use similarity digest, fuzzy hash function and similarity preserving hash function as synonyms.

A first algorithm for fuzzy hashing was introduced by (Kor06). He came up with context triggered piecewise hashing, abbreviated as CTPH, which is based on a spam detection algorithm of (Tri02). The main idea is to compute cryptographic hashes not over the whole file, but over parts of the file, which are called *chunks*. The end of each chunk is determined by a pseudo-random function that is based on a current context of 7 bytes. Since then several papers had been published which examined this approach carefully. For instance, improvements with respect to efficiency and security had been presented by (BB11b; RRM07; CW08; SLC^{+}09) whereas a security analysis is presented by (BB11a; Bre11). All in all this approach cannot withstand an active adversary with respect to blacklisting and whitelisting.

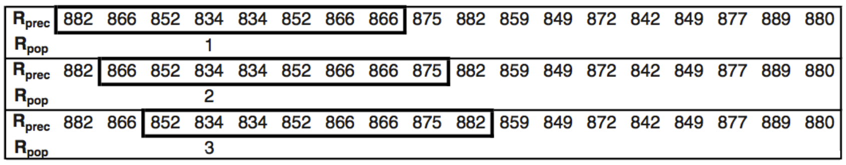
Based on some previous work named `md5bloom` (RCBR06), (Rou09; Rou10) came up with a new idea called Similarity Digests Hashing including a prototype called `sdhash` in 2010. The main idea is to identify "statistically-improbable features" using an entropy calculation for each 64 byte sequence. Once a "characteristic" feature is identified it is hashed using a cryptographic hash function (e.g., (Riv92; SHS95)) and inserted into a Bloom filter (Blo70). Hence, files are similar if they have common features. More details are given in Sec. 3.

(Rou11) provides a comparison of `ssdeep` and `sdhash` and shows that the latter "approach significantly outperforms in terms of recall and precision in all tested scenarios and demonstrates robust and scalable behaviour".

3 Foundations of `sdhash`

We introduce the main concepts of the similarity digest algorithm (`sdhash 1.2`) as proposed by (Rou10) in what follows. As the parallelized `sdhash 2.0` presented by (Rou12) was not available when doing our research, we only work on the older version. We argue that the updated algorithm is roughly the same and therefore working on version 1.2 is sufficient. In the following we summarize the main properties of Roussev's approach that are relevant for the remainder of this paper.

Let IN be a byte sequence $B_0, B_1 \ldots B_{L-1}$ of length L called input. Then a feature f_k is



R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				1														
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				2														
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				3														

Figure 1: Example for the R_{pop} calculation from (Rou10).

a sub byte sequence of length l in IN starting at B_k with $0 \leq k \leq L - l$, i.e.,

$$\begin{aligned} f_0 &= B_0, B_1 \ldots B_{63} \\ f_1 &= B_1, B_2 \ldots B_{64} \\ &\ldots \\ f_{L-l} &= B_{L-l}, B_{L-l+1} \ldots B_{L-1} \end{aligned}$$

For every feature f_k the following two steps are required:

- First, the normalized Shannon entropy score H_{norm} is calculated on base of the empirical entropy H of f_k

$$H = -\sum_{i=0}^{255} P(X_i) \cdot \log_2 \left(P(X_i) \right) \ , \tag{1}$$

  where $P(X_i)$ is the empirical probability (i.e., the relative frequency) of encountering ASCII code i in f_k. Then H is scaled to a value in the integer range $[0, 1000]$ using

$$H_{norm} = \lfloor 1000 \cdot H / \log_2 l \rfloor \ . \tag{2}$$

- Second, (Rou09) states “we associate a *precedence rank* [(abbreviated R_{prec})] with each entropy measure value that is proportional to the probability that it will be encountered. In other words the least likely features measured by its entropy score gets the lowest rank.” The result is a sequence of R_{prec} values.

Next is the identification of the *popular* features which is done using a sliding window Win of a fixed size W (`sdhash` uses $W = 64$) going through all R_{prec} values. At each position `sdhash` increments the R_{pop} score for the leftmost feature with the lowest R_{prec} within Win.

