ssdeeper: Evaluating and Improving ssdeep

By:
Carlo Jakobs (Fraunhofer FKIE), Martin Lambertz (Fraunhofer FKIE), and Jan-Niclas Hilgert (Fraunhofer FKIE)

From the proceedings of
The Digital Forensic Research Conference
DFRWS USA 2022
July 11-14, 2022

Forensic Science International: Digital Investigation 42 (2022) 301402

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2022 USA - Proceedings of the Twenty-Second Annual DFRWS USA

ssdeeper: Evaluating and improving ssdeep

Carlo Jakobs*, Martin Lambertz, Jan-Niclas Hilgert

Fraunhofer FKIE, Bonn, Germany

ARTICLE INFO

Article history:

Keywords:
ssdeep
Similarity hashing
Fuzzy hashing
Context-triggered piecewise hashing

ABSTRACT

This paper focuses on ssdeep, a widely used context-triggered piecewise hashing algorithm. We present an extensive discussion and evaluation of the ssdeep algorithm and its implementation and reveal several inconsistencies. We also propose various improvements, which we implemented and evaluated concerning the desired characteristics of a similarity hashing algorithm. Both, the implementations and our dataset are publicly available.

While our improvements lead to more stable hash values, better runtime performance and better detection rates, we argue that there is not much room for improvement without modifying the ssdeep core algorithm.

1. Introduction

With the growing and overwhelming amount of digital data, automated systems to filter the files on a storage medium are a necessity. The filtering process is commonly used to identify suspicious files or to filter uninteresting files, such as benign operating system files. For this, an investigator typically uses precomputed databases of cryptographic hash values and compares the hashes of the files at hand with the ones in the database. The identification and filtering of exact duplicates reduce the amount of data that has to be examined manually. Minimizing extraneous data or highlighting suspicious and relevant data is a prerequisite for an effective and efficient investigation.

However, cryptographic hash functions are limited to the identification of identical copies; edited versions of documents or updated software cannot be identified. Different code versions of a software, changed documents or embedded objects will most likely not match any of the previously computed hash values. Similarly, malware, which employs self-modification to alter its code on each infection, remains undetected. It also allows malicious users to counteract a filtering process by making even only a one-bit change to known files. To sum up, cryptographic hashing is not suited to identify similar files.

Approximate matching tries to solve this kind of file matching problem by gauging the similarity between two inputs. In general, there are different levels of abstractions on which this can be accomplished (Breitinger et al., 2014). The lowest level solely attempts to approximate the similarity between byte sequences without taking any structures or meaning of the inputs into account. At the highest level, we try to interpret the actual content represented by a byte sequence.

Algorithms working on the highest level represent the perceptual similarity but they are very file type specific as the file format has to be parsed and interpreted. On the other hand, algorithms working on the lowest level, do not require such parsing and can be used for any file format. The latter is commonly called similarity hashing and seems favorable in cases where we quickly want to find files that are similar to known sequences of bytes as it is file type agnostic and typically faster than algorithms working at higher levels.

There are various similarity hashing algorithms (e.g. Roussev (2010); Breitinger and Baier (2013)) and although prior work has shown that ssdeep (Kornblum, 2006) is outperformed by some of them in terms of precision and recall (Roussev, 2011), it is still widely incorporated into real-world applications such as the National Software Reference Library (NSRL) (National Institute of Standards and Technology, 2019), VirusTotal (VirusTotal), ThreatExchange (Meta, 2022) and the current STIX specification (OASIS Open, 2021).

