Similarity Preserving Hashing: Eligible Properties and a New Algorithm MRSH-v2

Frank Breitinger and Harald Baier

da/sec Biometrics and Internet Security Research Group
Hochschule Darmstadt, Darmstadt, Germany
{frank.breitinger,harald.baier}@h-da.de

Abstract. Hash functions are a widespread class of functions in computer science and used in several applications, e.g. in computer forensics to identify known files. One basic property of cryptographic Hash Functions is the avalanche effect that causes a significantly different output if an input is changed slightly. As some applications also need to identify similar files (e.g. spam/virus detection) this raised the need for *Similarity Preserving Hashing*. In recent years, several approaches came up, all with different namings, properties, strengths and weaknesses which is due to a missing definition.

Based on the properties and use cases of traditional Hash Functions this paper discusses a uniform naming and properties which is a first step towards a suitable definition of Similarity Preserving Hashing. Additionally, we extend the algorithm MRSH for Similarity Preserving Hashing to its successor MRSH-v2, which has three specialties. First, it fulfills all our proposed defining properties, second, it outperforms existing approaches especially with respect to run time performance and third it has two detections modes. The regular mode of MRSH-v2 is used to identify similar files whereas the f-mode is optimal for fragment detection, i.e. to identify similar parts of a file.

Keywords: Digital forensics, Similarity Preserving Hashing, fuzzy hashing, MRSH-v2, properties of Similarity Preserving Hashing.

1 Introduction

Within the area of computer forensics investigators are overwhelmed with digital data. Traditional books, photos, letters and long-playing records (LPs) turned into ebooks, digital photos, email and mp3. In order to handle this amount of data, investigators need methods to automatically identify suspect files (e.g., images of child abuses). Normally the proceeding is quite simple: the investigator computes hash values (fingerprints) of all files which he finds on a storage medium and performs database lookups, e.g., within the widespread National Software Reference Library (NSRL, [1]). Besides finding exact duplicates using a cryptographic Hash Function, it is also necessary to uncover similar files using *Similarity Preserving Hashing*.

M. Rogers and K.C. Seigfried-Spellar (Eds.): ICDF2C 2012, LNICST 114, pp. 167–182, 2013.

Cryptographic Hash Functions are well established and thus a clear definition exists. This is in contrast to Similarity Preserving Hashing where the (preferable) properties are unclear although there are existing approaches like `ssdeep` from Kornblum ([2], 2006), `sdhash` from Roussev ([3], 2010) or `bbHash` from Breitinger et al. ([4], 2012).

There are two main contributions of this paper. On the one hand it discusses foundations for *Similarity Preserving Hashing* as a first step towards a definition of Similarity Preserving Hashing. Our discussion comprises a uniform naming, five important properties, which we consider to be eligible to be part of a definition of Similarity Preserving Hashing and two use cases to understand our proposals. On the other hand, we present a new version of the existing Similarity Preserving Hashing algorithm `MRSH` ([5]). We show that our version `MRSH-v2` is compliant with the proposed defining properties. Moreover, practical tests and a theoretical analysis reveal that `MRSH-v2` outperforms existing approaches with respect to run time performance and allows to detect similar files and file fragments.

The rest of the paper is organized as follows: At first we present properties (Sec. 2.1), use cases (Sec. 2.2) and Bloom filters (Sec. 2.3) for Similarity Preserving Hashing in the section on foundations (Sec. 2). Next, we shortly review related work in Sec. 3. Sec. 4.1 introduces the concepts of the existing algorithm `MRSH`, while Sec. 4.2 deals with our improved variant `MRSH-v2`. Based on the foundations and the new algorithm, Sec. 5 evaluates `MRSH-v2` and presents experimental results. Sec. 6 concludes our paper.

2 Foundations of Similarity Preserving Hashing

The topic uses the term Similarity Preserving Hashing that is also known as similarity digest, fuzzy Hash Function or similarity preserving Hash Function. Within a first step we like to come up with a uniform naming.

Each existing approach consists of two sub-functions: one for generating hash values / fingerprints[1] and one for comparing them. Thus, Similarity Preserving Hashing[2] (abbreviated SPH) consists of a

similarity preserving hash function, (abbreviated SPHF) which is a function / algorithm to create a hash value / fingerprint and a
comparison function, (abbreviated CP) that outputs a similarity score for two hash values / fingerprints.

In contrast to cryptographic Hash Functions, we do not expect a fixed-length hash value (more in Sec. 2.1) and therefore the term *hashing* might be a little bit confusing. Nevertheless, as some uses cases are almost identical to traditional Hash Functions, we agreed on this term.

Talking about the similarity of files, one usually distinguishes between byte level similarity and semantic similarity. In what follows we treat each input as a

[1] The term fingerprint or hash value is due to cryptographic Hash Functions.
[2] We also use the long term *Approach for Similarity Preserving Hashing*.

byte sequence and consider byte level similarity. Thus, when talking about the similarity of two files, we generally talk about the similarity of the underlying byte sequences.