An example is given in Fig. 1 where the size of the window is set to $W = 8$. Let $R_{prec}(i)$ and $R_{pop}(i)$ denote the precedence and popularity rank of f_i, respectively. In Fig. 1 we have $R_{prec}(0) = 882$, $R_{prec}(1) = 866, \ldots$. As $R_{prec}(3) = 834$ has the leftmost lowest R_{prec} within Win, $R_{pop}(3)$ is incremented and the window slides. Within the second iteration $R_{prec}(3)$ is still the leftmost lowest R_{prec} in Win and $R_{pop}(3)$ is incremented again, and so on. All features whose R_{pop} score are higher-equal than a given threshold (`sdhash` uses 16) are part of the fingerprint. We denote these features $F_0, F_1, \dots F_p$ (capital F).

As the threshold is 16, the minimum byte distance between neighboring features F_i and F_{i+1} is 16. For instance, let E be the last element within the window and also having the lowest R_{prec}. As E is the last element, the R_{pop} could be at most one. When sliding the window there are two possibilities, the R_{prec} of the new element

1. is higher-equal, than R_{pop} of E is increased.
2. is lower, than the R_{pop} of the new element is increased.

A more general argumentation shows that if $R_{pop}(i) = k$ $(1 \leq k \leq 64)$, then $R_{pop}(i + n) \leq n$ for all $1 \leq n < k$.

In order to generate the similarity digest, the byte string of each corresponding feature $F_0, F_1, \dots F_p$ is hashed using SHA-1 and the resulting 160 bit hash value is split into five sub hashes of 32 bit length. As Roussev's Bloom filters consist of 256 bytes = 2048 bits = 2^{11} bits, he uses 11 bits from each sub hash to set the corresponding bit in the Bloom filter.

Roussev decides for a maximum of 128 features per Bloom filter which results in a maximum of 128 features $\cdot\ 5 \frac{\text{bits}}{\text{feature}} = 640$ bits per Bloom filter. If an input has more features, a new Bloom filter is created.

To define the similarity of two Bloom filters, we have to make some assumptions of the minimum and maximum overlapping bits by chance wherefore Roussev introduces a cutoff point C. Let $|bf|$ denote the number of bits set to one within a Bloom filter. If $|bf \cap bf'| \leq C$, then the similarity score is set to zero.

C is determined as follows

$$C = \alpha \cdot (E_{max} - E_{min}) + E_{min} \tag{3}$$

where α is set to 0.3, E_{min} is the minimum number of overlapping bits due to chance and E_{max} the maximum number of possible overlapping bits. Thus E_{max} is defined as

$$E_{max} = \min(|bf|, |bf'|). \tag{4}$$

Let j be the number of sub hashes (=5 within `sdhash`), $\overline{bf}$ the amount of features[1] within a Bloom filter, m the size of a Bloom filter in bits (=2048) and $p = 1 - 1/m$ the probability that a certain bit is not set to one when inserting a bit. Thus

$$E_{min} = m \cdot (1 - p^{j \cdot \overline{bf}} - p^{j \cdot \overline{bf'}} + p^{j \cdot (\overline{bf} + \overline{bf'})}) \tag{5}$$

is an estimation of the amount of expected common bits set to one in the two Bloom filters bf, bf' by chance.

[1]Except for the last Bloom filter this value is always 128 for an input.

Let $SD_1 = \{bf_1, bf_2, \dots bf_s\}$ and $SD_2 = \{bf'_1, bf'_2, \dots bf'_r\}$ the similarity digests of two inputs and $s \leq r$. If $\overline{bf_1} < 6$ or $\overline{bf'_1} < 6$ then the original input does not contain enough features and the similarity score is -1, not comparable. Otherwise the similarity score is the mean value of the best matches of an all-against-all comparison of the Bloom filters, formally defined as

$$SD_{score}(SD_1, SD_2) = \frac{1}{s} \sum_{i=1}^{s} \max_{1 \leq j \leq r} SF_{score}(bf_i, bf'_j) \tag{6}$$

where SF_{score} is the similarity score of two Bloom filters

$$SF_{score}(bf, bf') = \begin{cases} 0, & \text{if } e \leq C \\ [100 \frac{e-C}{E_{max}-C}], & otherwise \end{cases} \tag{7}$$

with $e = |bf \cap bf'|$.

