mvHash-B - a new approach for similarity preserving hashing

Frank Breitinger, Knut Petter Åstebøl, Harald Baier, Christoph Busch
da/sec - Biometrics and Internet Security Research Group
Hochschule Darmstadt, Darmstadt, Germany
Email: {frank.breitinger, harald.baier, christoph.busch}@{h-da, cased}.de,
knutpettercom@gmail.com

Abstract

The handling of hundreds of thousands of files is a major challenge in today's IT forensic investigations. In order to cope with this information overload, investigators use fingerprints (hash values) to identify known files automatically using blacklists or whitelists. Besides detecting exact duplicates it is helpful to locate similar files by using similarity preserving hashing (SPH), too.

We present a new algorithm for similarity preserving hashing. It is based on the idea of majority voting in conjunction with run length encoding to compress the input data and uses Bloom filters to represent the fingerprint. It is therefore called `mvHash-B`. Our assessment shows that `mvHash-B` is superior to other SPHs with respect to run time efficiency: It is almost as fast as SHA-1 and thus faster than any other SPH algorithm. Additionally the hash value length is approximately 0.5% of the input length and hence outperforms most existing algorithms. Finally, we show that the robustness of `mvHash-B` against active manipulation is sufficient for practical purposes.

Keywords

Digital forensics; fuzzy hashing; similarity preserving hashing; run-length encoding; Bloom filter.

I. Introduction

Due to the increasing amount of digital data within forensic investigations, it is important to reduce the amount of data that needs to be inspected manually. A common approach using cryptographic hash functions proceeds as follows: Each file on a storage medium is hashed and the hash is compared to a reference database [1], [2]. Depending on the underlying database all files can be classified as known-to-be-good, known-to-be-bad or unknown files.

However, the security requirements of a cryptographic hash function imply the avalanche effect. As a consequence, no matter how similar two inputs are, the hash values differ by approximately 50%. In contrast to cryptographic hash functions, similarity preserving hashing (SPH) aims at keeping resemblance by mapping similar inputs to similar hash values.

Similarity may be decided on different abstraction layers. While semantic similarity 'understands' the contents of an input, byte level similarity does not have this property. The latter approach has the key advantages of being faster and applicable to a broader range of input data (e.g. fragments of files, clusters, partitions). SPH using majority vote and Bloom filters (`mvHash-B`) is a byte level SPH, i.e. inputs are similar if they have similar underlying byte sequences.

A typical use case of byte level SPH functions are the automatic detection of known-to-be-bad files (blacklisting) in a forensic investigation, where the investigator has to deal with a huge amount of data. Eligible properties of SPH functions are thus run time efficiency, memory consumption, or manipulation robustness (see [3]). However, existing byte level SPH functions like `ssdeep` [4], `sdhash` [5], or `bbHash` [6] are either slow, their fingerprints are 'large' (e.g. 3.3% of the input length) or they may be easily fooled by an active adversary.

In this paper we propose a new method for SPH, which calculates its fingerprint in three steps: First, majority voting is used to map every byte of the input data to either $0x00$ or $0xFF$. Then run length encoding compresses these sequences of $0x00$s or $0xFF$s bytes. Finally, the run length encoding is inserted into Bloom filters to represent the actual fingerprint. We call our approach `mvHash-B`.

We evaluate `mvHash-B` by comparing it to existing byte level SPH. Our result is that `mvHash-B` is almost as fast as SHA-1 and thus faster than any other SPH algorithm. Its fingerprint length is approximately 0.5% of the input length and hence outperforms most existing algorithms. Concerning the security and active manipulation, a detailed security analysis is missing.

The rest of this paper has the following structure. In Sec. II, related work is presented, in Sec. III, the design of `mvHash-B` is described. Sec. IV is an assessment of our implementation followed by the last section which concludes our work.

II. Related Work

Currently we are aware of five SPH implementations: `dcfldd` [7], `ssdeep` [4], `sdhash` [5], `bbHash` [6] and `MRSH-v2` [3].

`dcfldd` or block-based hashing was developed by Nicholas Harbour in 2002 [7]. The algorithm splits an input into blocks of fixed size, hashes each block separately and concatenates all block hashes to a final hash. The weakness of `dcfldd` is that it is not alignment robust [8, p. 388] which means that inserting / deleting bytes shifts the borders of all blocks and results in a different hash value.

Context triggered piecewise hashing (CTPH) can be considered as an advancement over `dcfldd`, which fixes the alignment weakness. It was presented in [4] by Kornblum in 2006 and is based on a spam detection algorithm of [9]. 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 SPH function, it was improved in the upcoming years by [10], [11], [12], [13] with respect to both, efficiency and security. In 2011 [1] conducted 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 SPH approach and was presented in 2010 by Roussev [5] including a prototype called `sdhash`. Instead of dividing an input into pieces, `sdhash` identifies "statistically-improbable features" [14] using an entropy calculation.