2.1 Properties of Similarity Preserving Hash Functions

We bring five properties for Similarity Preserving Hashing into being that are discussed in the following and later used as a benchmark. This results from the necessity that we do not have a clear definition right now. Inspired by cryptographic Hash Functions we distinguish between *general properties* (P1-P3) and *security properties* (P4-P5). Finally this sections briefly discusses the impact of the properties to the existing algorithms: `ssdeep` ([2]), `sdhash` ([3]), and `bbhash` ([4]).

General Properties for SPH

P1 **- Compression.** The output (hash value) of a SPHF is much smaller than the input (the shorter the better). In contrast to traditional Hash Functions we do not expect a fixed-length hash value. The reason for compression is two-spread. First, a short hash value is space-saving and second, the comparison of small hash values is faster.

P2 **- Ease of Computation.** Generating a hash value is 'fast' in practice for all kinds of inputs. This is comparable to the property of a classical hash function like SHA-1. It is obvious that ease of computation is a prerequisite for a SPHF to be usable in practice.

P3 **- Similarity Score.** In order to compare two hash values we need a 'comparison function'[3]. Input of the comparison function are two hash values, its output is a value from 0 to X, where X is the maximum match score. A match score of X indicates that the hash values are identical or almost identical, which implies that the input files are identical or almost identical, too. Preferably the similarity score is between 0 and 100 and represents a percentage value. If the comparison function is linear, it is easy to map the match score in $[0, X]$ to the corresponding value in $[0, 100]$.

Security Properties for SPH

P4 **- Coverage.** Every byte of an input is expected to influence the hash value. We remark that this property is formulated in a statistical way. It means that given a certain byte of the input the probability that this byte does not influence the input's digest is insignificant. Otherwise it is possible that small changes will be uncovered. This property is in conformance with the corresponding characteristic of classical hash functions.

P5 **- Obfuscation Resistance.** It is the difficulty to achieve a false negative / non-match. For instance, let f be a file e.g., a suspect file. Then it should be difficult to manipulate f to f' so that a comparison yield a non-match but they are still very similar.

[3] In most cases the comparison of similarity preserving hash values is more complex than for traditional hashes where we can use the Hamming distance.

2.2 Use Cases

This section demonstrates that within the area of Similarity Preserving Hashing both mentioned security properties are sufficient. Assuming the applications computer forensics, malware or junk mail detection, which are reasonable in our mind, we identified two common aspects: *file identification* and *fragment detection*, which are explained in the following.

File Identification. The mentioned applications mostly use databases containing hash values of known inputs e.g., it stores fingerprints of known malware or files from previous investigations. Later on, if the application is faced with an unknown input, it generates the fingerprint and performs database lookups. Depending on the underlying database, this processing categorizes files into the categories: known-to-be-good, known-to-be-bad and unknown input.

Blacklisting. The main challenge for an active adversary is to hide suspect (=known-to-be-bad) files from an automatic identification through a 3^{rd} party e.g., investigators, anti-virus software or junk mail scanner. As this is easily feasible for cryptographic Hash Functions by flipping a single bit, it should not be possible within the area of SPH. This concludes *P5 - obfuscation resistance*.

Whitelisting. Within the area of whitelisting we believe that cryptographic Hash Functions are the mean of choice. For instance, an active adversary is able to manipulate the *ssh daemon* of an operation system and include a backdoor. Thus, the original file and the modified file are still very similar although it is a malicious ssh daemon.

As whitelisting is out of scope we argue that traditional security properties like preimage-resistance, second preimage-resistance and collision resistance are not necessary for SPH - no one likes to manipulate a file to look like a suspect file.

Fragment Detection. Another opportunity for SPH on the binary level is its ability to identify file fragments e.g., 200kiB out of 1MiB. One possible scenario is the computer forensics. For instance, an investigator receives a hard disk which is formatted in quick-mode. Thus he is only able to analyze the low level hdd blocks. If a match is identified, a known-to-be-bad files were present before the deletion. In the best case he even may recover the file.

2.3 Bloom Filters and the Comparison Function

A very promising way to represent hash values for SPH are Bloom filters because they allow a fast comparison using the Hamming distance. According to this, we briefly describe Bloom filters in general followed by a possible *comparison function* (CP) as mentioned in Sec. 2.

Bloom Filters. A Bloom filter is an array of m bits (all set to zero) and used to represent a set S of n elements. In order to 'insert' an element s into the filter, k independent Hash Functions are used where each Hash Function outputs a value

between 0 and $m-1$. For instance, to insert s we compute $h_0(s), h_1(s), \ldots h_{k-1}(s)$ where each h outputs a value between 0 and $m-1$. Thus, each Hash Function sets the corresponding bit within the Bloom filter.

To answer the question if s' is in S, we compute $h_0(s'), h_1(s'), \ldots h_{k-1}(s')$ and look if the bits at the corresponding positions are set to one. If all bits are set to one, s' is assumed to be within S with a high probability. Otherwise, if at least one bit is set to zero, we know that s' is not within S.

