Chapter 10

SIMILARITY HASHING BASED ON LEVENSHTEIN DISTANCES

Frank Breitinger, Georg Ziroff, Steffen Lange, and Harald Baier

Abstract It is increasingly common in forensic investigations to use automated pre-processing techniques to reduce the massive volumes of data that are encountered. This is typically accomplished by comparing fingerprints (typically cryptographic hashes) of files against existing databases. In addition to finding exact matches of cryptographic hashes, it is necessary to find approximate matches corresponding to similar files, such as different versions of a given file.

This paper presents a new stand-alone similarity hashing approach called `saHash`, which has a modular design and operates in linear time. `saHash` is almost as fast as SHA-1 and more efficient than other approaches for approximate matching. The similarity hashing algorithm uses four sub-hash functions, each producing its own hash value. The four sub-hashes are concatenated to produce the final hash value. This modularity enables sub-hash functions to be added or removed, e.g., if an exploit for a sub-hash function is discovered. Given the hash values of two byte sequences, `saHash` returns a lower bound on the number of Levenshtein operations between the two byte sequences as their similarity score. The robustness of `saHash` is verified by comparing it with other approximate matching approaches such as `sdhash`.

Keywords: Fuzzy hashing, similarity digest, Levenshtein distance

1. Introduction

A crucial task during the forensic evidence acquisition process is to distinguish relevant information from non-relevant information – this is often equivalent to searching for a needle in a haystack. In order to identify known files, a forensic examiner typically hashes all the files using cryptographic hash functions such as MD5 [16] and SHA-1 [14] and compares the hash values (fingerprints) against hash databases such

G. Peterson and S. Shenoi (Eds.): Advances in Digital Forensics X, IFIP AICT 433, pp. 133–147, 2014.

as the National Software Reference Library (NSRL) [13]. This enables the examiner to automatically filter out irrelevant files (e.g., operating system files) and focus on files of interest (e.g., illegal images or company secrets).

While this procedure is well-established and straightforward, it has one main drawback. Regardless of the number of bits that are different between two files, the hash outputs behave in a pseudorandom manner. An active adversary can leverage this property of cryptographic hash functions to change one bit within each file to circumvent the automated filtering process. Consequently, it is necessary to perform approximate matching to identify files that are slightly manipulated or files that are similar (e.g., different versions of a given file).

In general, there are two levels of file similarity: (i) byte-level similarity that focuses on structure; and (ii) semantic-level similarity that focuses on meaning. Working at the byte-level is faster and is independent of the file type. However, it is easy to hide similarity, e.g., by changing the file type from JPG to PNG or the file encoding from ASCII to Base64.

Several approaches have been proposed to measure byte-level similarity: `ssdeep` [11], `sdhash` [18], `bbhash` [5], `mrsh-v2` [6] and `mvhash-B` [2]. These approaches typically pre-process an input file by dividing it into pieces, take unique features and then employ cryptographic hash functions to create the file fingerprint. However, it is possible to change just one bit within each piece or feature and corrupt the fingerprint.

This paper presents a stand-alone similarity hashing approach called `saHash` (statistical analysis hashing), which is not based on cryptographic hash functions. It employs four sub-hash functions, each of which creates its own sub-hash value. The four sub-hash values are concatenated to create the final hash value. `saHash` enables the detection of "small" changes between files – up to several hundred Levenshtein operations.

The `saHash` approach offers three advantages. First, it is almost as fast as SHA-1 and is faster than existing byte-level similarity approaches. Second, it creates nearly fixed-size hash values with a length of 769 bytes (1,216 bytes for 1 GiB files). Although using a fixed hash length has some disadvantages, the approach allows for faster comparisons and better ordering in a database. Third, similarity is defined in terms of the well-known Levenshtein distance. In contrast, all the approaches listed above yield similarity values between 0 and 100; these values express levels of confidence about similarity instead of true levels of similarity.

2. Background

This paper uses the term "approximate matching" as a synonym for similarity hashing and fuzzy hashing. Such an approach uses an "approximate matching function" to create hash values or fingerprints of files and a "comparison function" to output a similarity score for two file fingerprints.