4 Evaluation of Implementation

In a first step this section compares the specification with the implementation – both have to coincide. As we treat the specification as 'correct', we show some discrepancies in Sec. 4.1 and Sec. 4.2. The impact of a deviating implementation is an unexpected behaviour of the `sdhash` programme. Furthermore we discuss three bugs which are mainly shortcomings by design during the comparison of Bloom filters.

Remark: To evaluate our results we used the t5-corpus from (Rou11, Sec. 4.1.) which is a collection of 4457 files (1.8 GB)[2] from 4 KB up to 16.4 MB.

4.1 Popularity Rank Computation Bug

While inspecting the code from `sdhash`, we discovered two important bugs when computing the popularity rank R_{pop}:

1. A bug concerning the window size used to compute R_{pop} avoids the correct identification of the minimal R_{prec} value in the current window. We call this bug the *window size bug* in what follows.

2. An inconsistency between the description of the algorithm in (Rou10) and its implementation yields unexpected results for R_{pop}. The inconsistency is related to the property of the algorithm in (Rou10) to consider the leftmost minimal value of R_{prec} in a window. We therefore denote this bug as *leftmost bug*.

[2] `http://roussev.net/t5/t5-corpus.zip`; visited 02.01.2011

These two bugs lead to false results of R_{pop} and thus to an unexpected behaviour of the whole algorithm. The discussion in this section is with respect to the code of the file `sdbf_score.c` of `sdhash` version 1.2 shown in Listing 1. However, the same bugs are found in the current version 1.3, where the file is now called `sdbf_core.c` (we are not aware of any reason of the renaming).

Before explaining the bugs in `sdbf_score.c` we first give an example of unexpected values for R_{pop}. In Sec. 3 we explained the computation of R_{pop} as presented in (Rou10). We point to a key property of the popularity rank: Let $R_{pop}(i)$ denote the popularity rank at position i. If $R_{pop}(i) = k$ ($1 \leq k \leq 64$), then $R_{pop}(i + n) \leq n$ for all $1 \leq n < k$.

However, the following listing shows some unexpected values of R_{pop}. The listing is similar to Fig. 1 and it is an abridgment of the R_{prec} and its corresponding R_{pop} for file `000110.jpg`.

```
Pos:      76  77  78  79  80  81  82  83
R_prec: 432 353 333 333 325 396 432 472
R_pop:    0   0   1  63   2   1   1   1
[removed]
Pos:     603 604 605 606 607 608 609 610
R_prec: 372 364 335 335 391 416 443 445
R_pop:    1   1  25  40   1   1   1   1
```

The first line shows the position i of R_{prec} in `000110.jpg`, i.e., the offset of the first byte of the underlying feature of R_{prec}.

To recall (Rou10, p.212) says that after a window Win has slided, the R_{pop} of *leftmost lowest* R_{prec} in Win is increased, which is in contrast to the above listing. For instance, for $i = 78$ the 1-neighborhood (positions 77 and 79) should have at most a R_{pop} of 1. In fact we would expect an output like:

```
Pos:      76  77  78  79  80  81  82  83
R_prec: 432 353 333 333 325 396 432 472
R_pop:    0   0   2   0  63   1   1   1
[removed]
Pos:     603 604 605 606 607 608 609 610
R_prec: 372 364 335 335 391 416 443 445
R_pop:    1   1  64   1   1   1   1   1
```

As an explanation we imagine a window Win of size 64 ending at position 78. In order for $R_{pop} = 1$ to hold at position 78, Win must contain 63 values $R_{prec} > 333$ to the left of position 78. We then slide Win by one to position 79. The new incoming R_{prec} is also 333 and thus the R_{pop} at position 78 should increase again as it is still the leftmost lowest R_{prec} in Win. Sliding one step further there is even a lower $R_{prec} = 325$, wherefore R_{pop} at position 80 should be increased.