Comparison Function for Bloom Filters. This paragraph explains Roussev's idea ([3,6]) for a CP. The proceeding how to obtain a set S out of the input is explained later in Sec. 4.1. Hence, the rest of this section only explains how we compare Bloom filters.

Let bf, bf' be two Bloom filters, let $|bf|$ denote the number of bits set to one within a Bloom filter and let e be the amount of bits in common ($e = |bf \cap bf'|$). 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. If $e \leq C$, then the similarity score is set to zero.

C is determined as follows

$$C = \alpha \cdot (E_{max} - E_{min}) + E_{min} \tag{1}$$

where α is set to 0.3 [4], 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

$$E_{max} = \min(|bf|, |bf'|). \tag{2}$$

As described in Sec. 2.3, k denotes the amount of hash functions and m the size of a Bloom filter in bits. Furthermore, let $\overline{bf}$ denote the amount of elements within a Bloom filter and $p = 1 - 1/m$ the probability that a certain bit isn't set to one when inserting a bit. Thus

$$E_{min} = m \cdot (1 - p^{k \cdot \overline{bf}} - p^{k \cdot \overline{bf'}} + p^{k \cdot (\overline{bf} + \overline{bf'})}) \tag{3}$$

is an estimation of the amount of expected common bits set to one in the two Bloom filters bf, bf' by chance. In order to receive a similarity score we use

$$SF_{score}(bf, bf') = \begin{cases} 0, & \text{if } e \leq C \\ [100 \frac{e-C}{E_{max}-C}], & otherwise\ . \end{cases} \tag{4}$$

Due to different file sizes, it might be possible that $|S|$ is very large and all bits within bf [5] would be set to one. To overcome this issue, we create a new Bloom filter if $\overline{bf} = BF_{max}$. Hence, the final hash value is not a single but a list of Bloom filters. If we'd like to compare them, it is an all-against-all comparison of Bloom filter sequences.

[4] This is done by best practice.

[5] The size m of a Bloom filter is fixed.

Let $SD_1 = \{bf_1, bf_2, \ldots bf_s\}$ and $SD_2 = \{bf'_1, bf'_2, \ldots bf'_r\}$ the Bloom filter sequences (hash values) 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

$$SD_{score}(SD_1, SD_2) = \frac{1}{s} \sum_{i=1}^{s} \max_{1 \leq j \leq r} SF_{score}(bf_i, bf'_j) \ . \tag{5}$$

3 Related Work

The beginning of similarity preserving hashing was in 2002 by Harbour who developed `dcfldd`[6] which extends the well-known disk dump tool `dd`. `dcfldd` is also called *block based hashing* as it divides an input into fixed-size blocks, hash each block separately and concatenate all hash values. In order to overcome this approach it is sufficient to insert / remove one byte in the beginning. Thus the offset of each block shifts and the resulting hash value is completely different.

Context triggered piecewise hashing (abbreviated CTPH) can be considered as an advancement of `dcfldd` which fixes the alignment weakness. It was presented in [2] by Kornblum in 2006 and is based on a spam detection algorithm of [7]. The basic idea is equal to the aforementioned block based hashing but instead of dividing an input into blocks of a fixed length, an input is divided based on the current context of 7 bytes.

As CTPH was the first Approach for Similarity Preserving Hashing, it was improved in the upcoming years by [8,5,9,10] with respect to both, efficiency and security. In 2011 [11,12] did a security analysis of CTPH where the authors focused on blacklisting and whitelisting and came to the conclusion that `ssdeep` fails in case of an active adversary.

Similarity Digest Hashing is a completely different Approach for Similarity Preserving Hashing and was presented in 2010 by Roussev ([3]) including a prototype called `sdhash`. Instead of dividing an input into pieces, `sdhash` identifies "statistically-improbable features" ([13]) using an entropy calculation.

These characteristic features, a sequence of length 64 bytes, are then hashed using the cryptographic Hash Function SHA-1 ([14]) and inserted into a Bloom filter ([15]). Hence, files are similar if they share identical features.

Comparison [16] 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 behavior". A security analysis ([17]) approved this statement but also showed some peculiarities and weaknesses of `sdhash`.

[6] `http://dcfldd.sourceforge.net`; visited 02.05.2012

4 Multi-Resolution Similarity Hashing (MRSH)

Roussev et al. ([5]) present a powerful variation of `ssdeep` called *multi-resolution similarity hashing* (abbreviated MRSH) that slided into obscurity. Therefore Sec. 4.1 explains the concept of the original algorithm and Sec. 4.2 shows changes to increase the performance.

4.1 Foundations of MRSH

As briefly described within Sec. 3 the main idea of `ssdeep` is to divide an input in several chunks based on the current context of 7 bytes where an input is a byte sequence. As MRSH is based on `ssdeep`, this algorithm is explained first.