These characteristic features, sequences of length 64 bytes, are then hashed using the cryptographic hash function SHA-1 [15] and inserted into a Bloom filter [16]. Hence, files are similar if they share identical features.

A Comparison of `sdhash` and `ssdeep` in [17] 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 [1] approved this statement but also showed some peculiarities and weaknesses of `sdhash`. Nevertheless, `sdhash` has also some drawbacks as presented in [18].

`bbHash` as introduced by Breitinger et al. in [6] was presented recently but does not reach practical relevance due to its unsuitable run time efficiency. A 10 MiB file needs nearly 2 minutes.

III. Design of `mvHash-B`

The aim of this section is to explain the design and the underlying idea of our new similarity preserving hash function. Our hash function consists of three phases: The first step makes use of a majority vote on the bit level and transforms an input byte sequence into long runs of 0s and 1s as explained in Sec. III-A. For each byte of the input majority vote is performed on a fixed sized neighborhood: if the

Input:
11111000.10101010.11001100.01000110.11001100.01110101.00111000.10101010.11001100.00000110.11001111

Majority vote:
11111111.11111111.00000000.00000000.11111111.11111111.11111111.00000000.00000000.11111111.11111111

RLE:
0|2|2|3|2|2

Figure 1. A sample majority vote of N_5 (upper part) and subsequent RLE (lower part) with parameters $n = 2$, $t = 12$.

majority of bits in this neighborhood is equal to zero, then this neighborhood is represented by a 0 and by a 1 otherwise. Then in the second phase run length encoding (RLE) is used to represent each sequence of 0s and 1s by its length (in bytes). We explain this step in Sec. III-B. Finally, in Sec. III-C we explain how to encode the run length encoded sequences by a Bloom filter. As we use a majority vote approach to compress the input and Bloom filters to represent the digest we call our approach `mvHash-B` which is available to download[1].

The underlying idea of `mvHash-B` is that small local changes of the input byte sequence does not affect the majority vote as the majority of bits remains unchanged. Thus the hash value does not alter and similarity is preserved.

A similarity preserving hash function requires a comparison function to decide about similarity on base of two given digests. We describe the comparison function of `mvHash-B` in Sec. III-D. To conclude the description of `mvHash-B` we explain our choices of algorithmic parameters in Sec. III-E.

A. *Phase 1 - majority vote*

The core idea of the first phase is to transform the input into long runs of equal bits, which may easily be compressed during the subsequent phases. The transformation is based on a majority vote at each position of the input with output $0x00$ or $0xFF$.

Before explaining the majority vote step we have to introduce some notation. Let BS denote the input, which we consider to be a byte sequence of length L. The byte at position k of the input $(0 \leq k < L)$ is written as B_k. Furthermore N_k denotes the n-neighborhood of B_k, i.e. a byte sequence of length $(n + 1)$ (n has to be even):

$$N_k := B_{k-\frac{n}{2}}, B_{k-\frac{n}{2}+1}, \ldots, B_{k-1}, B_k, B_{k+1}, \ldots, B_{k+\frac{n}{2}-1}, B_{k+\frac{n}{2}} \ .$$

At the beginning or the end of the input the length of the n-neighborhood is smaller than $n + 1$, that is we do not perform any padding. For instance, we have

$$\begin{aligned} N_0 &= B_0, B_1, B_2, \ldots, B_{\frac{n}{2}} \text{ (of length } \frac{n}{2} + 1) \\ N_1 &= B_0, B_1, B_2, \ldots, B_{1+\frac{n}{2}} \text{ (of length } \frac{n}{2} + 2) \end{aligned}$$

The function $bitcount(N_k)$ counts the amount of bits set to 1 in N_k. If $bitcount(N_k)$ is higher than or equal to a certain threshold t (i.e. $bitcount(N_k) \geq t$), the majority vote of N_k is $0xFF$ and $0x00$ otherwise. The specification of t is given below.

An example of the majority vote step is given in Fig. 1, where the 2-neighborhood of the byte B_5 is considered. We have $N_5 = 11001100.01110101.00111000$, which results in $bitcount(N_5) = 12$. As we use the threshold $t = 12$, the majority vote yields $0xFF$.