In general, file similarity can be expressed in terms of byte-level similarity (structure) or semantic similarity (meaning). In what follows, we treat each input as a byte sequence and, therefore, only consider the byte-level similarity of files.

Several approaches are used to measure the similarity or dissimilarity of strings. We employ an approximate matching function based on the Levenshtein distance, one of the most popular string metrics. Given two byte sequences x and y, the Levenshtein distance is the smallest number of deletions, insertions and reversals (also called substitutions) that transform x into y [12].

The measurement of similarity has a long history. One of the early metrics is the Jaccard index [10], which was published in 1901. However, since the approach is very time consuming, Rabin [15] suggested the use of random polynomials to create flexible and robust fingerprints for binary data.

Over the years, several approaches have been proposed for approximate matching at the byte level. In 2002, Harbour [9] developed `dcfldd` that extended the well-known disk dump tool `dd` with the goal of ensuring integrity at the block level during forensic imaging. Instead of creating a single hash value over the entire file, the file is divided into fixed-size blocks and a hash value is generated for each block. Thus, this approach is also called "block-based hashing." Fixed-size blocks are well suited to flipping bits because only the hash value of the corresponding block changes. However, any insertion or deletion of a byte at the beginning causes all the block hashes to be different. Therefore, this approach has low alignment robustness [22], i.e., resistance to the addition or deletion of bytes.

Kornblum [11] introduced the notion of context-triggered piecewise hashing, which is based on a spam detection algorithm proposed by Tridgell [22]. The approach is similar to block-based hashing, except that instead of dividing an input into fixed-size blocks, a pseudorandom function based on a current context of seven bytes is used to split the input. Several enhancements have been proposed to Kornblum's approach, including modifications of the `ssdeep` implementation to enhance efficiency [8, 21], and security and efficiency [3]. Baier and Breitinger [4]

and Breitinger [1] have demonstrated attacks against context-triggered piecewise hashing with respect to blacklisting and whitelisting, along with some improvements to the pseudorandom function.

Roussev [17, 18] has proposed a novel `sdhash` method for approximate matching. Instead of dividing an input, the method selects multiple characteristic (invariant) features from the data object and compares them with features selected from other objects (the unique 64-byte features are selected based on their entropy). Two files are deemed to be similar if they have the same features (byte sequences). Experiments demonstrate that `sdhash` performs significantly better than `ssdeep` in terms of recall, precision, robustness and scalability [19]. Roussev [20] has also developed `sdhash 2.0`, a parallelized version of `sdhash`.

3. Approximate Matching Algorithm

This section presents the `saHash` algorithm for approximate matching. The algorithm estimates byte-level similarity based on the Levenshtein distance.

The `saHash` algorithm uses k independent sub-hash functions, also called "statistical approaches." Each sub-hash function produces its own hash value; the final fingerprint is created by concatenating the k sub-hash values. A fingerprint is compared by splitting it into its k component sub-hash values and k comparison functions are used to estimate the Levenshtein distance and produce the similarity score.

`saHash` is designed to detect near duplicates; the term "near" means that the two byte sequences being compared are essentially the same, but vary by a few hundred Levenshtein operations. In this case, `saHash` provides a lower bound for the Levenshtein distance between the two byte sequences as their similarity score.

The selection of the k sub-hash functions was a relatively simple procedure. We started by using trivial sub-hash functions such as the byte sequence length and byte frequency. We stopped adding new sub-hash functions when we could not find an attack that would defeat the combination of sub-hash functions. The term "attack" is expressed as follows. Let bs and bs' be two byte sequences. Furthermore, let $LD(bs, bs')$ be the Levenshtein distance of bs and bs' and $saH(bs, bs')$ be the Levenshtein assumption of `saHash`. An attack is valid if it is possible to find bs and bs' such that $LD(bs, bs') \leq saH(bs, bs')$.

We also checked if all the sub-hash functions were necessary because one sub-hash function could subsume another sub-hash function. Note that the modular design of `saHash` allows a new sub-hash function and comparison function to be incorporated if an attack is discovered on an

existing sub-hash function. Interested readers are referred to [23] for additional details.