Let an input IN of length L be given as a byte sequence $b_0 b_1 \dots b_{L-1}$. In order to identify the end of a chunk (i.e., to divide the input into blocks), `ssdeep` uses a window of size 7 bytes that moves through the whole input, byte for byte. At each position p ($0 \leq p < L$) within IN the window contains a byte sequence $BS_p = b_{p-6} b_{p-5} \dots b_p$ which serves as input for a pseudo random function PRF. We denote this by $PRF(BS_p)$. If $PRF(BS_p)$ hits a certain value, the end of the current chunk is identified and b_p is called a *trigger point*. The subsequent chunk starts at byte b_{p+1} and ends at the next trigger point or EOF.

In order to define a hit for $PRF(BS_p)$, MRSH uses a fixed modulus called *blocksize* b e.g., 256 [7]. Thus, if $PRF(BS_p) \equiv -1 \bmod b$ then b_p is a trigger point and the algorithm identified the end of the chunk. If PRF outputs equally distributed values, the probability of a hit is reciprocally proportional to b and therefore the average chunk size should be b bytes[8].

In contrast to `ssdeep` which uses an algorithm called `rolling_hash`[9] for PRF, MRSH uses the polynomial Hash Function `djb2`[10] over the 7 byte window which is shown in Algorithm 1. For each window (at each position p within IN) the window needs to be computed.

Algorithm 1. Polynomial Hash Function `djb2`

```
unsigned long hash = 5381
int c
for i = 0 → 6 do                    ▷ Run through all bytes within the window
    c = BS[i]
    hash = ((hash<< 5) + hash) + c;                          ▷ 33 · hash + c
end for
return hash
```

[7] `ssdeep` used a variable modulus based on the file size.

[8] Therefore this modulus is called *blocksize*.

[9] This function is a variation of Adler-32; `http://en.wikipedia.org/wiki/Adler-32`; visited 04.06.2012

[10] `http://www.cse.yorku.ca/~oz/hash.html`; visited 21.05.2012

The biggest difference between `ssdeep` and MRSH is the hash value representation. `ssdeep` uses the non-cryptographic Hash Function FNV ([18]) to hash each chunk. For each chunk it uses the least significant 6 bits of the FNV hash and concatenates all of them. Thus the final hash value is a Base64 sequence.

MRSH works completely different. All identified chunks build the set S which is used as basis for the hash value generation using Bloom filters (see Sec. 2.3). Instead of using k different Hash Functions, MRSH "take[s] the MD5 hash and split[s] it into four 32-bit numbers and take[s] the least significant 11 bits from each part" ([5]). For instance, imagine the least significant 11 bits are $010\ 1000\ 1010 = 0x28A = 650$, thus the bit at position 650 within the Bloom filter is set to one. Having 4 sub-hashes, each chunk sets 4 bits within the Bloom filter. After inserting $BF_{max}(=256)$ chunks into a Bloom filter, it reaches its maximum and a new filter is created. Hence, the final hash value is a list of Bloom filters.

As stated before, b is the approximate length of a chunk. In comparison to `ssdeep`, MRSH uses a *minimum chunk size* which is $\frac{1}{4}$ of the chunk size b. Thus, whenever a trigger point is discovered the next $\frac{b}{4}$ bytes are skipped for PRF, so the chunk is guaranteed a minimum size of $\frac{b}{4}$.

4.2 MRSH Version 2

In the following we present an updated version of MRSH called `MRSH-v2`. Generally speaking `MRSH-v2` uses its precursor as a base frame but with some accommodations based on the aforementioned properties.

PRF. We impose two important requirements on a pseudo random function (PRF). First, it has to be very efficient with respect to its computation time as it is invoked for roughly every byte of the input. Second its output should behave pseudo randomly.

In his version of MRSH Roussev changed the PRF from `rolling_hash` to `djb2` which he motivates with respect to performance. As shown in Algorithm 1, `djb2` should be quite fast. However, our tests presented in Sec. 5.2 show different results. The point is, although `djb2` looks less complex, it needs to compute the hash value over the whole window at each time (7 loops per window) whereas the original version (rolling hash) is able to remove the last byte and add the new one to the hash value (only one loop per window).

[5, Sec. 3] compares the randomness of `djb2` with MD5 and concludes that `djb2` totally fulfills the expectations of a fast PRF. However, [11, Sec. V] shows that `rolling_hash` is suitable for `MRSH-v2`, too.

As outcome of both requirements we decided to make use of the original rolling hash as PRF in our algorithm `MRSH-v2`.

Chunk Hash Function. The motivation to change the chunk Hash Function from FNV to MD5 is that "FNV is not a collision-resistant function and has some known collision issues [...] especially for inputs with lower entropy which would present a serious problem for simple hashes" ([5]).

The latter argument is in contrast to [18] where it says that "the high dispersion of the FNV hashes makes them well suited for hashing nearly identical strings". Furthermore we argue that collision resistance is not necessary as discussed in Sec. 2.1. Moreover MRSH reduces the MD5 hash value from 128 bits to 44 bits in order to insert it into the Bloom filter. Thus, the hash looses its cryptographic properties.

Due to these facts our version uses FNV-1a (64 bit) and is therefore faster (some measurement results are given in Sec. 5.1).