[1]https://www.dasec.h-da.de/staff/breitinger-frank/#downloads

Finally, we have to explain how to set the threshold t to a reasonable value. By intuition one could assume that if at least half of the bits within N_k are 1 then the majority vote of N_k is $0xFF$. Thus $t = \frac{(n+1)\cdot 8}{2}$ seems to be a canonical threshold. As we discuss later there are byte sequences where the most significant bit (MSB) is most of the time 0 which would distort the results of the majority vote if the canonical threshold were used. Thus we revise the canonical choice of t and use

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

where ib is the average amount of influencing bits for a byte ($1 \leq ib \leq 8$). For instance, if MSB is always zero, then $ib = 7$, however, the default value is $ib = 8$.

Remark that we have to adjust the factor $n+1$ in Eq. 1 to the actual length of the n-neighborhood, if the neighborhood does not comprise $n+1$ bytes. As an example, we have $t = \frac{(\frac{n}{2}+1)\cdot ib}{2}$ in case of the n-neighborhood of B_0.

B. Phase 2 - encoding the majority vote bit sequence with RLE

In order to reduce the length of the majority vote bit sequence, we adjust the well-known run length encoding algorithm (RLE). RLE simply counts the amount of identical consecutive bytes and returns this number. Our implementation assumes that the majority vote bit sequence starts with a 0-run. Thus, if there is a 1-run in the beginning, the first RLE element is set to 0.

An example is shown in the lower part of Fig. 1. As the majority vote bit sequence starts with a 1-run, the first RLE output is 0.

C. Phase 3 - Fingerprint generation using Bloom filters

This section describes how `mvHash-B` processes the RLE sequence to build a fingerprint in form of a Bloom filter. We choose Bloom filters as they allow a fast comparison by performing an XOR and a bit count operation (Hamming distance). We only give a brief introduction into Bloom filters, more details are given in [16].

A Bloom filter is used to represent elements of a finite set S. It is an array of m bits initially all set to zero. In order to 'insert' an element $s \in S$ into the filter, k independent hash functions are used where each hash function outputs a value between 0 and $m-1$. The bits of the Bloom filter at the positions $h_0(s), h_1(s), \dots h_{k-1}(s)$ are set to one. Other similarity hash functions like `sdhash` make use of cryptographic hash functions (or divide the output of a cryptographic hash function in k parts). A typical value is $m = 2048$ as implemented by `sdhash` and `mrsh-v2`.

To answer the question if s' is in S, we compute $h_0(s'), h_1(s'), \dots h_{k-1}(s')$ and analyze if the bits at the corresponding positions in the Bloom filter are set to one. If this holds, s' is assumed to be in S, however, we may be wrong as the bits may be set to one by different elements from S. Hence Bloom filters suffer from a non-trivial false positive rate. Otherwise, if at least one bit is set to zero, we know that $s' \notin S$. It is obvious that the false negative rate is equal to zero.

`mvHash-B` uses Bloom filters in a slightly different way. It only uses a self-defined hash function as described below. This hash function is not cryptographically strong, as the security goals of this family of hash functions are not suitable for similarity preserving hashing. Furthermore our hash function is very efficient with respect to run time efficiency.

Recall, the input for this third phase is a RLE encoded sequence. We aim at the common Bloom filter size of $m = 2048$. Our hash function shall operate on fixed-sized parts of the RLE sequence and insert its output into a Bloom filter. We call our fixed-sized input parts of the RLE sequence *groups*. Each group consists of $\log_2(m)$ consecutive RLE elements, that is $\log_2(2048) = 11$ bits.

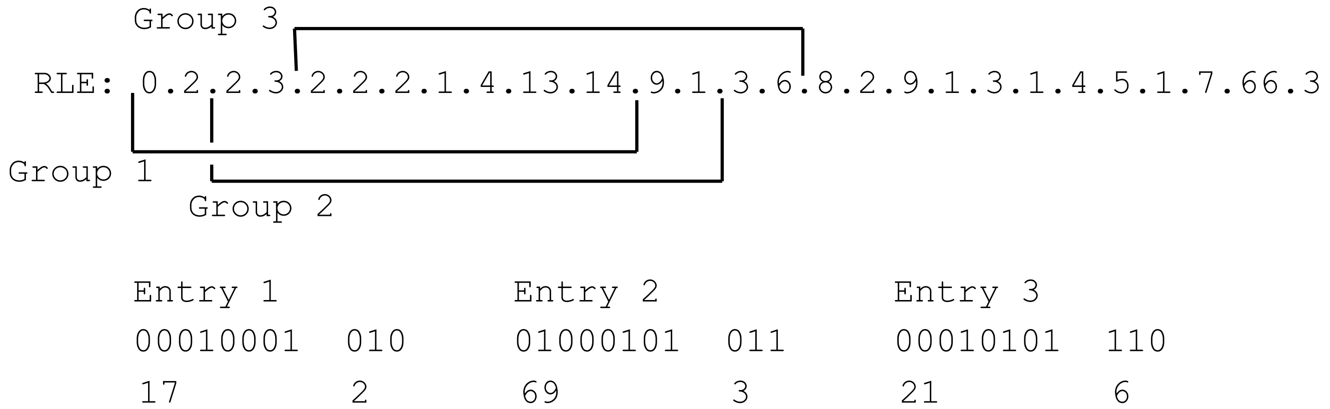

Figure 2. An illustration of how the RLE encoded string is processed by our hash function.

