Properties of a Similarity Preserving Hash Function and their Realization in sdhash

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

***Abstract*—Finding similarities between byte sequences is a complex task and necessary in many areas of computer science, e.g., to identify malicious files or spam. Instead of comparing files against each other, one may apply a similarity preserving compression function (hash function) first and do the comparison for the hashes. Although we have different approaches, there is no clear definition / specification or needed properties of such algorithms available.**

This paper presents four basic properties for similarity preserving hash functions that are partly related to the properties of cryptographic hash functions. *Compression* and *ease of computation* are borrowed from traditional hash functions and define the hash value length and the performance. As every byte is expected to influence the hash value, we introduce *coverage*. *Similarity score* describes the need for a comparison function for hash values.

We shortly discuss these properties with respect to three existing approaches and finally have a detailed view on the promising approach sdhash. However, we uncovered some bugs and other peculiarities of the implementation of sdhash.

Finally we conclude that sdhash has the potential to be a robust similarity preserving digest algorithm, but there are some points that need to be improved.

I. Introduction

The identification of similar byte sequences is relevant in many areas of computer science. Possible working fields are a forensic investigation, malware / spam detection or biometrics. In order to compare two different byte sequences a lot of different algorithms like the Hamming distance, Levenshtein distance and so on are available. Since both input byte sequences might be very large, we run into two problems. Let ω_1, ω_2 be two inputs:

1) If ω_1 and ω_2 are 'large', the calculation is very time consuming (run time problem).
2) If ω_1 is known and we want to compare it against ω_2, we need to have ω_1 available (disk space problem).

Thus one possible solution is to apply a compression function that preserves the similarity of the domain and to compare the compressed inputs instead of the original ones.

If we neglect similarity and focus on exact duplicates, this issue is solved by cryptographic hash functions as they produce a very short and fixed-length output. But cryptographic hash functions meet several security requirements and thus the hash value behaves pseudo-randomly. If the input changes slightly, approximately 50% of the output bits change. Comparing the similarity of files using cryptographic hash functions is therefore not possible.

To overcome this drawback there are a few approaches focusing on this issue called similarity preserving hash function (abbreviated SPHF) like ssdeep ([10]), sdhash ([14]), and bbhash ([7]). While thinking about new algorithms or evaluating existing ones, we run into the problem that there is no clear specification or definition for such functions (which is also in contrast to cryptographic hash functions).

The paper at hand therefore introduces four important properties for similarity preserving hash functions: Compression, ease of computation, coverage and similarity score. As we also use the term hash function, some of the properties are related to the ones from cryptographic hash functions. Thus this paper is a first step towards a definition for SPHF. In a second step we validate an appropriate tool for similarity hashing called sdhash against these properties. Due to a detailed analysis of sdhash we also uncovered some weaknesses and present improvements.

The rest of the paper is organized as follows: In the subsequent Sec. I-A we introduce notation and terms, which we use throughout this paper. Then, in Sec. II we sketch the state of the art and discuss relevant literature. Next, we show in Sec. III 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. IV and Sec. V, where we discuss the properties and make an assessment of sdhash based on these properties, respectively. Sec. VI shows some further peculiarities of the algorithm. The conclusion (Sec. VII) completes our paper.

A. Notation and Terms used in this Paper

In this paper, we make use of the the following notation and terms:

- h denotes a cryptographic hash function (e.g., SHA-1, MD5, RIPEMD-160).
- IN is a byte string of length L, $BS = B_0B_1B_2 \dots B_{L-1}$ 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 of sdhash 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.

II. Related Work

Hash functions are very widespread in computer science and mainly have two basic properties *compression* and *ease of computation* ([11]) where compression means that independent of the input length the output (hash value) is of a fixed size. Besides their use in cryptography and databases ([20, Sec. 9.6]), hash functions are also used within computer forensics to find *identical* files ([1, p.56++]). However, in order to uncover the similarity between two inputs, we need some kind of similarity preserving hash function[1].