Minimum Chunk Size. A minimum chunk size comes with two improvements. First, it overcomes one of the main attacks on `ssdeep` presented in [11] called 'adding trigger points'. Second, it increases the performance as the PRF needs not to be computed at each offset within the input sequence. A drawback is that some details may lost. This is the case if two subsequent trigger points have a distance of at most $\frac{b}{4} - 1$.

We illustrate this characteristics on base of an extreme example. We assume that the input byte sequence has a trigger point every $\left(\frac{b}{4} - 1\right)$-th byte. They are denoted by $t_0, t_1, t_2, ...$ Then every second trigger point is skipped (only trigger points with an even index are used). Removing the first trigger point t_0 from the input results in considering the trigger points $t_1, t_3, ...$ yielding a fundamental different hash value.

However, for performance reasons we agree on the same minimum chunk size as used in MRSH, $\frac{b}{4}$.

Bloom Filters. MRSH uses Bloom filters of size $m = 2048$ and inserted $BF_{max} = 256$ chunks each setting 4 bits within the Bloom filter which is in contrast to our implementation. Within MRSH a maximum of 1024 bits could be set and each Bloom filter could represent approximately 65,536 byte (using the blocksize $b = 256$).

The final hash value of `MRSH-v2` is mostly based on the settings identified for `sdhash` in [3,6]. Thus the Bloom filter size is still $m = 2048$ bits but we changed $BF_{max} = 256$ and $k = 5$ (five sub-hashes). The maximum is therefore 800 bits and one Bloom filter could represent approximately 40,960 byte (more see Sec. 5.1). Also MRSH has a better compression, `MRSH-v2` has a better false positive rate as shown in Sec. 5.6.

In order to insert the chunk hash value into a Bloom filter, we use the least significant $k \cdot \log_2(m)$ bits (MRSH divides the chunk hash values). As a consequence our chunk Hash Function needs at least so many bits which is fulfilled by FNV-1a using the default setting $k = 5$, $m = 2048$. A performant proceeding is given in Algorithm 2. In addition the design of `MRSH-v2` allows to change the parameters like k, m or the chunk Hash Function.

5 Experimental Results and Evaluation

The following sections discuss the properties from Sec. 2.1 with respect to our algorithm. Furthermore we compare `MRSH-v2` to `sdhash`, `bbHash` and `ssdeep`.

Algorithm 2. Insertion of a chunk hash into a Bloom filter

h is the chunk hash value

```
k = 5                                          ▷ Amount of sub-hashes
m = 0x7FF                                      ▷ m = 2048 − 1
shiftOps = 11                                  ▷ Calculated by log2(m + 1)

for i = 0 → k − 1 do                           ▷ Create k sub hashes
    bit = (h >> (shiftOps · i)) & m
    setBitInBloomFilter(bit)
end for
```

All experimental tests were performed on a 64Bit Mac OS X with a 2.4 GHz Intel Core 2 Duo processor.

5.1 P1 - Compression

Due to the design of `ssdeep` and thus of `MRSH-v2`, the hash value length depends on the blocksize b, the amount of chunks per Bloom filter BF_{max} and the size of a Bloom filter m (in bits). Each Bloom filter represents approximately $BF_{max} \cdot b$ bytes of a given input and thus the compression ratio is $\frac{m}{8} \cdot \frac{1}{BF_{max} \cdot b}$.

As discussed in Sec. 4.2 we use $m = 2048$ bits and $BF_{max} = 160$. Assuming these values, the compression ratio is $\frac{2048}{8} \cdot \frac{1}{160 \cdot b} = \frac{8}{5 \cdot b}$ for $b > 0$ and therefore adjustable by changing b. Table 1 shows the proportion between blocksize b and the expected hash value length. For instance, by default we set $b = 320$ and thus the compression ratio is at 0.5%.

Table 1. Proportion between blocksize b and the hash value length in percent

b	128	160	256	320	512
expected length in %	1.250	1.000	0.625	0.500	0.313

`ssdeep` produces outputs having at most 100 Base64 characters. This rather good compression implies a security drawback as discussed in [11]. Put simply, if there are too many chunks, the last chunks are combined into one large one. Due to the poor result in the security analysis we neglect `ssdeep` and focus on two other approaches.

The hash values of `bbHash` and `sdhash` are proportional to the input length, where the proportionality factor is 0.5% and 3.3%, respectively. However, the performance of `bbHash` isn't acceptable wherefore we come up with the following classification: 1. `MRSH-v2` (0.5%), 2. `sdhash` (3.3%), 3. `bbHash` (0.5%) and `ssdeep`.

5.2 P2 - Ease of Computation

This section is roughly divided into two parts. First, we analyze MRSH-v2 itself as the performance of MRSH-v2 is based on two issues: the pseudo random function (PRF) and the chunk hash function. Second, we compared our implementation against other existing algorithms.

All tests are based on a 500MiB file from /dev/urandom.