Input of our hash function is therefore a group of 11 subsequent RLE elements. In order to compute the output, we first reduce all RLE elements modulo 2. Due to the modulo 2 operation, the result is a bit sequence $b_{10}b_9b_8 \ldots b_0$ which is then divided into two parts:

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

To identify the bit which should be set, go to byte number v_1 simply by counting from left to right and set bit number v_2 within this byte, where $v_2 = 0$ means that the least significant bit is set.

In order to be alignment robust, the groups cannot be consecutive and need to have an overlap. The only two possibilities are an overlap by 9 oder 10 elements whereby 9 has obviously a better compression. Thus when sliding through the RLE sequence, two subsequent groups need to share 9 RLE elements, that is we shift the group by 2 elements when turning to the next one. An example is given below and exact details are provided in [19].

An overall example is illustrated in Fig. 2 where we marked the first three groups. Group 1 sets one bit within the Bloom filter, group 2 sets the next bit within the Bloom filter and so on. The position of the bit of group 1 is determined as follows:

1) Identify the first group of 11 RLE elements: 0.2.2.3.2.2.2.1.4.13.14
2) Use the modulo 2 operation for each element: 0.0.0.1.0.0.0.1.0.1.0
3) Divide the bit sequence in 2 parts: $v_1 = 01010001 = 0x51$ and $v_2 = 010 = 0x2$.
4) Go to byte $v_1 = 17$ of the Bloom filter and set bit $v_2 = 2$ to one.

Input and output of group 2 and group 3 is illustrated in Fig. 2, too.

One may argue that overlapping groups by 9 is redundant. In fact there are two reasons why it is this way. First, if only one RLE element changes, this affects up to 6 groups and thus there is a high certainty that Bits flip within the Bloom filter. Second is the more important fact, an overlapping by 9 makes our algorithm alignment robust which is best explained based on an example.

For instance, taking the marked three groups from Fig. 2 there are two options in case of a manipulation. First, it is possible that only the value of a RLE element itself changes, e.g., the second byte which is 2 turns to X. This would influence `Group 1` only. Second, it is possible that a manipulation causes a 'deletion' of the RLE element, e.g., there are lots of changes within the underlying byte sequence which changes the 'polarity' from a 1-run to a 0-run.

Assume the RLE sequence given in Fig. 2, a heavy *change* could cause a polarity flip of the second RLE element. Due to the fact that we only changed bytes, the RLE sequenced `0.2.2.` turned into a

single RLE element 4 $(0+2+2)$. The exact example is given in the following where we can see that G3 and G2 are equal.

Original sequence

```
Input 0.2.2.3.2.2.2.1.4.13.14.9.1.3.6.8.2
G1    0.2.2.3.2.2.2.1.4.13.14
G2        2.3.2.2.2.1.4.13.14.9.1
G3            2.2.2.1.4.13.14.9.1.3.6
```

manipulated sequence

```
Input 4.3.2.2.2.1.4.13.14.9.1.3.6.8.2
G1    4.3.2.2.2.1.4.13.14.9.1
G2        2.2.2.1.4.13.14.9.1.3.6
```

To conclude, a RLE polarity flip affects three RLE elements and result in a new one and thus we need a group shifting by two.

D. Bloom filter comparison

In this section we explain the computation of a similarity score of two given outputs of mvHash-B. The comparison algorithm was developed by us and is motivated in the description below. As our similarity hash output is composed of a sequence of Bloom filters, we first explain how to determine a distance score of two Bloom filters. In a second step, the computation of the final match score is presented.

The distance score di_{score} of two Bloom filters is based on the Hamming distance and hence the lower score, the more similar are the Bloom filters. The number of groups which are entered into the Bloom filters is fixed, except that the last Bloom filter can have fewer entries, the same applies for the one and only Bloom filter of a small file. Therefore, the Hamming distance is considered relatively to the number of entries in the Bloom filters.