Before introducing different similarity preserving hash functions, we shortly discuss the term similarity. In general we distinguish between a syntactic level (byte level) and a semantic level. For instance, in case of a picture the underlying byte sequence would be the syntactic level and the interpretation is the semantic level.

The first idea for similarity preserving hashing was called block based hashing and was presented in 2002 by Harbour. The proceeding is quite simple: divide a given input in blocks of fixed size, hash each block separately and concatenate all hash values. A sample implementation is given by `dcfldd`[2]. In order to overcome this approach, it is sufficient to add/remove one byte in the beginning. Thus the whole input shifts and all hash values will be different.

Therefore another idea was context triggered piecewise hashing (abbreviated CTPH) which divides an input based on its context. It was presented in 2006 by Kornblum ([10]) and based on a spam detection algorithm of Tridgell ([21]). 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 [6], [9], [17], [18] whereas a security analysis ([2], [5]) had shown that this approach cannot withstand an active adversary with respect to blacklisting and whitelisting.

A newer approach called `sdhash` was introduced by Roussev in 2010 ([14], [16]) based on some previous work in [13]. `sdhash` (similarity digest hashing) extracts "statistically-improbable features" using an entropy calculation for each 64 byte sequence, i.e., bytes 0 to 63, 1 to 64, 2 to 65, ... Once a 'characteristic feature' is identified it is hashed using the cryptographic hash function SHA-1 ([19]) and inserted into a Bloom filter ([3]). Hence, files are similar if they have common features. More details are given in Sec. III.

[15] 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".

The third algorithm for similarity preserving hashing is called `bbhash` and presented in [7]. The idea is to use 16 different building blocks (byte sequences with a length of 128 bytes) to rebuild a given input byte sequence as good as possible. Basically `bbhash` calculates the Hamming distance of all 16 building blocks at each position/offset of the input. If the Hamming distance is smaller than a given threshold, the index of this building block $0, 1, 2 \dots f$ is appended to the current state of the fingerprint. Thus, the final fingerprint is the sequence of the indices of the appended building blocks.

III. `sdhash`

Sec. III-A introduces the main concepts of the similarity digest algorithm (`sdhash 1.2`, [14]). The most recent version is 2.0 and from 20^{th} of April this year where he mainly changed the programming language from C to C++ and fixed some bugs ([16]).

Mostly an analysis is done due to the specification and not the implementation as both have to coincide. Therefore we did a detailed code review and identified some bugs which are shown in Sec. III-B.

A. Foundations

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 $B_0, B_1 \dots B_{L-1}$ of length L called input. Then a feature f_k is a sub byte sequence of length l in IN starting at B_k with $0 \leq k \leq L - l$:

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

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, according to [13], "we associate a *precedence rank* [(abbreviated R_{prec})] with each entropy measure

[1]This term might be a little bit confusing and *similarity digest* is more appropriate as hashing normally indicates a fixed size output. 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.

[2]http://dcfldd.sourceforge.net/; last accessed on 2012-06-18

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 $R_{prec}(0) = 882$, $R_{prec}(1) = 866$, ... 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 $F_0, F_1, \ldots F_p$ (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 $F_0, F_1, \ldots F_p$ 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 $128 \text{ features} \cdot 5 \frac{\text{bits}}{\text{feature}} = 640$ 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 featureswithin 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.

Let $SD_1 = \{bf_1, bf_2, \ldots bf_s\}$ and $SD_2 = \{bf'_1, bf'_2, \ldots bf'_r\}$ 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 $e = |bf \cap bf'|$.

`sdhash 2.0` is a parallelized version that divides an input in blocks of a fixed size, pick out 160 features with the lowest popularity score and insert them into a Bloom filter. Thus each block has its own Bloom filter.

B. Popularity Rank Computation Bug

[8] inspected the source code from `sdhash`, where the authors discovered two important bugs when computing the popularity rank R_{pop}. Although they discovered them in version 1.2 and communicated them to the author of `sdhash`, they are still present in the current version. As a detailed description is available in [8], we only summarize them here.