PRF. In the following we show that the rolling_hash is faster than djb2. In order to test both algorithms we separated them, run them 'stand-alone' and used all optimizations modes of the gcc compiler[11]. Of course, both versions are improved for performance, e.g., the struct of the rolling_hash from ssdeep was removed. The result is given in Table 2 [12].

Table 2. Performance of two possible pseudo random function (PRF)

optimization mode	-	O1	O2	O3
djb2	23.620s	11.021s	1.236s	1.241s
rolling hash	9.835s	4.315s	1.138s	1.085s
djb2 / rolling hash	2.402	2.554	1.086	1.143

Actually we cannot explain these serious differences. We recognized them by comparing

- djb2 (8.532s) and rolling_hash (3.808s) within MRSH-v2 and
- djb2 (1.241s) and rolling_hash (1.085s) as 'stand-alone'.

Chunk Hash Function. As discussed in Sec. 4.2 we decided for FNV-1a instead of the cryptographic Hash Function MD5[13]. To test MD5 we took a library provided by OpenSSL and used an optimized version of FNV-1a. The test is focused on the algorithm time (read-in time is neglected) and solved using the clock()-function from C++. The result is 10^{-6}s from FNV vs. 1.354s of MD5. Using these functions within MRSH-v2 it is 5.235s vs. 6.569s.

The second part of this section is the comparison against other existing algorithms. We skipped bbHash as its performance is not acceptable and focused on ssdeep and sdhash. Furthermore we also included SHA-1 as a reference time. All times were measured using the time command and the algorithm CPU-time (time denotes this by user-time). The results are shown in Table 3.

As expected SHA-1 outperforms every similarity preserving Hash Function and sdhash is the slowest one due to the high complexity. The difference between MRSH-v2 and ssdeep relies on the *minimum chunk size* which allows to skip some calculations and an improved implementation of the rolling_hash.

[11] http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html; visited 21.6.2012

[12] We do not measure the time it takes to read the file into a buffer.

[13] Due to the minimum hash value length of 55 bits (see Sec. 4.2, Bloom filters) it is not possible to use djb2 which would be even faster.

[14] We used sdhash with 2 threads.

Table 3. Performance comparison of similarity preserving Hash Functions and SHA-1

	SHA-1	MRSH-v2	sdhash 2.0	sdhash 2.0[14]	ssdeep 2.8
runtime	2.549s	5.235s	28.641s	28.493s	7.131s
algorithm / SHA-1	1.000	2.054	11.236	11.178	2.798

5.3 P3 - Similarity Score

Our algorithm MRSH-v2 makes use of the Bloom filter comparison algorithm from Sec. 2.3 and findings from [17, Sec. 4.3] including an own improvement. In a first step we show that the existing algorithm is well suited for fragment detection, but has drawbacks for file similarity detection. As a result we recommend to use the original comparison function of sdhash for fragment detection. However, we modify the algorithm to decide about file similarity, too.

Fragment Detection. We discussed the original comparison algorithm in Sec. 2.3. We explain its shortcomings in what follows based on an example (a generalization is easy). As a result MRSH-v2 makes use of this comparison algorithm for fragment detection only.

Let f and f' be two files where f' is a fragment of f, e.g. the first 25% of f. Let $SD = \{bf_1, bf_2\}$ be the hash value of f and let $SD' = \{bf_1^*\}$ be the hash value of f' where $|bf_1^*| < |bf_1|$.

To receive the similarity score we first have to identify the best matching Bloom filters where the filter similarity is identified by Eq. (4). Recall,

$$SF_{score}(bf, bf') = \begin{cases} 0, & \text{if } e \leq C \\ [100\frac{e-C}{E_{max}-C}], & \text{otherwise .} \end{cases} \tag{4}$$

In case of fragments we have $e = E_{max}$ as $e = |bf \cap bf'| = |bf'|$ and $E_{max} = \min(|bf|, |bf'|) = |bf'|$. Thus the $SF_{score}(bf_1^*, bf_1) = 100$.

Knowing the best matching Bloom filters, the final similarity score is generated using Eq. (5). Recall,

$$SD_{score}(SD_1, SD_2) = \frac{1}{s} \sum_{i=1}^{s} \max_{1 \leq j \leq r} SF_{score}(bf_i, bf_j') \ , \tag{5}$$

where $s = |SD'|$, $r = |SD|$ (s needs to be smaller). As $s = 1$, the SD_{score} is $\frac{100}{1}$. To sum it up, we compared two obviously different hash values and resulted in a 100% match score. As f' is a fragment of f this algorithm is perfect for *fragment detection*.

File Similarity Detection. Besides fragments we also interested in identifying similar files. Taking the aforementioned example concerning f, f' we expect a file similarity of 25%, if $|bf_1|, |bf_2|$ are at their maximum and $|bf_1^*| = \frac{bf_1}{2}$. In order to achieve file similarity, there are two adaptations:

1. As proposed in [17] MRSH-v2 makes use of a new function $E'_{max} = \max(|bf|, |bf'|)$ in Eq. (4) (the min function is replaced by the max function).
2. Additionally we replace $\frac{1}{s}$ by $\frac{1}{r}$ within Eq. (5). As $s \leq r$ all Bloom filters are considered (in contrast to [17], where only the first s Bloom filters are relevant).