`saHash` currently employs $k = 4$ sub-hash functions. The implementation (available at `www.dasec.h-da.de/staff/breitinger-frank`) was tested using Microsoft Visual Studio C++ and GCC on 32-bit systems. Thus, there might be problems running it in the 64-bit mode.

3.1 Sub-Hash Functions

This section describes the four sub-hash functions (i.e., statistical approaches) used by `saHash`.

Let bs denote a byte sequence (i.e., file) of length l comprising the bytes $b_0, b_1, \ldots b_{l-1}$. A modulus m defined by:

$$m = 2^{\max\left(8, \left\lceil \frac{\log_2 l}{2} \right\rceil\right)} \tag{1}$$

may be used to reduce sub-hash values. The value of m depends on the exponent $\max\left(8, \left\lceil \frac{\log_2 l}{2} \right\rceil\right)$, which results in a minimum modulus of $m = 256$. The modulus ensures uniformity in the use of the range of values. For instance, Breitinger and Baier [3] have observed that the average file size in a Windows XP installation is 250 KB $\approx 250 \cdot 10^3$ bytes. Assuming a uniform distribution of bytes, every byte will appear approximately 10^3 times. The modulo computation counters overflow and the probability distribution for all the modulo counters approach a uniform distribution. As a consequence, the lower bound ensures an adequate magnitude of m. Large files, which require more information to be stored due to more Levenshtein operations, have higher values of m.

The following sub-hash functions h_k $(0 \leq k \leq 3)$ are used:

- **Sub-Hash Function (h_0):** This function returns the byte sequence length equal to l as an unsigned integer. The bit size of the integer depends on the compilation mode and is either 32 bits or 64 bits. h_0 is an order criteria and is used to drop unnecessary comparisons: if the lengths of two byte sequences deviate widely (based on a threshold), the comparison is canceled.

  Trivial Attack: A byte sequence bs' with the same length l has the same hash value $h_0(bs')$.

- **Sub-Hash Function (h_1):** This function returns the byte frequency $freq$, a histogram showing the bytes `0x00` to `0xff` (i.e., `0` to `255`). For example, to obtain the byte frequency of `0xaa`, the

occurrences of `0xaa` in bs are counted. The result of $h_1(bs)$ is an array containing the frequencies of all 256 bytes ($freq_0$ to $freq_{255}$) modulo m. In order to reduce the fingerprint size, only the values from $freq_0$ to $freq_{254}$ are stored. This is because $freq_{255}$ can be predicted using the equation:

$$freq_{255}(bs) = (h_0(bs) - \sum_{i=0}^{254} freq_i(bs)) \mod m.$$

Note that the byte length $|h_1(bs)|$ is equal to $\frac{255 \cdot \log_2 m}{8}$.

Trivial Attack: A byte sequence bs' of length l containing the same bytes in any order defeats the combination of h_0 and h_1.

- **Sub-Hash Function (h_2):** This function returns the transition frequency $tfreq$. First, a left circular bit shift by four is performed; following this, the computations proceed as for h_1. In other words, the shift corresponds to one specific transition within bs where the transition is the four lowest bits of b_x and the four highest bits of b_{x+1}.

  As in the case of sub-hash function h_1, only $tfreq_0$ to $tfreq_{254}$ modulo m are stored and, therefore, $|h_1(bs)| = |h_2(bs)|$.

  Trivial Attack: A bs' of length l containing the same bytes in any order where blocks with the same transition are switched defeats the combination of h_0, h_1 and h_2.

- **Sub-Hash Function (h_3):** This function returns the unevenness array $uneva$ that captures relevant statistical features. In addition to the frequency of occurrence, the frequency of repeated occurrences is considered. Also, the unevenness of a byte b within bs is used as a measure of how evenly the occurrences of b in bs are spread.

  The result is an ordered array of 256 bytes starting with the less "uneven" bytes. Because the last uneven byte is predictable (the byte is not found in $uneva$), the byte length $|h_3(bs)| = 255$.