1) The *window size bug* is a typical off-by-one error concerning the window size used to compute R_{pop}. Thus the implementation does not correctly identify the minimal R_{prec} value in the current window and does not always select the 'statistically improbable feature' as defined by the specification.
2) The *leftmost bug* means that the implementation does not necessarily uses the leftmost feature as described in the specification. The impact of this second bug is low but we argue that the specification and the implementation should coincide.

These two bugs lead to false results of R_{pop} and thus to an unexpected behavior of the whole algorithm. Sec. V will show that after fixing these issues the coverage increase as there is less overlapping of features.

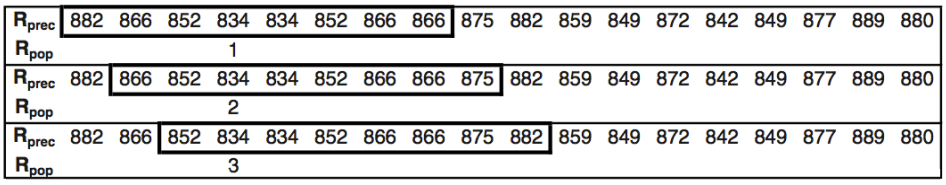

Fig. 1. Example for the R_{pop} calculation from [14].

IV. Properties of Similarity Preserving Hash Functions

Although `ssdeep` ([10]), `sdhash` ([14]), and `bbhash` ([7]) are candidates for similarity preserving hash functions, at the moment there is no clear specification or definition for similarity preserving compression functions / hash functions (abbreviated SPHF). Thus, in order to later evaluate the SPHF `sdhash`, in a first step we enumerate properties that we expect from an SPHF.

Since we use the term hash function, some of our properties are borrowed from the ones of cryptographic hash functions. Our proposed properties are as follows:

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

P2 **- Ease of computation.** Generating a hash value is 'fast' in practice for all kinds of inputs. This is similar to the comparable property of a classical hash function like SHA-1. It is obvious that ease of computation is a prerequisite for an SPHF to be usable in practice. As we assume SHA-1 to be fast, we take this algorithm as a benchmark.

P3 **- 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.

P4 **- Similarity score.** In order to compare two hash values we need a 'comparison function'[3]. Input of the comparison function are two similarity digests, 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 $[0, 100]$.

[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.

We point out that there are further (non-functional) requirements of an SPHF, especially security requirements like collision / second preimage resistance or resistance against anti-forensics attacks. However, in this paper our aim is to introduce the basic properties as described above[4].

To give a first impression of these properties, we discuss them in the context of the three currently available approaches `ssdeep`, `sdhash`, and `bbhash`. As `sdhash` seems to be the best trade-off candidate beween our properties and security requirements, `sdhash` is evaluated thoroughly in the subsequent sections.

`ssdeep` outputs a similarity digest of length about 100 Base64 characters, independent of the length of the input. This is due to the fact that the internal trigger function does not trigger any more, if the maximum fingerprint length is reached. Although this implies security drawbacks as discussed in [2], `ssdeep` achieves in general the best compression. In contrast, the digest of `bbhash` and `sdhash` is proportional to the input length, where the proportionality factor is 0.5% and 3.3%, respectively.

With respect to property P2, `ssdeep` is relatively fast compared to the two competitors. Its runtime is at $2.7 \cdot SHA$-1. This is due to the use of a fast trigger function and a non-cryptographic FNV hash function (Fowler, Noll, Vo, [12]).Our practical tests showed that `sdhash 2.0` is at $11.2 \cdot SHA$-1. [7] states that `bbhash` needs nearly 2 minutes for a 10MiB file and is neglected.

If we consider coverage, both `ssdeep` and `bbhash` fulfill property P3 and every byte of the input is expected to influence the digest. As we show in Section V-C 20% of the input bytes are expected to not influence the `sdhash` digest.