Over the years, a variety of improvements have been proposed in the literature (e.g. Breitinger and Baier (2012); Chen and Wang (2008)) and new features have been added to the code. The paper at hand assesses the state and the correctness of the current ssdeep implementation, discusses feasible manipulations and proposes

* Corresponding author.
E-mail addresses: carlo.jakobs@fkie.fraunhofer.de (C. Jakobs), martin.lambertz@fkie.fraunhofer.de (M. Lambertz), jan-niclas.hilgert@fkie.fraunhofer.de (J.-N. Hilgert).

https://doi.org/10.1016/j.fsidi.2022.301402
2666-2817/

improvements for a more efficient hash value generation. In addition, our evaluation compares a variety of ssdeep implementations covering various combinations of improvements and features, to enable an investigator to determine a more efficient configuration of the algorithm dependent on the application at hand. While our improvements can enhance certain aspects of ssdeep, our evaluation suggests that without fundamental changes to the core algorithm, major enhancements are most likely not possible.

2. Similarity hashing and ssdeep

A similarity hashing algorithm consists of a similarity preserving hash function (SPHF) to generate hash values and of a comparison function (CF) to score their similarity. The desired characteristics of similarity hashing algorithms can be described by three general and two security properties (Breitinger and Baier, 2013; Breitinger et al., 2014), which will be discussed in the following:

1. **Compression:** In favor of storage capacity and ease of computation of the comparison function, the output of the similarity preserving hash function should be much smaller than the input. Note that, in contrast to traditional hashing algorithms, the output does not have to be of a fixed length.
2. **Ease of computation:** To be useable in practice, it is crucial that generating hash values as well as comparing them is sufficiently fast.
3. **Similarity score:** The output is a similarity score that approximates the similarity of the corresponding inputs. Its value is preferably between 0 and 100, representing the percentage of the similarity.
4. **Coverage:** To detect small changes when comparing hash values, it is expected that every byte of the input file has an equally high statistical probability to influence the output in some way.
5. **Obfuscation resistance:** It should be difficult to manipulate the input file, such that comparing the hash values of the original file and the manipulated version yields a false negative.

Similarity preserving hash functions often generate hash values for segments within the input file similar to block-based hashing. Hence, by measuring the number of similar segments, the similarity of two files can be computed.

To counteract manipulations, such as a character deletion or insertion at the beginning of the file, in contrast to simple block-based hashing, the segments are often chosen based on a current context, dividing the file into different sized segments.

Kornblum called this type of similarity hashing context-triggered piecewise hashing (Kornblum, 2006). In his paper from 2006, he introduced a method based on the spam detection algorithm spamsum by Tridgell (2002) and provided an implementation of the algorithm in form of the open-source tool ssdeep (Kornblum et al., 2017b).

The basic idea is to divide the input up into segments by identifying trigger points using a rolling hash function on a context of 7 bytes. For each segment, a Fowler-Noll-Vo (FNV) hash is computed and the resulting similarity digest is a concatenation of the individual FNV hashes.

A more detailed description of the algorithm is given by the following step-by-step description:

1. At first, a window with a fixed size of 7 bytes slides through the input file to determine the trigger points.
2. For each window a hash value h is generated using a rolling hash function inspired by the Adler32 checksum.
3. If $h \equiv -1 \bmod b$, where b denotes the expected block size, the corresponding window is considered to be a trigger point, yielding a new segment. This step is also performed for double the block size $2b$, such that the algorithm divides the input up into two different segmentations.
4. The trigger points determine the segments, which are used as an input for the 32 bit FNV hash function.
5. To achieve a compressed representation of the input and consequently a fast hash value comparison, for each FNV hash value only the 6 least significant bits are stored as a base64 character.
6. The final similarity hash value consists of the block size and two sets of base64 character strings, called signatures.

Since Kornblum aims for a maximum of $S = 64$ segments, an initial expected block size is defined by the following saltus function:

$$b_{init} = b_{min} \cdot 2^{\left\lfloor log_2\left(\frac{n}{S \cdot b_{min}}\right)\right\rfloor},$$

with a minimum expected block size of $b_{\min} = 3$ and input length n. If the digest with a current block size of b does not exceed the minimum length of $\frac{S}{2} = 32$ base64 characters, the input is processed again using $b = \frac{b}{2}$.