  Figure 1 shows the algorithm for computing the unevenness of all 256 bytes in a byte sequence bs. Three `for` loops are involved in the algorithm. The first loop calculates the average distance between the same b. The second loop computes the deviations for each b, which is equal to the square of deviations between each

```
01:array uneva[256] = means[256] =  {0, 0, ...}

02: for i = 0 to 255  {Create mean values}
03:   means[i] = [length(bs) + 1]/[freq(bs, i) + 1]
04: end for

05: array lastOcc[256]={-1,-1,...}
06: for i = 0 to length(bs) - 1   {Create deviations for each byte}
07:   bytes= bs[i]
08:   dev=means[byte] - (i-lastOcc[byte])
09:   uneva[byte] += dev*dev
10:   lastOcc[byte] = i
11: end for

12: for i = 0 to 255   {Deviation from last occurrence until end}
13:   dev = means[i] - (length(bs) - lastOcc[i])
14:   uneva[i] += dev*dev
15:   uneva[i] *= (freq(bs,i) + 1)
16: end for
```

Figure 1. Computation of the *uneva* array.

occurrence of a b. The third loop computes the deviation from the last occurrence until end of a b.

The final `saHash` value $H(bs)$ is the concatenation of the four sub-hashes:

$$H(bs) = h_1(bs) \parallel h_2(bs) \parallel h_3(bs) \parallel h_4(bs).$$

3.2 Distances between Sub-Hashes

Let bs_1 and bs_2 be two byte sequences and let d_k be a distance function that returns a measure of the distance between the sub-hash values $h_k(bs_1)$ and $h_k(bs_2)$. Note that $d_k\left(h_k(bs_1), h_k(bs_2)\right)$ measures the "inverse similarity" of two sub-hash values, i.e., the more distant the sub-hash values the less similar they are.

- **Sub-Hash Function (h_0):** An obvious distance function for d_0, which represents the byte sequence length is given by:

$$d_0 = |h_0(bs_1) - h_0(bs_2)|.$$

- **Sub-Hash Function (h_1):** In order to define the measure for h_1 (an array containing the frequencies of all the bytes), the auxiliary function $sub_i(h_1(bs_1), h_1(bs_2))$ is used to subtract the i^{th} element in $h_1(bs_1)$ from the i^{th} element in $h_1(bs_2)$. The distance function

d_1 is given by:

$$tmp = \sum_{i=0}^{255} |sub_i(h_1(bs_1), h_1(bs_2))|$$
$$d_1 = \left\lceil \frac{tmp - d_0}{2} \right\rceil + d_0.$$

First, the absolute values for all position-differences for all the frequencies are added. This yields the number of byte frequencies that are different. In general, there are two possibilities. If $|bs_1| = |bs_2|$, then $\left\lceil \frac{tmp-d_0}{2} \right\rceil$ substitutions are needed. If $|bs_1| \neq |bs_2|$, then d_0 must be added.

For example, `AAAAA` and `AABA` yield $d_0 = 1$ and $tmp = 2$. Thus, $d_1 = \lceil (2-1)/2 \rceil + 1 = 2$. The difference in length is considered by $d_0 = 1$ while all other differences can be corrected due to a substitution (`B` into `A`).

- **Sub-Hash Function (h_2):** Sub-hash function h_2 is similar to h_1, which is why the same auxiliary function sub is used. One difference is the division by four instead of two, which is due to the initial left circular bit shifting operation: one deletion or insertion can influence up to four positions in the $tfreq$ array. The distance function d_2 is given by:

$$tmp = \sum_{i=0}^{255} |sub_i(h_2(bs_1), h_2(bs_2))|$$
$$d_2 = \left\lceil \frac{tmp - d_0}{4} \right\rceil + d_0.$$

- **Sub-Hash Function (h_3):** To compute the similarity of two `uneva` arrays, an auxiliary function $pos_b(h_3(bs))$ is used to return the position of byte b in an array. The maximum distance is $256 \cdot 128$, which occurs if the array is shifted by 128. The distance function d_3 is given by:

$$tmp = \sum_{b=0}^{255} |pos_b(h_3(bs_1)) - pos_b(h_3(bs_2))|$$
$$d_3 = \left(1 - \frac{\text{tmp}}{256 \cdot 128}\right) \cdot 100$$