Finally, we turn to the similarity score function. While `ssdeep` makes use of the well-knwon weighted edit distance, `bbhash` may use this comparison function, too. A similarity score as percentage is available. However, the comparison function of `sdhash` has some peculiarities, which we address in Section V-D.

Although `ssdeep` seems to be a good approach concerning our properties of an SPHF, we have to keep in mind the security analysis ([2], [5]). An active adversary can easily overcome this approach. Regarding `bbhash` as stated in [7] the 'ease of computation' is rather poor and is approximately 150 times slower than `sdhash`. Thus we have a closer look at `sdhash` in what follows.

[4]This is similar to the general properties of hash functions and additional ones for cryptographic hash functions.

TABLE I
DIFFERENT STATISTICS ON SDHASH USING THE T5-CORPUS.

	average...	**improved**	**original**
1.	file size*	428,912	428,912
2.	gaps count	2888	2889
3.	min_gap*	1.090	1.076
4.	max_gap*	1834	1834
5.	avg_gap*	33.46	34.27
6.	ratio to file size	20.65 %	21.21 %
7.	overlap count	4387	4402
8.	min_lap*	1.110	1.108
9.	max_lap*	47.81	62.50
10.	avg_lap*	22.53	22.71
11.	ratio to file size	21.41 %	21.86 %
12.	inserted features	6923	6937
13.	ratio to file size	58.47 %	57.45 %
14.	all features	7276	7292
15.	required mod.	4248	4214
16.	ratio mods.	60.14 %	59.48 %
17.	SDsize % of file	3.321 %	3.344 %

* values are given in amount of bytes.

V. ASSESSMENT OF SDHASH BASED ON THE PROPERTIES

In the following we validate sdhash against the properties presented in Sec. IV. Therefore we divide this section into four parts, according to the properties and show how sdhash fulfills these properties.

To evaluate our results we use some real-world data the t5-corpus from [15, Sec. 4.1.] which is a collection of 4457 files (1.8 GB)[5] from 4 KB up to 16.4 MB.

A core result of our work is the statistics given in Table I, which shows the *average measurements* for all files within the t5-corpus (also the min/max values are no absolute values, but averaged). Besides the original version we also built the statistics for an *improved version* of sdhash, where we fixed the bugs from Sec. III-B.

Table I has two core statements:

1) The first two blocks (rows 2-11) show that a lot of features are overlapping, which results in wide gaps between two non-overlapping consecutive features.
2) The last block (rows 12-16) describes the impact of the overlappings. In the case that we want to manipulate each feature (e.g., to achieve a non-match) we do not need to change all *inserted features* (row 12) but only 60% of it due to the overlappings.

In the following we give some more details about Table I. The first block describes the statistics for gaps where row 2 is the amount of gaps followed by the minimum, maximum and average gap in bytes. Row 6 gives the ratio between the gap and the file size. The next block (rows 7-11) is constructed identically but with respect to overlaps. Details about the features are given in the last block (rows 12-16). Row 13 describes the ratio between features and file size *without* overlappings.

A. P1 - Compression

In Section IV we stated that the fingerprint size of sdhash is proportional to the input size and has a compression rate of 2.6% (see [14]). The basis for this calculation are six sample 100 MB document sets from the NPS Corpus containing doc, html, jpg, pdf, txt, xls and a 100MB file from /dev/urandom ([14, Sec. 5.1]). In order to validate this result we used the t5-corpus, hashed all files and compared the fingerprint length to the original file size. As shown in Table I line 17 we obtained a compression rate of approximately 3.3%.

For practical applications in computer forensics this is a rather bad compression rate. For instance, if the accumulated original file size is 1 terabyte, the corresponding sdhash database of fingerprints is about 33 gigabyte. For an 'ordinary' investigator it is difficult to handle this size, as his IT system will not supply a sufficient amount of RAM.

[5]http://roussev.net/t5/t5-corpus.zip; last accessed on 2012-06-10

```
uint32_t sum_hash(unsigned char c, uint32_t h) {
   h *= HASH_PRIME;
   h ^= c;
   return h;
}
```

Listing 1. FNV hash function from ssdeep 2.7.