Let bf_1, bf_2 be two Bloom filters and let $|bf|$ denote the number of 1-bits within a filter. Furthermore $hd(bf_1, bf_2)$ computes the Hamming distance between two filters. Then the distance score di_{score} of two filters is

$$di_{score} = \frac{hd(bf_1, bf_2)}{|bf_1| + |bf_2|} \cdot 100 \ . \tag{2}$$

Obviously we have $0 \leq di_{score} \leq 100$ and a distance score of 0 is a perfect match. A distance score close to 0 means a small distance of the two Bloom filters and thus a similarity of the underlying RLE sequences.

As stated in the previous section a hash value is a sequence of 1 or more Bloom filters. Let $BFS = \{bf_1, bf_2, \ldots bf_s\}$ and $BFS' = \{bf'_1, bf'_2, \ldots bf'_t\}$ be two Bloom filter sequences (hash values), which shall be compared. We assume $s \leq t$. Then the final similarity score SC is

$$SC(BFS, BFS') = \begin{cases} -1, & \text{if } t - s > 4 \\ 100 - \frac{1}{s}\sum_{i=1}^{s} \min\limits_{1 \leq j \leq t} di_{score}(bf_i, bf'_j), & \text{otherwise .} \end{cases} \tag{3}$$

In other words, to receive a final similarity score between two hashes we do an all-against-all comparison of Bloom filters and average the lowest distance scores. Obviously, we did not have to do an all-against-all comparison, we could have compared bf_1 against bf'_1, bf_2 against $bf'_2, etc.$ If some bytes are insert in the beginning of a file, the bytes which originally was represented in bf_1 would have been moved to bf_2, from bf_2 to bf_3, etc. To detect similar files in this case we decided to implement the all against-all-comparison. As a high similarity shall be represented by a large final match score, we subtract this value in a last step from 100 (recall that a small distance score means a high similarity). If the number of Bloom filters in BFS, BFS' is too different ($t - s > 4$) a comparison is not possible and the algorithm returns -1.

Table I
STATISTICS OF THE C-CORPUS.

	JPG	DOC	PDF	TXT
Number of files	2066	2000	1723	2000
Average file size (kB)	251.4	293.0	1022.1	53.2

E. Algorithmic parameters

The previous sections show that `mvHash-B` depends on multiple parameters and therefore this section motivates their default values. The main aim is to have a hash value length of approximately 0.5% of the input file length, which is adjustable by 3 settings. For tuning the configuration of `mvHash-B` we build a *c-corpus* (configuration corpus) by using various key words in a search engine. Table I[2] provides an overview of the file types and their average file size in our c-corpus.

`mvHash-B` has three parameters which are discussed in what follows:

ib is the amount of influencing bits per byte and 8 by default, $1 \leq ib \leq 8$. This parameter is used to adjust the threshold t of the majority vote as defined by Eq. 1. There are file types where the most significant bit (MSB) is mostly zero. For instance, if we analyze ASCII-encoded[3] text documents we realize that the MSB is hardly ever used, which influences the majority vote. Thus for text files the ib parameter should be adapted to $ib = 7$.

BF_a is the amount of groups in the RLE sequence, which are inserted into a Bloom filter. After BF_a groups are inserted, a new Bloom filter is created. Thus the final hash value of an input is a sequence of 1 or more Bloom filters each with a size of 256 bytes.

In order to find a suitable value BF_a, we have to consider that the more groups are inserted in a Bloom filter, the shorter the hash value is. However, if BF_a increases the possibility for collisions rises, too. Thus we need to find a good trade-off between compression and detection rate.

Table II summarizes the main results of our analysis. Row $p(bit = 1)$ is according to Eq. 4 the probability that a certain bit is 1 after inserting BF_a groups. The formula of Eq. 4 may easily be derived if a uniform probability distribution is assumed:

$$p(bit = 1) = 1 - \left(1 - \frac{1}{2048}\right)^{BF_a} \tag{4}$$

The relative hash value length is the ratio between input length (original file size) and the corresponding hash value. Row 4 *hash value sensitivity* denotes the expected amount of bits which will be influenced in the Bloom filter if one RLE element changes.

Recall, a group starts at every second RLE element and covers 11 RLE elements. Thus changing one RLE element effects 5 to 6 groups except in the beginning or at the end of the RLE sequence. Assuming $BF_a = 2048$, approximately 63.22% of the Bloom filter bits are set to one. As a consequence the manipulation of one RLE element changes approximately $(1 - 0.6322) \cdot 5 = 1.84$ bits within a Bloom filter.

`mvHash-B` uses $BF_a = 2048$ as 1.84 seems plausible for such a small change and 0.59% is a good compression.

The reason why the hash value length is not proportional is explained in Sec. IV-A.

[2]More details about the c-corpus are available on https://www.dasec.h-da.de/staff/breitinger-frank/#downloads

[3]http://www.asciitable.com/; last accessed on 2012-07-17

Table II
STATISTIC OF THE MVHASH-B APPROACH FOR JPG-FILES.