When comparing two signatures their respective block sizes have to be identical, because the block size influences the determination of segments. Therefore, only ssdeep hash values of files of similar length can be compared. To be more flexible in the file comparison, the output is a concatenation of two signatures using the block sizes b and $2b$. Thus, for a comparison, the other hash value must have block sizes from the ranges $(\frac{b}{2}, b)$, $(b, 2b)$, or $(2b, 4b)$. Files that differ in length exceeding a factor of 4 produce similarity hash values, which are most likely not comparable. If two comparable hash values are present, a distance between the base64 character strings with matching block sizes is calculated using a weighted Levenshtein distance as implemented in the spamsum algorithm. The Levenshtein distance between two strings is defined as the minimum number of operations required to change one string into the other. Here, insertions and deletions are weighted with a distance of 1, changes add a distance of 3 and each swap is weighted with a distance of 5. Finally, the resulting distance is inverted and rescaled to a value between 0 and 100 to represent the percentage of the similarity.

3. Analysis and improvement of ssdeep

In this section, we present our detailed analysis of ssdeep. We differentiate our results into three categories: inconsistencies, errors and optimizations. Inconsistencies denote differences between Kornblum's paper and the implementation, errors denote implementation errors and optimizations denote aspects, which we consider as points with potential for improving ssdeep. During the following subsections, the individual inconsistencies, errors and optimization starting points are represented in italics. Our proposed modifications to address the respective issues are represented in bold. In our evaluation, we use these modifications to show which combination of them leads to an improved ssdeep performance.

Note that our modifications do not change the core algorithm or the representation of hash values. Instead, we only exchange building blocks or alter parameters. Still, most of our modifications will lead to different hash values which means that in most cases

the new hash values do not interoperate with the ssdeep hash values in existing databases for the purpose of similarity determination.

3.1. Inconsistencies

Here, we document inconsistencies between the original algorithm presented in Kornblum's paper (Kornblum, 2006) and the current version 2.14.1 of the ssdeep implementation. Inconsistencies include prior optimizations, but also tacit deviations from the original publication. Some inconsistencies can be considered to be a reasonable design choice; these are listed here for the sake of completeness only. Other inconsistencies are further analyzed in our evaluation.

Different initialization parameter: The prime value of the FNV-1 algorithm is set corresponding to the documentation (Fowler et al.), but the FNV-1 initialization parameter in ssdeep does not match the least significant bits of the initialization parameter listed there. We are not aware of any reason for the initialization parameter currently used in ssdeep. This is, however, not considered an error, because almost any initialization parameter serves as long as it is non-zero. Thus, it does not require any modification and is not considered any further.

Restructuring of the SPHF: With the release of ssdeep version 2.10 (Kornblum and Grohne, 2013), the hashing algorithm has been rewritten. This significantly improved the algorithm in terms of runtime by preventing a recalculation of the hash, if the minimum number of 32 base64 characters is not reached. Now, for every block size, the base64 character signature is calculated, such that after processing the input, the signature with the largest block size, which exceeds the minimum number of 32 characters, is selected to represent the input as the similarity hash.

To achieve a faster calculation of all the signatures, a block size b and the corresponding signature are discarded as soon as 64 characters are reached and double the block size $2b$ reaches the minimum requirement of 32 characters. In addition, the saltus function—previously used to initialize the block size—is used to limit the number of signatures being calculated.

For the sake of completeness, we also point out that the function to compute the initial block size b_{init} differs between what is presented in the paper and what is actually implemented. While the former uses a floor operation in the exponent, the latter uses a ceiling operation. This has already been mentioned by Breitinger and Baier (2012) and is not discussed any further in this work.

To stay compatible with the ssdeep output, the size of the second signature is limited to 32 characters. This character limitation, however, requires an additional FNV hash value calculation for every initialized block size at each step of the iteration through the input and, in addition, truncates valuable information about the input file.