3.3 Final Similarity Decision

We decided to have a binary decision and, therefore, `saHash` outputs whether or not the two input two byte sequences are similar. To produce a result, `saHash` requires two thresholds, t_{LB} and t_{CS}. The first threshold is for the lower bound on the number of Levenshtein operations LB while the second is for a certainty score C that expresses the quality. The lower bound on the number of Levenshtein operations LB is assumed to be $\max(d_0, d_1, d_2)$ and CS is simply set to d_3. If $LB \leq t_{LB}$ and $CS \geq t_{CS}$, then the two byte sequences are considered to be similar.

Default threshold settings of $t_{LB} = 282$ and $t_{CS} = 97$ are recommended. These default values can be adjusted by users, but this may increase the probability of obtaining false positives.

A set of 12,935 files was used to obtain the default thresholds t_{LB} and t_{CS}. First, an all-versus-all-others comparison was performed and the corresponding LB values were entered into a matrix. Next, all file pairs were compared by hand starting at the lowest LB in the matrix and terminating at the first false positive, which had an LB value of 283. Thus, the threshold t_{LB} was set to 282. The threshold t_{CS} was set to 97 because the CS values for all the true positives exceeded 97.

The use of a large test corpus provides confidence that $t_{LB} = 282$ is a realistic threshold to avoid false positives. Note, however, that this represents a lower bound, so the actual number of Levenshtein operations would be higher; this is discussed later in the section on correctness.

3.4 Adjusting Sub-Hashes to the Modulus

This section discusses the special case when different moduli exist. Let bs_1 and bs_2 be two inputs that yield the moduli m and m' where $m < m'$ according to Equation (1). Because `saHash` is designed to identify small changes between two inputs, the moduli will at most differ by a factor of 2 and, therefore, $m' = 2m$.

In order to compute the distances d_1 and d_2, it is necessary to adjust $h_1(bs_2)$ and $h_2(bs_2)$. Because $(x \bmod 2m) \bmod m = x \bmod m$, the array is searched for $h_1(bs_2)$ and $h_2(bs_2)$, and m is used to recompute the frequency values; the sub-hash values can then be compared.

4. Evaluation

This section evaluates the correctness and performance of `saHash` with respect to efficiency (ease of computation and compression). Furthermore, `saHash` is compared with the `ssdeep` and `sdhash` approaches to assess its benefits and drawbacks.

The experimental environment comprised a Dell laptop running Windows 7 (64 bit) with 4 GB RAM and Intel Core 2 Duo 2×2 GHz. All the tests used a file corpus containing 12,935 files (2,631 DOC, 67 GIF, 362 JPG, 1,729 PDF and 8,146 TXT files) that were collected using a Python script via Google. The random byte sequences discussed in the evaluation were generated using the function `int rand()` with seed `srand(time(NULL))`.

4.1 Impact of a Changing Modulus

According to Equation (1), the modulus is fixed for all byte sequences with less than 2^{16} bytes. For larger byte sequences, the factor increases by two as the input size increases by a factor of four. We assume that a larger modulus is able to determine more modifications and, thus, has a better detection rate.

Because small files do not provide enough information and do not exhaust the counters, the test only used files from the corpus that were larger than 1 MB, a total of 749 files. To analyze the consequences of increasing the modulus, we manually changed the modulus to 7, 8, 9 and 10 bits. Based on the discussion above regarding the final similarity decision, we searched for the smallest LB that yielded a false positive and obtained the values $2,527$, $4,304$, $7,207$ and $13,503$, respectively. Hence, the higher the modulus, the more robust is `saHash`.

During the transition from 9 to 10 bits, it was possible to identify more true positives, but this was negatively impacted when a smaller modulus was used. Thus, it makes sense to use a larger modulus.