We now explain the location of the window size bug using Listing 1 and show that the false value $R_{pop} = 63$ at position 79 is due to this bug. The window size bug means that two different window sizes W are used in the code to compute R_{pop}. In line 92 of the listing the current window Win starts at position i and uses a window size W. However, the positions i until $i+W$ comprise $W+1$ positions and therefore a -1 needs to be added. On the other side, at line 101 of Listing 1 the correct window size W is used to find the minimal value in Win[3].

```
 86  UINT min_pos = 0;
 87  UINT min_rank = ranks[min_pos];
 88  for( i=0; i<sdbf_sys.file_size-sdbf_sys.pop_win_size; i++) {
 89     // try sliding on the cheap
 90     if( i>0 && min_rank) {
 91        UINT ix=0;
 92        while( ranks[i+sdbf_sys.pop_win_size] >= min_rank && i<min_pos && i<
              sdbf_sys.file_size-sdbf_sys.pop_win_size+1) {
 93           if( ranks[i+sdbf_sys.pop_win_size] == min_rank)
 94              min_pos = i+sdbf_sys.pop_win_size;
 95           pop_scores[min_pos]++;
 96           i++;
 97        }
 98     }
 99     min_pos = i;
100     min_rank = ranks[min_pos];
101     for( j=i+1; j<i+sdbf_sys.pop_win_size; j++) {
102        if( ranks[j] < min_rank && ranks[j]) {
103           min_rank = ranks[j];
104           min_pos = j;
105        } else if( min_pos == j-1 && ranks[j] == min_rank) {
106           min_pos = j;
107        }
108     }
109     if( ranks[min_pos] > 0) {
110        pop_scores[min_pos]++;
111     }
112  }
113  free( ranks);
114  fclose( in);
115  return pop_scores;
116  }
```

Listing 1: Abridgment of `sdbf_score.c` from `sdhash 1.2`.

We shortly discuss the implications of this bug. Due to the window size $W+1$ in line 92 the `while`-loop is always one R_{prec} ahead. Thus when the `while`-loop in line 92 processes position 80, it reads in $R_{pop} = 325$ where the first condition does not hold and we jump to line 99. However, the implementation now checks in line 101 for a window of size W which ends at position 79 and therefore computes a minimal value $R_{pop} = 333$.

Additionally the leftmost bug in lines 93 and 94 chooses for the rightmost position if the righmost R_{pop} is equal to the previous minimal value (a similar bug is implemented in lines 105 and 106).

To resolve these two bugs we propose the following changes to the source code:

[3]Actually also W is used, but the condition uses a $<$ instead of $\leq$.

1. Replace the first condition in line 92 by

   ```
   ranks[i+sdbf_sys.pop_win_size-1]
                                >= min_rank
   ```

   to resize the window to its correct length of 64.

2. Remove lines 93, 94, $105 - 107$ to correct the leftmost bug.

4.2 Unnoted Footer Features

Unnoted Footer Features means that we may have features at the end of a file, which are ignored by `sdhash` for the similarity digest computation. This behaviour results in two different effects besides the fact that it is not mentioned in (Rou10). On the one side it is possible to do manipulations at the end of a file which will not be discovered and on the other side it is possible to append information.

We shortly explain the Unnoted Footer Features and refer to (BB12) for further details. Let r denote the amount of Bloom filters of the similarity digest of an input. Based on (Rou10) there is only one restriction: If $r = 1$ and $\overline{bf_r} < 6$ the generation process stops and prints an error message. In fact this is different to the actual implementation where a second condition is present: If $r \geq 2$ and $\overline{bf_r} < 16$ then bf_r is skipped (`sdbf_api.c` line 163). This means that we may append up to 15 features without being noticed if the similarity digest of the input file comprises $128 \cdot r$ features, $r \geq 1$.