B. P2 - Ease of Computation

sdhash uses SHA-1 for hashing the features. Although SHA-1 is optimized for performance, it is slower than non-cryptographic hash functions or cryptographic hash functions like MD5, which is one reason that sdhash is slower than ssdeep. Thus we identified two possibilities to increase the performance:

1) In the case that preimage resistance is important we would change the feature hash function to MD5 as it is faster ([4]). The security benefits of SHA-1 can be neglected as sdhash only relies on 55 bits out of the 160 bit SHA-1 fingerprint.
2) In the case that preimage resistance is not a prerequisite we would change the feature hash function to FNV as given in Listing 1 or any other non-cryptographic hash function. It is obvious that this simple hash function containing one multiplication and one XOR outperforms SHA-1 and MD5. Due to the large amount of features it is very unlikely that an active adversary brute forces each feature and result in a useful file.

In order that the security properties of SHA-1 are indispensable for sdhash we recommend to use all bits as given by our following idea: 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 address within the Bloom filter. Hence, sdhash only uses 55 bits of the SHA-1 hash value which allows a brute force attack as one attack vector.

To minimize this possibility we suggest to pad one bit at each sub hash, divide each sub hash into 3 blocks of 11 bits and XOR them, $\oplus$. A sample instruction sequence is given below, where sH_{new} is the new sub hash and sH_{int} the old

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 its security.

C. P3 - Coverage

In general hash functions are designed so that each bit of the input influences the hash value, otherwise it might be possible that a modification is not discovered. Therefore we present two major drawbacks of `sdhash` in Sec. V-C1 and Sec. V-C2 that allow to change up to 20% of an input with an unaltered fingerprint. Then Sec. V-C3 shows a minor issue that allows to change 30 bytes.

1) Unnoted Footer Changes: This section shows that it is possible to have two inputs that differ by 11% but still result in the highest similarity score of 100. As this issue is not addressed within the specification [14], it was found during the code review and is based on appending, deleting or modifying the end of an input.

Let r denote the amount of Bloom filters of an `sdhash` digest. Based on [14] 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.

Due to this behavior it is possible to append or delete data at the end of a file. For instance, if a file has exactly 128 features, we can append data containing up to 15 features or the other way round we can cut off features if the input has between 129 and 143 features. In both cases `sdhash` outputs the highest similarity score of 100.

The average byte length of an input containing exactly 15 features can be estimated using Table I where we averaged a large file corpus. In average a file has $428,912$ bytes and contains 7292 features. Thus, a byte sequence containing 15 features has a medial length of approximately $\frac{428,912 \cdot 15}{7292} = 882.29$ bytes.

To conclude, if we have an input containing exactly 128 features which results in a length of approximately $\frac{428,912 \cdot 128}{7292} = 7528.90$ bytes then we can append 882.29 bytes (which is over 11%) and `sdhash` outputs the highest similarity score. It is questionable if the highest match score is acceptable if a file changes by 11%.

2) Gaps and Overlaps: In the following we show that only approximately 80% of an input is considered within the fingerprint although it is possible to have full coverage. This raises the question if the parameters for the window size and the popularity rank threshold are chosen suitably.

Let L denote the length in bytes of an input and s the size of all Bloom filters as percentage in relation to L. Thus the size of all Bloom filters (the fingerprint) can be estimated by $BFs_{size} = \frac{L \cdot s}{100}$. As each Bloom filter consists of 256 bytes the number of Bloom filters is $BFs_{amount} = \left\lceil \frac{BFs_{size}}{256} \right\rceil$. Furthermore each Bloom filter except the last one contains 128 features and therefore can represent at most $128 \cdot 64$ bytes $= 8192$ bytes of the input. An upper bound of *coverage* (cv) (influencing bytes) can be estimated by

$$\begin{aligned} cv &= BFs_{amount} \cdot 8192 \\ &= \frac{L \cdot s}{100 \cdot 256} \cdot 8192 \\ &= 0.3200 \cdot s \cdot L. \end{aligned}$$