BF_a	**512**	**1024**	**2048**	**4096**
$p(bit = 1)$	22.12%	39.35%	63.22%	86.47%
Relative hash value length	1.26%	0.78%	0.59%	0.52%
Hash value sensitivity (in bits)	3.89	3.03	1.84	0.68

Table III
AVERAGE LENGTH OF RUNS (arl) FOR DIFFERENT FILE TYPES ($ib = 8$).

	n=20	n=50	n=100	n=200
JPG	13.75	24.97	41.91	73.40
PDF	14.31	29.49	47.15	81.89
DOC	26.86	55.88	96.69	172.39
TXT	37.41	275.39	915.62	1886.59

n is the size of the neighborhood to perform the majority vote as explained in Sec. III-A. Our tests show that a large neighborhood produces long runs, but loses accuracy and vice versa. Table III presents the average length of runs for different file types. We denote this quantity by arl. As expected the average length for compressed file types like JPG/PDF is shorter than for uncompressed file types like DOC/TXT. Thus besides the neighborhood size, the run lengths also depend on the file type, more specifically the entropy of the byte sequence.

Finally, we explain our approach to set n for different file types. We chose one compressed file type (JPG) and one uncompressed file type (DOC). We neglect TXT and PDF as they show similar behavior than DOC and JPG, respectively. If the RLE sequence of the second phase of `mvHash-B` is shorter than 11 elements, it is not possible to generate an entry for the Bloom filter. Using our c-corpus and the neighborhoods of Table III we search for the largest neighborhood which creates an RLE encoded string of at least 11 elements for all files in the c-corpus. The result is $n = 20$ for DOC-files and $n = 50$ for JPG-files.

IV. ASSESSMENT

To assess `mvHash-B` we use the t5-corpus [17, sec. 4.1] containing the files from Table IV and the c-corpus as described in Sec. III-E. Overall `mvHash-B` yield similar results for both corpuses. In the assessment we use the parameters which we found reasonable in Sec. III-E:

JPG-Files
- n: 50
- ib: 8 (default)
- BF_a: 2048

DOC-Files
- n: 20
- ib: 7
- BF_a: 2048

A. Hash value length

In Sec. III-E we agreed on inserting 2048 groups in each Bloom filter. As each group consists of 11 RLE elements but overlap by 9 RLE element, each group effectively represents 2 RLE elements.

Table IV
STATISTIC OF THE T5-CORPUS.

	JPG	DOC	PDF	TXT
Number of files	362	533	1073	711

Table V
PERFORMANCE COMPARISON OF SIMILARITY PRESERVING HASH FUNCTIONS AND SHA-1 FOR 100MIB.

	SHA-1	mvHash-B	sdhash 2.3	ssdeep 2.9
runtime	0.408s	1.14s	11.15s	1.82s
algorithm / SHA-1	1.000	1.48	14.48	2.36

Accordingly, each Bloom filter is able to compress approximately $m \cdot 2 \cdot arl$ bytes and thus we obtain a compression ratio of $\frac{m/8}{BF_a \cdot 2 \cdot arl}$.

For instance, let $n = 50$ and the average run length for JPG is $arl = 24.97$. This results in a compression ratio of $\frac{2048/8}{2048 \cdot 2 \cdot 24.97} = 0.0025 = 0.25\%$ for JPG files.

Practical tests on the c-corpus showed that the actual hash length differs. Having $n = 50$ for JPG files and $n = 20$ for DOC files, `mvHash-B`result in 0.59% and 0.47%, respectively. This is due to lots of small files which will at least have a signature length of 256 bytes (= one Bloom filter).

One may wonder why the hash value length is not proportional. This results from the file size: there are more small than large files. For instance, imagine all files are small and the hash value consists of only one Bloom filter. Thus the hash value length would not change at all for an increasing BF_a.

Compared to other similarity preserving hashing algorithms ours results in quite short hash value. `ssdeep` produces outputs having at most 100 Base64 characters. This rather good compression implies a security drawback as discussed in [1]. 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.

B. *Run time efficiency*

Hash functions are designed to be fast and thus we benchmark our approach against `sdhash` and SHA-1. `sdhash` is a very widespread algorithm and an ongoing project. SHA-1 is chosen to have a reference point. All three algorithms have to process a 100MiB file from `/dev/urandom`. The results are given in Table V where time represents the sum of the user-time and system-time from the Linux `time`-command. Moreover, practical tests showed that the results for `mvHash-B` is not significantly affected by adjusting its parameters. To conclude, `mvHash-B` is almost as fast as SHA-1 and therefore close to the maximum throughput that a SPH algorithm can reach. The run time efficiency of `bbHash` isn't acceptable and therefore we neglect this approach.