4.3 Design Errors within the Comparison Class

The comparison function of a similarity digest algorithm is crucial for detecting related files. However, when reviewing the code of the matching part of `sdhash`, we discovered some shortcomings by design:

- In case that one of the files has at most 63 features, the comparison functions yields two different results depending on the order of the two files. If the file containing at most 63 features is invoked as the second argument it results in the similarity score -1, a not comparable. We denote this bug by *not-comparable bug*.
- A self-comparison of a file may result in a similarity score significantly smaller than 100 (a lower bound is 50). We denote this bug by *self-comparison bug*.
- It is possible that two different files yield a 100 match by design (and not at random). We denote this bug by *collision bug*.

Responsible for the first two bugs is the misplaced if-condition `if(s2 < 64)` in the file `sdbf_score.c` in line 198 where `s2` denotes the amount of features within a Bloom filter. Roughly speaking this if-condition skips the comparison of partially filled Bloom filters, if the threshold of 63 features is not exceeded.

Not-comparable bug: This bug can arise while comparing two files where at least one has at most 63 features and thus results in only one Bloom filter. Listing 2 shows the comparison of a 48-feature-file against a 84-feature-file and vice-versa.

As a result we obtain a match score of 100 or a -1 (not-comparable) depending on the ordering. Both results are not reasonable. The supposed perfect match is due to the collision bug as explained below. The not-comparable result is due to the aforementioned if-condition: The comparison is skipped if the second input file does not have enough (≥ 64) features.

```
$ sdhash -g 48.ftrs 84.ftrs
48.ftrs 84.ftrs 100

$ sdhash -g 84.ftrs 48.ftrs
84.ftrs 48.ftrs -01
```

Listing 2: Wrong order comparison yields a not comparable.

A possible solution is given at the end of ‘self-comparison bug’.

Self-comparison bug: Let $SD = \{bf_1, bf_2\}$ be the similarity digest of the input for the self-comparison, where $\overline{bf_1} = 128$ and $\overline{bf_2} < 64$ (i.e., the first Bloom filter contains the maximum amount of 128 features and the second one is below the self-comparison threshold of 64 features as defined by the `if`-condition from above).

To receive the similarity score there is an all-against-all comparison of Bloom filters as defined by Eq. (6) of the two similarity digests. In our sample case, SD is compared against itself. Thus $SF_{score}(bf_1, bf_1)$ results in a 100 match and is therefore the best match. But due to the `if`-condition it is not allowed to compute $SF_{score}(bf_2, bf_2)$ as the if-condition requires at least 64 features for the second Bloom filter bf_2. Therefore $SF_{score}(bf_2, bf_1)$ is used. However, we have $0 \leq SF_{score}(bf_2, bf_1) \leq 100$, which yields a lower bound of 50 for the self-comparison similarity score.

```
1 $ sdhash t5/000256.doc
2 t5/000256.doc sdbf:sha1:256:5:7ff:128:2:18:
      IiYTAaVQMQRUUlEFbL0BDKUQAIkE0JEDMEEAgPIoiAi
      gDQNYAdsADCEdEGBAC9wBYiEA4AFARGYJdMASZEJABS [removed]
      AAAAAQECAAAAAAAAAAAEEAgAQAACIAAAAEBAAA=
3
4 $ 1.2/sdhash -g t5/000256.doc t5/000256.doc
5 t5/000256.doc t5/000256.doc 055
```

Listing 3: A file compared to itself with a match score of only 55.

Let us look at an example from the t5-corpus given in Listing 3. First, we print the similarity digest for `00256.doc`. Row 2 shows some basics about the digest itself, e.g., `sdhash` uses SHA-1, has Bloom filters of 256 bytes each with 128 features, overall there are 2 Bloom filters and the last one contains 18 features. Due to the second Bloom filter with only 18 features the self-comparison in rows 4 and 5 yield an overall similarity score of 55. Eq. (6) yields the similarity score $SF_{score}(bf_2, bf_1)$:

$$\frac{100 + SF_{score}(bf_2, bf_1)}{2} = 55 \;,$$

hence $SF_{score}(bf_2, bf_1) = 10$.

To generalize, if the similarity score of a file is built up of r Bloom filters and the last one contains less than 64 features, it is not trustful. If such a file is compared to itself a lower bound of its similarity score is therefore $\frac{(r-1)\cdot 100+0}{r} = 100 - \frac{100}{r}$.