Modification `-no32lim`: The removal of the character limitation may achieve a significant performance improvement in the calculation and might lead to more accurate results for comparisons with a second signature. The higher precision results from a larger hash length that is expected to represent the input more accurately, but at the same time remains limited to at most 64 characters.

Inconsistent description of the comparison function: There are some inconsistencies and changes concerning the comparison function. First, for two similarity hash values with a matching block size, every sequence of repeating characters longer than 3 is eliminated. This shortens the strings and speeds up the calculation of the Levenshtein distance. While this technique is mentioned in the original publication, the exact threshold of 3 has not been documented in the paper.

Second, the distance scores of the Levenshtein distance are weighted differently. The implementation scores an insertion or deletion of a character as a distance of 1 and a substitution as a distance of 2. The swap of characters is not considered explicitly but corresponds to a distance that equals two character substitutions, i.e. $2 + 2 = 4$.

Furthermore, for inputs generating hash values with a block size smaller than 48, the similarity score is being limited by the length of the smaller base64 character string multiplied by $\frac{b}{3}$. This limitation is justified by the fact, that in cases with a small block size and a similarity hash of a short length, the similarity score has a high probability to be a false positive. We consider the inconsistencies in the comparison function, which have been mentioned so far, as design choices and do not analyze them any further.

There is another inconsistency in the comparison of hash values, which we do consider. For two similarity hash values that have both their corresponding block sizes in common, the maximum matching score is selected to approximate the similarity of the input files. This means, that two similarity scores are calculated, one for a block size b and one for a block size $2b$. After the calculation of both similarity scores, the maximum is selected.

The similarity corresponding to the block size $2b$, however, results from a comparison of base64 character strings of shorter length and is, therefore, considered to be less accurate. Hence, selecting the maximum similarity score is expected to increase the false positive rate.

Modification `-nomax`: We propose to remove the maximum score selection and always use the distance of the signatures of block size b. This should lead to an output of higher reliability and enhanced accuracy, due to the comparison of longer hash values. Additionally, fewer edit distance calculations are performed which optimizes the comparison process in terms of runtime.

With the release of ssdeep version 2.14 (Kornblum et al., 2017a), the edit distance of two character strings is calculated using a position array to achieve a more efficient runtime. The effectiveness of this optimization has not been assessed yet and will be included separately in our evaluation for the sake of completeness.

Modification `-nopa`: In our evaluation the ssdeep versions without the position array are denoted with `-nopa`, the implementations with the array do not have this suffix. Both implementations are taken from the original ssdeep source code.

Lastly, the implementation contains an optimization, which defines that two similarity hash values are comparable only if they have a common substring of length 7. This substring search optimization decreases the set of comparable hash values considerably. However, while the runtime of the database lookups is improved by reducing the number of Levenshtein distance calculations, enforcing a false negative and successfully obfuscating a file from the filtering process becomes more feasible with this modification in place. By modifying every seventh segment, the similarity hash values have no common substring of length 7 and their distance will not be calculated at all.

Modification `-nocommonsub`: Therefore, depending on the application, it should be considered to remove the substring search optimization. This is expected to enhance the security and reliability of the similarity approximation. With `-nocommonsub` we indicate implementations, which do not use the common substring optimization.

3.2. Errors

As mentioned before, this section presents an implementation error, found in the current ssdeep implementation as well as our proposed fix.

Last segment bug: If the last byte of the input file triggers a new initialization of a segment, the resulting segment is appended to the similarity hash value. An empty segment however contains no information about the input file. This can also cause an incorrect selection of a block size, resulting in a similarity hash value of length 32 and a loss of information. The error is within the fuzzy digest construction, in which a base64 character is appended to the digest if the rolling hash value h does not equal 0 after processing the input. In addition, in the case of $h = 0$, no base64 character is appended, leading to a missing segment and worsening the coverage.