C. *Accuracy*

Accuracy is the ability to detect similar files with low costs in terms of false positive and false negative results. Therefore, first we need to identify a threshold which may distinguish between similar and non-similar files. The c-corpus is a collection of randomly collected files, therefore we assume that most files are non-similar. We performed an all-against-all comparison of all JPG-files in the c-corpus, the results showed that up to the score 70, there was several file pairs for each value, but none file pairs got a score of 70 or higher. Therefore, we assume that no non-similar files will get a score above 70 and defines 70 as the threshold. We want to measure the accuracy of `mvHash-B` for JPG-files. As none file pairs got a score above 70, we state that `mvHash-B` does not produce false positive results for JPG-files. To identify false negative results, we performed the all-against-all comparison by using `sdHash`. If `sdhash` detects similar files that `mvHash-B` did not, it is a false negative. Using `sdhash`,

Table VI
REAL-LIFE EXAMPLE OF mvHash-B .

File	Size	mvHash-B score	sdhash score
Barker.doc	115kB		
Barker.doc with table	125kB	99	65
Barker.doc with picture	149kB	95	65

no file pairs got a score above 21 which is sdhash' threshold for non-text files [17]. This means that for JPG, mvHash-B produces neither false positive or false negative results.

We performed the corresponding tests for DOC-files, it turned out that there were lots of file pairs for each value up to 70, but only some few file pairs above. Therefore, 70 seems as a suitable threshold also for DOC-files. 21 file pairs got a score above 70, it was 8 non-similar file pairs and 13 similar file pairs. It means that mvHash-B will produce some false positive results for DOC-files. We also performed the all-against-all comparison by using sdhash. For sdhash, 5 is the threshold for text-files [17]. There was in total 1,999,000 comparisons and some few thousands of them got a score above 5. All DOC-files share some common syntax due to the file format, but if the only similarity between them is the bytes which are specific for the file format, it makes no sense to interpret them as similar for mvHash-B as it only compares files of the same type. But, this is the other way round for sdHash, as it compares files of different file format. Particularly small DOC-files will be very similar due to the bytes that are specific for the file format. Therefore, the files with a score above 5 may be true positive in the terms of sdhash, but false positive in the terms of mvHash-B. As there are some thousands files, it is infeasible to manually determine whether or not they are similar. We do not know if false negative is an issue for mvHash-B .

At least we decided to include a real-life example. Our basis is a 20 pages DOC-file named Barker.doc[4] with text only, based on the original file we made two different versions. In the first version we included a table in the middle of the file, in the second we included a picture in the middle. A comparisons showed true positives for both algorithms, the detailed results are given in Table VI.

D. Robustness

We define robustness as the amount of changes that need to be done until a SPH algorithm could not identify any match. This section compares mvHash-B only to sdhash as bbHash is too slow for practical use and ssdeep could only handle 7 to 10 random changes [17].

To test the robustness of mvHash-B and sdhash we build a test called *edit operations ratio* (EOR). EOR measures the amount of random changes until the similarity score come below a certain threshold in relative to the original file length. The test randomly changes bytes all over an input whereby a change operation is either an insertion, deletion or substitution of a randomly chosen byte. For insertion and substitution the new byte is also randomly chosen. The procedure is quite simple: copy the original input, perform some changes, compare the modified copy with the original input by measuring the similarity and increase the edit counter EOR. If the similarity score is below a certain threshold the test stops and returns EOR. We used the same threshold as in Sec. IV-C.

The results of the edit operation ratio test are given in Table VII. One may say that sdhash outperforms mvHash-B as it has a higher EOR value. We argue that the ratio between compression (hash value length) and EOR is important and we therefore calculated $\frac{EOR}{compression} = IC$ (information content). The information content is motivated by the fact that it should be linear. The longer the hash

[4]It is included in the *mvhash (v2.0)* ZIP-file and available at https://www.dasec.h-da.de/staff/breitinger-frank/#downloads.

sequences, the more edit operations are needed to obtain a non-match - for one specific algorithm. Thus, if we allow longer signatures than 0.50% for `mvHash-B`, the EOR will also increase.

Table VII
EDIT OPERATION RATIO TEST FOR MVHASH-B AND SDHASH.

	JPG			DOC		
	compression	EOR	IC	compression	EOR	IC
`mvHash-B`	0.59	0.50%	0.85	0.47	0.51%	1.09
`sdhash`	2.60	0.92%	0.35	3.10	1.41%	0.45

To conclude we rate `mvHash-B` as a robust approach. Of course we need some more exhaustive test to verify our results.