To avoid the two aforementioned bugs and Unnoted Footer Features from Sec. 4.2, all thresholds need to be adjusted where we recommend an upper bound of 6. In order not to reject the last Bloom filter if it contains less than 6 features, there should be an extension where we allow more than 128 elements within the last filter. Thus in case the last Bloom filter would be skipped due to too less features we merge the last two filters. As a consequence the last Bloom filter could have at most 133 features.

5 Security Aspects

In the following we address the security aspects of `sdhash`. Section 5.1 describes a possibility to make undiscovered changes within a file – the match score of the modified file to the original one remains 100. Then Sec. 5.2 discusses the relevance of the cutoff point C and analyzes the minimum amount of features that need to be manipulated for a complete non-match of two Bloom filters (i.e. a match score of 0). Next, the most obvious attack is described in Sec. 5.3 where we manipulate as much features until the similarity score is zero. Finally, Sec. 5.4 addresses further security considerations concerning an idea to reduce the similarity score of two files by inserting self-made features and the preimage resistance of `sdhash`.

5.1 Undiscovered Modifications

In Sec. 4.2 we've already discussed the possibility to make undiscovered modifications. Besides this special case, most of the input files allow to do modifications that won't be discovered which is discussed in this section.

By design the first 15 bytes will never influence the fingerprint as there have to be 16 slides of the window to obtain a popularity score of at least 16. An example where we exploited this issue is given in Listing 4. We used `00220.text` from the t5-corpus, copied it and compared them using flag `-g`. Next we edited the first 15 bytes within the file and compared them again using `sdhash` and `diff`.

Another possibility is based on the findings from Sec. 5.4.3 where it is shown that approximately 20% of the input bytes do not influence the similarity digest. Thus it is possible to do undiscovered modifications within gaps.

As a solution we propose to create the SHA-1 hash value over the whole input and treat it as a feature, i.e., insert it as first/last element into a Bloom filter.

```
$ cp 000220.text 000220.text.edt
$ 1.2/sdhash -g 000220.text 000220.text.edt
000220.text 000220.text.edt 100

$ vim 000220.text.edt
[modify first 15 bytes]

$ 1.2/sdhash -g 000220.text 000220.text.edt
000220.text 000220.text.edt 100

$ diff 000220.text 000220.text.edt
1c1
< WWL DATA POINT ANALYSIS
---
 > PASSWORD dfgjT ANALYSIS
```

Listing 4: A match score of 100 despite some changes.

5.2 Bloom Filter Resistance

Let *IN*, *IN'* be two identical inputs yielding one full Bloom filter as similarity digest. Then this section addresses the question, how many features do we have to change in *IN'* to obtain a full non-match with *IN*. As explained in Sec. 3, if e, the number of bits in common of two Bloom filters, is lower-equal than a given cutoff point C, their similarity is set to 0 (see Eq. (7)).

The average amount of bits set to one within a full Bloom filter bf of size $m = 2048$ with $j = 5$ sub hash functions by chance is

$$\begin{aligned} E_{avg} &= m \cdot \left(1 - (1 - \frac{1}{m})^{j \cdot \overline{bf}}\right) \\ &= 2048 \cdot (1 - 0.99951172^{5 \cdot 128}) = 549.77. \end{aligned}$$

As both filters approximately contain 550 bits, E_{max} is equal to E_{avg}.

On the other side the minimum overlapping bits by chance is given in Eq. (5) where $\overline{bf} = \overline{bf'} = 128$ as we concentrate on *full* Bloom filters. If we set $j = 5$ then

$$E_{min} = 147.433.$$

Based on this key data the cutoff point is $C = 0.3 \cdot (549.77 - 147.43) + 147.43 = 268.13$.

After defining the underlying conditions, we estimate how many bits change when we manipulate one feature. The probability that one bit is set to zero in a full Bloom filter is

$0.99951172^{5 \cdot 128} = 0.7315598$. Thus the manipulation of one feature will approximately change $0.7315598 \cdot 5 = 3.65779898$ bits within a Bloom filter. We successfully verified this value through an empirical test where we used 10,000 random files having exactly 128 features, manipulated the first one and analyzed the amount of varying bits.