To receive the final similarity score, we first have to generate the SF_{score} defined by $SF_{score} = [100\frac{e-C}{E'_{max}-C}]$. Thus we need to determine e, E'_{max} and C where

- $e = |bf_1 \cap bf_1^*| = |bf_1^*|$,
- $E'_{max} = \max(|bf_1|, |bf_1^*|) = |bf_1|$ and
- $C = \alpha \cdot (E_{max} - E_{min}) + E_{min}$ where
  - $E_{max} = \min(|bf_1|, |bf_1^*|) = |bf_1^*|)$ and
  - $E_{min} = m \cdot (1 - p^{k \cdot \overline{bf_1}} - p^{k \cdot \overline{bf_1^*}} + p^{k \cdot (\overline{bf_1} + \overline{bf_1^*})}) = 117.548$

We first estimate the amount of bits set to be one using the following

$$|bf_1| = m \cdot \left(1 - (1 - \frac{1}{m})^{k \cdot |bf_1|}\right) = 2048 \cdot (1 - 0.99951172^{5 \cdot 160}) = 662.386$$
$$|bf_1^*| = m \cdot \left(1 - (1 - \frac{1}{m})^{k \cdot |bf_1^*|}\right) = 2048 \cdot (1 - 0.99951172^{5 \cdot 80}) = 363.442$$

and calculate C by

$$C = 0.3(363.442 - 117.548) + 117.548 = 191.316$$

To sum it up, we result in

$$SF_{score} = [100\frac{e-C}{E'_{max}-C}] = 100 \cdot \frac{363.442 - 191.316}{662.386 - 191.316} = 100 \cdot \frac{172.126}{471.0698} = 36.539.$$

In the very last step we use the adopted version Eq. (5) (instead of $\frac{1}{s}$ we use $\frac{1}{r}$. Thus we have to divide SF_{score} by 2 and result in a final similarity score of 18.270.

Implementation. These properties allow to extend our algorithm to have two modes as listed in Fig. 1.

Regular mode (default setting) is used to identify the similarity between two files.

Fragment mode (use `-f` option) is the fragment mode and used to find smaller parts of a file.

```
$ dd if=/dev/urandom of=2MiB bs=1m count=2
$ split -b 512k 2MiB

$ ./mrsh-v2 2MiB xaa
Similarity of files 2MiB and xaa is: 27.113835

$ ./mrsh-v2 -f 2MiB xaa
Similarity of files 2MiB and xaa is: 99.417396
```

Fig. 1. A sample for fragment and similar file detection

5.4 P4 - Coverage

Full coverage means that every byte of an input should influence the output. By design all bytes influence the final hash value and therefore especially some greater random changes influence the final hash value. Recall, a high similarity score (e.g., 100) means that two inputs are very similar but it does not imply that they are completely identical.

`ssdeep` and `MRSH-v2` have a better (full) coverage compared to `sdhash`, as [17] shows that there are bytes which don't influence the similarity digest at all. `bbHash` claims to have a full coverage but this is not attest.

5.5 P5 - Obfuscation Resistance

Obfuscation resistance is the difficulty to achieve a non-match. Thus we roughly analyze the amount of changes an active adversary has to do in order to overcome this approach. However, this section does not replace a comprehensive security analysis.

The most obvious attack is to change one byte within each chunk which will change all chunk hash values. Recall, the chunk size in bytes is approximately the blocksize b. Let $b = 320$. Assuming a file of 1048576 bytes (=1 MiB), this result in $\frac{1{,}048{,}576}{320} = 3276.8$ changes. Due to the comparison algorithm it is not necessary to have changes within each chunk.

[17] showed the possibility of 'Bloom filter shifts'. It is possible to reduce the similarity score down to approximately 25 by inserting data at the beginning of a file. However, the authors also present a first idea to solve this issue which will be analyzed for the next upcoming version of `MRSH-v2`.

Nevertheless the possibility to make changes within a specific file depends on the file type. Generally we classify files in one of the following categories.

Locally sensitive file types (e.g., jpg, pdf, zip, exe) nearly impossible to manipulate at each position (e.g., [11] showed that the jpg-header allows changes). A flipped bit can have 'global' consequences such that the file is not readable

anymore. We believe that an active adversary will not overcome MRSH-v2 for these kind of types[15].

Locally non sensitive file types (e.g., txt, doc, bmp) are mostly small (e.g., doc, txt) and sometimes not so wide-spread (e.g., bmp). Manipulations only influence the local area e.g., changing a letter within a txt file. For small files, reducing b increases granularity of the hash value and force an attacker to do more changes. Of course there are also large doc-files but they mostly contain images (which give them their unique characteristic).