V. CONCLUSION

This paper at hand presented a new algorithm for similarity preserving hashing named `mvHash-B`. Due to three trivial phases majority vote, run-length encoding and Bloom filters we have several advantages compared to existing algorithms.

A strong advantage of `mvHash-B` is the reduced generation time for a hash value. Our approach is almost as fast as the cryptographic hash function SHA-1 and thus approximately 20 times faster than `sdhash`. Another benefit is the compression ratio of 0.5% of our implementation which outperforms `sdhash` having 3.3%. As `mvHash-B` is using Bloom filters, the fingerprints are compared by using Hamming distance. Hamming distance requires only some few low-level operations and is therefore really fast. Overall this approach aims at a high throughput.

The current version has some configuration options and each file type requires its own configuration - no standard configuration works for all file types. In this paper `mvHash-B` is tested for JPG and DOC-files. For JPG-files `mvHash-B` seems to work very well, it is able to distinguish between similar and non-similar files. Considering DOC-files, `mvHash-B` works, but not as excellent as for JPG-files and some few false positive results may be expected. However, this is similar to other existing algorithms.

Our future work will focus on methods to automatically identify the file type and select the configuration settings accordingly. This might be possible using the entropy of the input sequences. For instance, JPG files are supposed to have a high entropy whereas DOC/TXT will have a lower entropy.

ACKNOWLEDGMENT

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

REFERENCES

[1] H. Baier and F. Breitinger, “Security aspects of piecewise hashing in computer forensics,” in *IT Security Incident Management and IT Forensics (IMF), 2011 Sixth International Conference on*, may 2011, pp. 21 –36.

[2] S. Chawathe, “Effective whitelisting for filesystem forensics,” in *Intelligence and Security Informatics, 2009. ISI ’09. IEEE International Conference on*, june 2009, pp. 131 –136.

[3] F. Breitinger and H. Baier, “Similarity Preserving Hashing: Eligible Properties and a new Algorithm MRSH-v2,” *4th ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*, October 2012.

[4] J. Kornblum, “Identifying almost identical files using context triggered piecewise hashing,” *Digital Investigation*, vol. 3, no. Supplement 1, pp. 91 – 97, 2006, the Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS ’06).

[5] V. Roussev, "Data fingerprinting with similarity digests," in *Advances in Digital Forensics VI*, ser. IFIP Advances in Information and Communication Technology, K.-P. Chow and S. Shenoi, Eds. Springer Boston, 2010, vol. 337, pp. 207–226.

[6] F. Breitinger and H. Baier, "A Fuzzy Hashing Approach based on Random Sequences and Hamming Distance," *ADFSL Conference on Digital Forensics, Security and Law*, pp. 89–101, May 2012.

[7] N. Harbour, "Dfcldd," http://dcfldd.sourceforge.net, last visited: August 2011.

[8] A. Divakaran, *Multimedia content analysis: theory and applications*. Springer, 2008.

[9] A. Tridgell, "Spamsum readme," http://www.samba.org/ftp/unpacked/junkcode/spamsum/README, last visited: 14.10.2012.

[10] L. Chen and G. Wang, "An efficient piecewise hashing method for computer forensics," in *Knowledge Discovery and Data Mining, 2008. WKDD 2008. First International Workshop on*, jan. 2008, pp. 635 –638.

[11] V. Roussev, G. G. R. III, and L. Marziale, "Multi-resolution similarity hashing," *Digital Investigation*, vol. 4, no. Supplement 1, pp. 105 – 113, 2007.

[12] K. Seo, K. Lim, J. Choi, K. Chang, and S. Lee, "Detecting similar files based on hash and statistical analysis for digital forensic investigation," in *Computer Science and its Applications, 2009. CSA '09. 2nd International Conference on*, dec. 2009, pp. 1 –6.

[13] F. Breitinger and H. Baier, "Performance Issues about Context-Triggered Piecewise Hashing," in *3rd ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*, vol. 3, October 2011.

[14] V. Roussev, "Hashing and data fingerprinting in digital forensics," *Computing in Science and Engineering*, vol. 7, pp. 49–55, March 2009.

[15] SHS, "Secure Hash Standard," 1995.

[16] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[17] V. Roussev, "An evaluation of forensic similarity hashes," *Digital Investigation*, vol. 8, no. Supplement 1, pp. 34 – 41, 2011, the Proceedings of the Eleventh Annual DFRWS Conference.

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

[19] K. P. Åstebøl, "mvHash - a new approach for fuzzy hashing," Master's thesis, Gjøvik University College, June 2012.

[20] V. Roussev, "Building a Better Similarity Trap with Statistically Improbable Features," *42nd Hawaii International Conference on System Sciences*, vol. 0, pp. 1–10, 2009.