According to Eq. (7), if $e \leq C$ the Bloom filters are treated as a non-match and thus we need to change $e - C$ bits. Two identical full Bloom filters have approximately 550 bits in common and a cutoff point C of 268 which result in $550 - 268 = 282$ bits. Due to the pseudo-randomness of SHA-1 we do a linear approximation. Thus by changing one feature we approximately change 3.66 bits wherefore we have to manipulate $\frac{282}{3.66} = 77.05$ features in *IN'*.

A further test on real-world data showed that approximately 83.37 changes are necessary to receive a non-match which might be because of the reoccurring features. Thus each feature changes $\frac{282}{83.37} = 3.383$ bits within a Bloom filter.

We consider the amount of manipulations that needs to be done for a non-match as high and we therefore rate `sdhash` as a very robust approach for fuzzy hashing.

5.3 Feature Modification

The most obvious idea to obtain a non-match is to manipulate the features as the similarity digest is based on the hashed features. Due to the use of a cryptographic hash function, one changed bit is enough to change the hash value.

Sec. 5.2 showed that it is sufficient to change 83.37 features per Bloom filter to reduce the similarity score of two Bloom filters to zero. As it is enough to change one bit per feature, we need to flip 83.37 bits. Furthermore a full Bloom filter represents $128 \cdot 64 = 8192$ bytes of the input file. From Sec. 5.4.3 we know that a lot features are overlapping which reduces the amount of needed changes.

As a consequence, within each chunk of 8192 bytes we have to change approximately $83.37 \cdot 0.6 = 50.02$ bits. This results in a lot of changes all over the file which is only feasible for locally non-sensitive file types, e.g., `bmp`, `txt` but only hardly for locally sensitive file types, e.g., `jpg`, `pdf`.

5.4 Further Security Considerations

This section addresses two further security aspects of `sdhash`: First, in Sec. 5.4.1 we refer to (BB12), where a method called *Bloom filter shifts* is discussed to reduce the similarity score of initially identical files to approximately 28 in constant time. Second, Sec. 5.4.2 discusses an improvement of the preimage resistance property of `sdhash`. Finally, Sec. 5.4.3 deals with the coverage of `sdhash`. The result is that not all input bytes influence the fingerprint.

5.4.1 Generating False Non-Matches

(BB12) discusses the issue of generating a false non-match by decreasing the match score to approximately 28. The idea is as follows: Given an input *IN*, insert at the beginning of *IN* a (fixed) sequence of bytes, which contains 64 features $F_0, \cdots, F_{63}$. As a consequence, the subsequent features of the original byte sequence *IN* are shifted by 64 places. As a full Bloom filter contains 128 features, the overlap of common features within Bloom filters is minimal and thus the expected similarity score of the original and manipulated byte sequence. (BB12) propose two different such feature sequences, which may be used to generate false non-matches.

5.4.2 Preimage Resistance

Our second security consideration in this section is a short discussion of the preimage resistance property of `sdhash`. Although `sdhash` uses the 160 bit cryptographic hash function SHA-1, its preimage resistance only relies on 55 bits. In what follows we show how to improve this minor security issue.

Recall, after `sdhash` identified a feature, it is hashed using SHA-1 and divided into $5 \cdot 32$ bit sub hashes. Afterwards only 11 out of 32 bits are used to derive the bit within the Bloom filter. Hence, `sdhash` only uses 55 bits of the SHA-1 hash value which enables one attack vector – brute force.

To exarcerbate a preimage attack we suggest to first pad one bit at each sub hash, divide then the padded 33 bit string into 3 blocks of 11 bits, and finally XOR all three blocks. Then all 32 bits of the sub hash are used to determine its bit position in the Bloom filter. A sample is given in the following equation where sH_{new} is the new sub hash and sH_{int} the original padded one.

$$\begin{aligned} sH_{new} &= sH_{int} \oplus (sH_{int} >> 11) \oplus (sH_{int} >> 22) \\ sH_{new} &= sH_{new} \,\&\, 0x7FF \end{aligned}$$

As these are only low level operations this will not influence the performance of `sdhash` but increases the security.