As we aim at a coverage of 100% we equate cv with L.

$$\begin{aligned} L &= 0.3200 \cdot s \cdot L \\ s &= 3.125 \ . \end{aligned}$$

Thus, in order to achieve a full coverage we need a hash value length of 3.125% which is fulfilled by `sdhash` as discussed in Sec. V-A. But although full coverage might be possible, this property is not fulfilled. In order to obtain the coverage of features rows 11 and 13 of Table I need to be added up. Thus the improved version results in 79.88% and the original one in 79.07 % which also coincides with the gaps from row 6. Due to the averaging the percentage values are not exactly 100%.

Although it is questionable if all bytes need to influence the final fingerprint, `sdhash` statistically allows to change approximately 20% of an input and the similarity score its still maximal. On the other hand, due to the overlapping of features one changed byte of the input changes multiple features.

3) Unnoted Byte Changes: Besides overlaps and gaps, [8] also discovered a minor issue concerning the first and last 15 bytes of a byte sequence. By design the first and last 15 bytes will never influence the fingerprint as there have to be 16 slides of the window to obtain a popularity score above 16. We rate this as a minor weakness due to the following two reasons:

- Besides text files, most file types do not allow to change the header or footer information.
- After changing 30 bytes both files are quite similar for files of practical interest. Nevertheless it is a drawback of `sdhash` that it outputs the highest match score even if the first and final 15 bytes are modified.

An easy way to resolve this weakness is to create a cryptographic hash value over the whole input, treat it as a 'feature' and insert it as first element into a Bloom filter.

D. P4 - Similarity Score

`sdhash` is promoted in two ways. First it can be used to identify similar files and second it can be used to find small pieces of an original file (fragments). Both issues are very important in the area of computer forensics but the way `sdhash` handles this point is questionable, because `sdhash` does not distinguish these two use cases.

Recall, we treat `sdhash` as a similarity preserving compression function. Therefore we like to have a high similarity score if two inputs are almost similar. The peculiarity is that comparing a fragment against an original file yields the highest similarity score. For instance, having a file containing 128

features and a fragment of this file with 20 features, `sdhash` outputs the maximum similarity score of 100 although these files are definitely very different.

This behavior is due to the computation of the similarity score $SF_{score}(bf, bf') = [100\frac{e-C}{E_{max}-C}]$ as given in Eq. (7) and the definition of $E_{max} = \min(|bf|, |bf'|)$ (see Eq. (4)). Recall that e denotes the amount of common bits in the Bloom filters bf and bf'.

A full match score occurs, if a full Bloom filter is compared to a non-full Bloom filter and if both Bloom filters are related to each other. More precisely let bf be a full Bloom filter and bf' be a Bloom filter where we dropped x features compared to bf. Roussev scales the comparison score to the number of bits set in the non-full Bloom filter bf', i.e., we have $E_{max} = \min(|bf|, |bf'|) = |bf'|$. However, as every feature of bf' is also represented by bf, we have $e = |bf'|$, too. Thus if $e > C$ the similarity score is 100, although bf and bf' are obviously different. Hence this comparison algorithm yields the highest match score by design.

In order to avoid this issue, we recommend to change the min-function in Eq. (7) into a max-function: $E_{max} = \max(|bf|, |bf'|)$. The cutoff point should still use the old variant of E_{max}. As an example we consider a non-full Bloom filter bf', where we ignore 64 features compared to bf. Then the number of expected bits set in bf' is $2048 \cdot \left(1 - \left[1 - \frac{1}{2048}\right]^{5\cdot 64}\right) = 296.3$. As we now have $E_{max} = 549.8$ instead of 296.3, our proposal yields an overall similarity score of $100 \cdot \frac{296.3-42.87}{549.8-42.87} = 50.00$, which is a reasonable result for this setting.