4.2 Correctness

This section discusses a randomly-driven test case where the evaluation mainly focused on false positives. The term "randomly" in this instance means that random changes were made all over the file.

First, we analyzed the impact of random changes to a byte sequence bs of length 250 KB, the average file size for the Windows XP operating system [3]. A random change involved a typical deletion, insertion or substitution operation, each with a probability of $\frac{1}{3}$ and all bytes in bs having the same probability to be changed.

We generated a test corpus T_{700} of 700 files using the `rand()` function; each file was 250 KB in size. For each n in the range $1 \leq n \leq 700$ and $bs \in T_{700}$, n random changes were applied to produce the file bs'. Next, all pairs (bs, bs') were used as inputs to `saHash` to obtain similarity decisions. As n increases, the dissimilarity of the files is expected to increase.

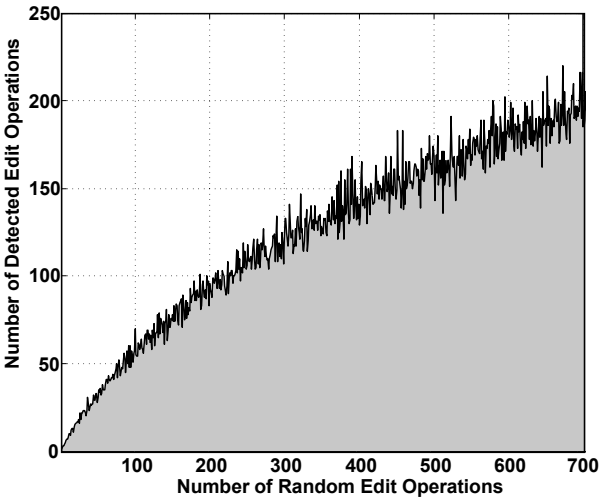

Figure 2. Edit operations vs. detected edit operations.

The `saHash` algorithm outputs two values, LB and CS. Figure 2 shows the results for up to $n = 700$ edit operations (x-axis) and the number of edit operations detected by `saHash` (y-axis). Even $n = 700$ yields a number of operations lower than 282 whereas only true positives were obtained previously. Thus, although the default value for LB is 282, `saHash` can detect many more modifications without any false positives. Note that the output is the lower bound on the number of Levenshtein operations.

As described above, the default threshold for CS was set to 97. Figure 3 shows the certainty score in relation to the number of random edit operations. Using the threshold of 97, `saHash` was able to reliably detect similarity for $n < 600$.

4.3 Cross Comparison

This section discusses the performance and correctness of `saHash`.

Compression. The overall hash value length $|H(bs)|$ for a byte sequence bs of length l is the sum of the lengths of the k sub-hashes. Thus, the length (in bits) is given by:

$$\left.\begin{array}{l} |h_0(bs)| = 4 \text{ bytes (32 bits)} \\ |h_1(bs)| = \frac{255 \cdot \max\left(8, \left\lceil \frac{\log_2 l}{2} \right\rceil\right)}{8} \text{ bytes} \\ |h_2(bs)| = \frac{255 \cdot \max\left(8, \left\lceil \frac{\log_2 l}{2} \right\rceil\right)}{8} \text{ bytes} \\ |h_3(bs)| = 255 \text{ bytes} \end{array}\right\} \left\lceil 4 + \frac{2 \cdot 255 \cdot \max\left(8, \left\lceil \frac{\log_2 l}{2} \right\rceil\right)}{8} + 255 \right\rceil$$

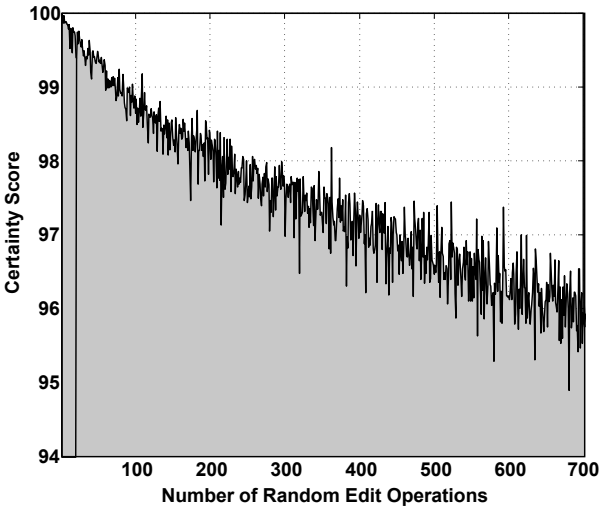

Figure 3. Ratio of the certainty score to the number of edit operations.