5.4.3 Coverage

We summarize important results from (BB12) concerning the coverage of `sdhash`. In general hash functions are designed so that each bit influences the hash value, otherwise it might be possible that a modification is not discovered. (BB12) shows that only $\sim 80\%$ of all bits influence the similarity digest. A more detailed analysis of (BB12) reveals that there are a lot of overlaps / gaps between two consecutive features, i.e. a byte of the input stream either influences multiple features or none. However, both cases are undesirable.

6 Conclusion

We did an implementation evaluation and discussed some security issues of `sdhash` as proposed by Roussev. The implementation lacks through inconsistencies between algorithm and description and shows different design errors wherefore we proposed possible solutions. Concerning the security aspects, it is possible to beat down the similarity score to approximately 28 with a time complexity $O(1)$.

Besides the design errors we discovered some further weaknesses and proposed improvements to make `sdhash` more secure and reliable.

All in all `sdhash` is much more robust than `ssdeep` but in order to eliminate the *Bloom Filter Shifting* issue, the comparison function should be adapted.

7 Acknowledgment

This work was partly supported by the EU (integrated project FIDELITY, grant number 284862).

References

[BB11a] Harald Baier and Frank Breitinger. Security Aspects of Piecewise Hashing in Computer Forensics. *IT Security Incident Management & IT Forensics (IMF)*, pages 21–36, May 2011.

[BB11b] Frank Breitinger and Harald Baier. Performance Issues about Context-Triggered Piecewise Hashing. In *3rd ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*, volume 3, October 2011.

[BB12] Frank Breitinger and Harald Baier. Properties of a Similarity Preserving Hash Function and their Realization in sdhash. *2012 Information Security for South Africa (ISSA 2012)*, August 2012.

[Blo70] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13:422–426, 1970.

[Bre11] Frank Breitinger. Security Aspects of Fuzzy Hashing. Master's thesis, Hochschule Darmstadt, February 2011. last accessed on 2012-07-05.

[CW08] L. Chen and G. Wang. An Efficient Piecewise Hashing Method for Computer Forensics. *Workshop on Knowledge Discovery and Data Mining*, pages 635–638, 2008.

[Kor06] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Forensic Research Workshop (DFRWS)*, 3S:91–97, 2006.

[MOV97] A. Menezes, P.v. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[NIS12] NIST. National Software Reference Library, May 2012.

[RCBR06] Vassil Roussev, Yixin Chen, Timothy Bourg, and Golden G. Rechard. md5bloom: Forensic filesystem hashing revisited. *Digital Investigation 3S*, pages 82–90, 2006.

[Riv92] Ron Rivest. The MD5 Message-Digest Algorithm. 1992.

[Rou09] Vassil Roussev. Building a Better Similarity Trap with Statistically Improbable Features. *42nd Hawaii International Conference on System Sciences*, 0:1–10, 2009.

[Rou10] Vassil Roussev. Data Fingerprinting with Similarity Digests. *Internation Federation for Information Processing*, 337/2010:207–226, 2010.

[Rou11] Vassil Roussev. An evaluation of forensic similarity hashes. *Digital Forensic Research Workshop*, 8:34–41, 2011.

[Rou12] Vassil Roussev. Scalable Data Correlation. *International Conference on Digital Forensics (IFIP WG 11.9)*, January 2012.

[RRM07] Vassil Roussev, Golden G. Richard, and Lodovico Marziale. Multi-resolution similarity hashing. *Digital Forensic Research Workshop (DFRWS)*, pages 105–113, 2007.

[SHS95] SHS. Secure Hash Standard. 1995.

[SLC$^+$09] Kimin Seo, Kyungsoo Lim, Jaemin Choi, Kisik Chang, and Sangjin Lee. Detecting Similar Files Based on Hash and Statistical Analysis for Digital Forensic Investigation. *Computer Science and its Applications (CSA '09)*, pages 1–6, December 2009.

[Tri02] Andrew Tridgell. Spamsum. Readme, 2002. last accessed on 2012-07-05.