Overall we think that `sdhash` should have two modes, one for finding fragments and one for finding similar files. If a user aims at finding fragments the original setting is perfect. Otherwise our suggestions should be used.

VI. Further Analysis of `sdhash`

In the following we address a further aspect that is related to the robustness of `sdhash`. [8] showed that a full Bloom filter represents a byte sequence with 7545.46 bytes. In order to reduce the similarity score between two Bloom filters down to zero it is enough to flip 50 bits within this byte sequence. Thus there need to be a lot of changes all over the file which is only feasible for locally non-sensitive file types like `bmp` or `txt` but only hardly for locally sensitive file types like `jpg` or `pdf`.

Bloom filter shifts is another idea to reduce the similarity score of two files by inserting self-made features in the beginning of a file.

Let SD, SD' be two identical similarity digests. Below we describe an easy way how to reduce the similarity of both digests down to approximately 28.

An SD is comprised of $bf_0, bf_1, ...bf_s$ where

- bf_0 contains $F_0, F_1, \ldots F_{127}$,
- bf_1 contains $F_{128}, F_{129}, \ldots F_{255}$,
- and so on.

The similarity score $SD_{score}(SD, SD')$ is determined by Eq. (6) which is an all-against-all comparison of Bloom filters. The scores of all best matches are added up and divided by the amount of Bloom filters.

The idea to diminish the similarity score is to build own features that we insert at the beginning of an input. For instance, we build a feature F^*_{-1} and insert it. As a consequence

- bf_0 contains $F^*_{-1}, F_0, \ldots F_{126}$,
- bf_1 contains $F_{127}, F_{128}, \ldots F_{254}$
- and so on,

which will reduce the similarity score for all following bf. As all Bloom filters, except the last one, contain 128 features, we expect to achieve the lowest score by inserting 64 own features F^*.

[8] analyzed the impact of changing one feature and concluded that approximately 3.383 bits change within the Bloom filter for real-world data. Thus, if we insert 64 new features in the beginning, this will approximately change $3.383 \cdot 64 = 216.51$ bits. As described in Eq. (7) the similarity score of two Bloom filters is $SF_{score} = 100 \cdot \frac{e-C}{E_{max}-C}$. Due to the feature insertion, e should reduce from 550 to $550 - 216 = 334$ and

$$\begin{aligned} SF_{score} &= 100 \cdot \frac{e-C}{E_{max}-C} \\ &= 100 \cdot \frac{334-268}{550-268} \\ &= 23.40\ . \end{aligned}$$

In order to test our conclusion, we used our improved version of `sdhash` and did three tests:

1) We changed 64 features of 10,000 randomly generated files containing exactly 128 features and obtained an average similarity score of 24.61.
2) We inserted 64 features into cut files from the t5-corpus where 'cut' mean we reduced to original files down to 128 features. The result was 21.34.
3) We inserted 64 features into each file of the t5-corpus and resulted in an average similarity score of 27.27.

Inserting byte sequences is often possible for a lot of file types. [2] demonstrated that for the locally sensitive file type `jpg` and `pdf`. Focusing `txt` or `bitmap` it is even more trivial.

An important question in this context is: how much of an input do we have to change at least in order to reduce the similarity score to a minimum? Theoretically it is possible to create a shortest byte sequence with $16 \cdot 63 + 64 = 1072$ bytes as there is a minimum distance of 16 between two features. However, in our test case we used the shortest sequences we found within the t5-corpus which has a length of 2765 bytes. Once we identified such a byte sequence, the time complexity of manipulation is $O(1)$ because we can use this pre-computed feature sequence and insert it in any given file.

The current version 2.0 is a parallelized version and therefore splits an input in fixed blocks of l_b bytes and proceeds each block as described in Sec. III. Thus, instead of inserting a feature sequence in the beginning, we insert/delete a byte sequence of $\frac{l_b}{2}$ and therefore shift the offset of all blocks.