For instance, the fingerprint of a 1 MiB file ($=2^{20}$) is 897 bytes and, thus, has a compression ratio of 0.08554%.

The compression property is best satisfied by `ssdeep`, which generates hash values up to a maximum of 104 Base64 characters regardless of the input length.

Roussev [18] states that `sdhash` creates hash values whose length corresponds to about 2.6% of the input. Our experiments with several thousand files reveal that `sdhash 2.3` creates fingerprints of length approximately 2.5%. Therefore, `saHash` has better compression for all files larger than approximately $\frac{769 \cdot 100}{3.3} \approx 23,303$ bytes. Recall that the average file size of a Windows XP installation is 250 KB [3].

Ease of Computation. A highly desirable property of an approximate matching algorithm is ease of computation, which is why we gave a high priority to runtime efficiency during the design of `saHash`. In fact, the time complexity to compute $H(bs)$ for a byte sequence bs is $O(|bs|)$.

We generated a 100 MiB file from `/dev/urandom/` to compare the runtime performance of `saHash` with that of other algorithms. The C++ `clock()` function was used in the experiments; this function has the benefit that the times of parallel processes do not influence the timing. The results in Table 1 show that `saHash` is almost as fast as SHA-1 and faster than `ssdeep 2.9` and `sdhash 3.2`.

Table 1. Ease of computation of SHA-1 and approximate matching schemes.

	SHA-1	**saHash**	**ssdeep 2.9**	**sdhash 3.2**
Runtime	1.23 s	1.76 s	2.28 s	4.48 s

Correctness. It is well known that `ssdeep` is vulnerable to random changes made all over a file. For instance, if $n > 70$, the match score is always zero. In addition, Breitinger and Baier [4] have shown that it is possible to obtain an absolute non-match by editing ten bytes and to obtain a false positive with a similarity score of 100. Additionally, Roussev [19] notes that `ssdeep` yields no false positives for a similarity score greater than 90%. However, our random test demonstrates that, for $n = 5$, the similarity score could be reduced to less than 90%.

Two groups of researchers [7, 19] have shown that `sdhash` is robust. In fact, `sdhash` allows up to 1.1% random changes while still detecting similarity. However, because the score is not a percentage value, it is difficult to assess the actual similarity between two inputs.

In summary, `saHash` is very robust when small changes have been made all over a file. However, it has one main drawback – it only can compute the similarity of inputs that have similar sizes. Therefore, we recommend that `sdhash` be used to detect embedded objects and fragments.

5. Conclusions

The `saHash` algorithm is a new stand-alone approach for approximate matching that is based on the Levenshtein metric. The principal advantages of `saHash` are its modular design that enables it to be enhanced to cope with new attacks and its robustness to changes made all over a file. The final hash value of `saHash` does not exceed 1,216 bytes for files smaller than 1 GiB and, therefore, has a nearly fixed size. Also, `saHash` has good runtime efficiency and is nearly as fast as SHA-1 and is significantly faster than other approximate matching approaches.

Topics for future research include designing an approach that outputs the upper limit of Levenshtein operations and conducting a detailed security analysis of `saHash`. Additionally, because `saHash` can improve existing algorithms by replacing their underlying cryptographic hash functions, it would be interesting to attempt to combine `saHash` and `sdhash`, and to analyze the performance of the new version of `sdhash`.

Acknowledgement

This research was partially supported by the European Union under the Integrated Project FIDELITY (Grant No. 284862) and by the Center for Advanced Security Research Darmstadt (CASED).