5.6 False Positive Rate

Within 4.2 we explained that we have to find a good trade-off between compression and false positive rate. Due to the changes from $[k = 4, BF_{max} = 256]$ to $[k = 5, BF_{max} = 160]$ we reduced the false positive rate

$$\left(1 - \left(1 - \frac{1}{m}\right)^{k \cdot BF_{max}}\right)^{k} = \left(1 - \left(1 - \frac{1}{2048}\right)^{4 \cdot 256}\right)^{4} = 0.0240 \qquad (6)$$

down to

$$\left(1 - \left(1 - \frac{1}{m}\right)^{k \cdot BF_{max}}\right)^{k} = \left(1 - \left(1 - \frac{1}{2048}\right)^{5 \cdot 160}\right)^{5} = 0.0035\ . \qquad (7)$$

which is a factor of approximately 7.

6 Conclusion

Currently there are no constant naming, definition or properties for Similarity Preserving Hashing which are necessary to classify them. But due to the increasing amount of data, it is necessary to rate different approaches. Thus this paper at hand presents 5 properties: 3 general properties and 2 security related properties. As a conclusion, the identified properties coincide only partially with traditional Hash Functions which comes due to the different use cases.

Additionally we improved an existing Approach for Similarity Preserving Hashing from 2007 with respect to performance and introduced MRSH-v2. We briefly compared it against other algorithms based on the properties. As a result MRSH-v2 outperforms existing algorithms with respect to performance. The hash value length is at 0.5% and only surpassed by ssdeep which failed a security analysis. As a highlight MRSH-v2 is the first algorithm that has two modes: fragment detection and similar file detection.

There are three next steps: First, we like to complete the functions of our implementation (e.g., read directory) . Second, a detailed security analysis of MRSH-v2 is needed. And third, we would like to implement sdhash using FNV and analyze the performance.

[15] We focused on the binary level and not on a semantic level where it is possible to rotate an image.

Acknowledgments. This work was partly funded by the EU (integrated project FIDELITY, grant number 284862) and supported by CASED (Center for Advanced Security Research Darmstadt).

We thank Mustafa Karabat for supporting us with programming and testing.

References

1. NIST, "National Software Reference Library" (May 2012), http://www.nsrl.nist.gov
2. Kornblum, J.: Identifying almost identical files using context triggered piecewise hashing. In: Digital Forensic Research Workshop (DFRWS), vol. 3S, pp. 91–97 (2006)
3. Roussev, V.: Data fingerprinting with similarity digests. In: Chow, K.-P., Shenoi, S. (eds.) Advances in Digital Forensics VI. IFIP AICT, vol. 337, pp. 207–226. Springer, Heidelberg (2010)
4. Breitinger, F., Baier, H.: A Fuzzy Hashing Approach based on Random Sequences and Hamming Distance. In: ADFSL Conference on Digital Forensics, Security and Law, pp. 89–101 (May 2012)
5. Roussev, V., Richard, G.G., Marziale, L.: Multi-resolution similarity hashing. In: Digital Forensic Research Workshop (DFRWS), pp. 105–113 (2007)
6. Roussev, V.: Scalable data correlation. International Conference on Digital Forensics (IFIP WG 11.9) (January 2012)
7. Tridgell, A.: Spamsum. Readme (2002), http://samba.org/ftp/unpacked/junkcode/spamsum/README
8. Chen, L., Wang, G.: An Efficient Piecewise Hashing Method for Computer Forensics. In: Workshop on Knowledge Discovery and Data Mining, pp. 635–638 (2008)
9. Seo, K., Lim, K., Choi, J., Chang, K., Lee, S.: Detecting Similar Files Based on Hash and Statistical Analysis for Digital Forensic Investigation. In: Computer Science and its Applications (CSA 2009), pp. 1–6 (December 2009)
10. Breitinger, F., Baier, H.: Performance Issues About Context-Triggered Piecewise Hashing. In: Gladyshev, P., Rogers, M.K. (eds.) ICDF2C 2011. LNICST, vol. 88, pp. 141–155. Springer, Heidelberg (2012)
11. Baier, H., Breitinger, F.: Security Aspects of Piecewise Hashing in Computer Forensics. In: IT Security Incident Management & IT Forensics (IMF), 21–36 (May 2011)
12. Breitinger, F.: Security Aspects of Fuzzy Hashing. Master's thesis, Hochschule Darmstadt (February 2011), https://www.dasec.h-da.de/offerings/theses/
13. Roussev, V.: Building a Better Similarity Trap with Statistically Improbable Features. In: 42nd Hawaii International Conference on System Sciences, pp. 1–10 (2009)
14. SHS, "Secure Hash Standard" (1995)
15. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13, 422–426 (1970)
16. Roussev, V.: An evaluation of forensic similarity hashes. In: Digital Forensic Research Workshop, vol. 8, pp. 34–41 (2011)
17. Breitinger, F., Baier, H., Beckingham, J.: Security and Implementation Analysis of the Similarity Digest sdhash. In: First International Baltic Conference on Network Security & Forensics (NeSeFo) (August 2012)
18. Noll, L.C.: Fowler / Noll / Vo (FNV) Hash (2001), http://www.isthe.com/chongo/tech/comp/fnv/index.html