The problem due to these shifts is that we now do have two best matching Bloom filters. Thus, an idea to overcome this

issue would be to use to change the matching algorithm and consider the two consecutive Bloom filters (if the similarity score of the first one is lower than a certain threshold).

VII. Conclusion

Currently three different similarity preserving hash functions on the syntactical level are available. They all have different strengths and weaknesses. The developers of these algorithms pursue different strategies and thus it is very hard to compare these approaches. Therefore the paper at hand presents four basic properties and serves as a first step towards a definition for SPHF.

We evaluated the approach `sdhash` thoroughly, as there is yet no independent analysis available and `sdhash` is promoted strongly by its inventor. We did a detailed code review and identified two bugs that impair the original idea of the algorithm. The evaluation showed different weaknesses of `sdhash`, e.g., that there is no full coverage, but a large amount of overlapping features. Furthermore we showed that the chosen design of the fingerprint comparison function is made for fragment detection, but not for comparing two files.

Finally, we showed the robustness of `sdhash` against an active adversary and conclude that it is very hard to beat down the similarity score down to 0.

VIII. Acknowledgments

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

We thank Jesse Beckingham for supporting us with programming and collecting data.

References

[1] C. Altheide and H. Carvey, *Digital Forensics with Open Source Tools: Using Open Source Platform Tools for Performing Computer Forensics on Target Systems: Windows, Mac, Linux, Unix, etc*. Syngress Media, May 2011.

[2] H. Baier and F. Breitinger, "Security Aspects of Piecewise Hashing in Computer Forensics," *IT Security Incident Management & IT Forensics (IMF)*, pp. 21–36, May 2011.

[3] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, pp. 422–426, 1970.

[4] T. Boyles, *CCNA Security Study Guide: Exam 640-553*. John Wiley & Sons, March 2010, vol. 1.

[5] F. Breitinger, "Security Aspects of Fuzzy Hashing," Master's thesis, Hochschule Darmstadt, February 2011, last accessed on 2012-07-05. [Online]. Available: https://www.dasec.h-da.de/offerings/theses/

[6] 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.

[7] ——, "A Fuzzy Hashing Approach based on Random Sequences and Hamming Distance," *ADFSL Conference on Digital Forensics, Security and Law*, May 2012.

[8] F. Breitinger, H. Baier, and J. Beckingham, "Security and implementation analysis of the similarity digest sdhash," *First International Baltic Conference on Network Security & Forensics (NeSeFo)*, August 2012.

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

[10] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Forensic Research Workshop (DFRWS)*, vol. 3S, pp. 91–97, 2006.

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

[12] L. C. Noll. (2001) Fowler / Noll / Vo (FNV) Hash. Last accessed on 2012-07-05. [Online]. Available: http://www.isthe.com/chongo/tech/comp/fnv/index.html

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

[14] ——, "Data fingerprinting with similarity digests," *Internation Federation for Information Processing*, vol. 337/2010, pp. 207–226, 2010.

[15] ——, "An evaluation of forensic similarity hashes," *Digital Forensic Research Workshop*, vol. 8, pp. 34–41, 2011.

[16] ——, "Scalable data correlation," *International Conference on Digital Forensics (IFIP WG 11.9)*, January 2012.

[17] V. Roussev, G. G. Richard, and L. Marziale, "Multi-resolution similarity hashing," *Digital Forensic Research Workshop (DFRWS)*, pp. 105–113, 2007.

[18] K. Seo, K. Lim, J. Choi, K. Chang, and S. Lee, "Detecting Similar Files Based on Hash and Statistical Analysis for Digital Forensic Investigation," *Computer Science and its Applications (CSA '09)*, pp. 1–6, December 2009.

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

[20] S. Sumathi and S. Esakkirajan, *Fundamentals of Relational Database Management Systems*. Springer Berlin Heidelberg, February 2007, vol. 1.

[21] A. Tridgell, "Spamsum," Readme, 2002, last accessed on 2012-07-05. [Online]. Available: http://samba.org/ftp/unpacked/junkcode/spamsum/README