References

[1] F. Breitinger, Security Aspects of Fuzzy Hashing, M.Sc. Thesis, Department of Computer Science, Darmstadt University of Applied Sciences, Darmstadt, Germany, 2011.

[2] F. Breitinger, K. Astebol, H. Baier and C. Busch, `mvhash-b` – A new approach for similarity preserving hashing, *Proceedings of the Seventh International Conference on IT Security Incident Management and IT Forensics*, pp. 33–44, 2013.

[3] F. Breitinger and H. Baier, Performance issues about context-triggered piecewise hashing, *Proceedings of the Third International ICST Conference on Digital Forensics and Cyber Crime*, pp. 141–155, 2011.

[4] F. Breitinger and H. Baier, Security aspects of piecewise hashing in computer forensics, *Proceedings of the Sixth International Conference on IT Security Incident Management and IT Forensics*, pp. 21–36, 2011.

[5] F. Breitinger and H. Baier, A fuzzy hashing approach based on random sequences and Hamming distance, *Proceedings of the Conference on Digital Forensics, Security and Law*, 2012.

[6] F. Breitinger and H. Baier, Similarity preserving hashing: Eligible properties and a new algorithm `mrsh-v2`, *Proceedings of the Fourth International ICST Conference on Digital Forensics and Cyber Crime*, 2012.

[7] F. Breitinger, G. Stivaktakis and H. Baier, FRASH: A framework to test algorithms for similarity hashing, *Digital Investigation*, vol. 10(S), pp. S50–S58, 2013.

[8] L. Chen and G. Wang, An efficient piecewise hashing method for computer forensics, *Proceedings of the First International Workshop on Knowledge Discovery and Data Mining*, pp. 635–638, 2008.

[9] N. Harbour, `dcfldd` (`dcfldd.sourceforge.net`), 2006.

[10] P. Jaccard, Distribution de la flore alpine dans le bassin des drouces et dans quelques regions voisines, *Bulletin de la Société Vaudoise des Sciences Naturelles*, vol. 37(140), pp. 241–272, 1901.

[11] J. Kornblum, Identifying almost identical files using context triggered piecewise hashing, *Digital Investigation*, vol. 3(S), pp. S91–S97, 2006.

[12] V. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, *Soviet Physics Doklady*, vol. 10(8), pp. 707–710, 1966.

[13] National Institute of Standards and Technology, National Software Reference Library, Gaithersburg, Maryland (`www.nsrl.nist.gov`).

[14] National Institute of Standards and Technology, Secure Hash Standard (SHS), FIPS Publication 180-4, Gaithersburg, Maryland, 2012.

[15] M. Rabin, Fingerprinting by Random Polynomials, Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, 1981.

[16] R. Rivest, The MD5 Message-Digest Algorithm, RFC 1321, 1992.

[17] V. Roussev, Building a better similarity trap with statistically improbable features, *Proceedings of the Forty-Second Hawaii International Conference on System Sciences*, 2009.

[18] V. Roussev, Data fingerprinting with similarity digests, in *Advances in Digital Forensics VI*, K. Chow and S. Shenoi (Eds.), Springer, Heidelberg, Germany, pp. 207–226, 2010.

[19] V. Roussev, An evaluation of forensic similarity hashes, *Digital Investigation*, vol. 8(S), pp. S34–S41, 2011.

[20] V. Roussev, Managing terabyte-scale investigations with similarity digests, in *Advances in Digital Forensics VIII*, G. Peterson and S. Shenoi (Eds.), Springer, Heidelberg, Germany, pp. 19–34, 2012.

[21] K. Seo, K. Lim, J. Choi, K. Chang and S. Lee, Detecting similar files based on hash and statistical analysis for digital forensic investigations, *Proceedings of the Second International Conference on Computer Science and its Applications*, 2009.

[22] A. Tridgell, `spamsum` (`mirror.linux.org.au/linux.conf.au/2004/papers/junkcode/spamsum/README`), 2002.

[23] G. Ziroff, Approaches to Similarity-Preserving Hashing, B.Sc. Thesis, Department of Computer Science, Darmstadt University of Applied Sciences, Darmstadt, Germany, 2012.