Modification `-bugfix`: With the removal of this error, the reliability is enhanced without affecting performance. Most of the time, the error either appends an additional empty segment or misses out on the last character but rarely leads to an incorrect selection of the block size. Therefore, we consider it a minor error in the output.

3.3. Optimizations

In this section, we propose various improvements, which we implemented and assessed in our evaluation.

Increasing trigger progression constant: With a trigger progression constant of 2 ssdeep generates two signatures for a block size b and double the block size $2b$.

Modification `-4b`: Increasing the progression constant to a value of 4 could potentially improve the algorithm in terms of runtime. This is because a suitable block size with a minimum of 32 characters will be determined more efficiently. In addition, fewer signatures and, therefore, fewer FNV hash values have to be calculated when processing the input. Another positive effect is that the hash value length is increased to a value between 32 and 128 base64 characters, which is expected to lead to a more reliable approximation of the similarity.

For the comparison with a hash database, the number of hashes being compatible for a comparison increases. On the one hand, this decreases the false-negative rate and optimizes the accuracy property, but on the other hand, it requires additional Levenshtein distance calculations. Moreover, the longer hash values are expected to decrease the runtime efficiency of the edit distance calculation. In our evaluation, we include both, the increased trigger progression constant as well as the original one, to assess the positive and negative effects of this modification.

An approach called FKsum (Chen and Wang, 2008) similarly increased the progression constant to a value of 4 to achieve a faster calculation of hash values. Furthermore, the authors state, that the FNV hash values for a block size b could be used to calculate the FNV hash value for double the block size $2b$. Thus, it would be possible to retain a runtime improvement in the SPHF without influencing the comparability of hash values. Unfortunately, the authors give no detailed explanation on how to combine multiple FNV hash values into one. In line with others (Breitinger and Baier, 2012), to the best of our knowledge, we are not aware of a method to perform this combination.

Exchanging the Adler32 checksum: The choice of the rolling hash function is influenced by the runtime and the ability to produce evenly sized segments. For MRSH, Roussev (Roussev et al., 2007) compared the djb2 algorithm with the MD5 and concludes, that the djb2 algorithm appears to be a reasonable choice due to its computational advantage. Breitinger argues that the original Adler32 based rolling hash function is better for ssdeep as there is no rolling version of djb2 (Breitinger and Baier, 2012). This means that the hash has to be recomputed from scratch for every window.

Modification `-djb2`: For this modification, we developed a rolling version of the djb2 algorithm, which does not have the above-mentioned drawback. Since djb2 has fewer operations, we hope to achieve a better runtime than with the Adler32 based algorithm.

The original djb2 algorithm is defined as:

$$h_0 = 5381, \quad h_{k+1} = 33h_k + n_k \bmod 2^{32}, \quad \text{for } k \geq 0,$$

where n_k denotes the k^{th} character of the input. The first window of length 7 corresponds to:

$$h_7 = 33^7 h_0 + \sum_{k=0}^{6} 33^{6-k} n_k.$$

Now, to implement the djb2 algorithm efficiently as a rolling hash function, the subsequent hash value can be calculated using the following:

$$h_8 = (h_7 - 33^7 h_0 - 33^6 n_0) \cdot 33 + n_7 + 33^7 h_0.$$

Note that the calculation above requires only the knowledge of the last character n_0, the one to be removed, and the one added to the rolling window, n_7. As mentioned before, in comparison to the Adler32 checksum, this polynomial djb2 hash function requires fewer operations to process the sliding window.

The djb2 algorithm has been proven to produce evenly sized segments (Roussev et al., 2007) and in theory, a collision-resistant hash function with a great distribution is not of importance. For any chosen rolling hash algorithm, specific sequences of bytes or words in a document determine a segment. Which word or byte sequence exactly sets a trigger point of the input is not significant.

Modification `-poly`: To go one step further, we implemented an even simpler rolling hash based on a 32-bit polynomial function:

$$h = \sum_{k=0}^{6} n_k 32^{6-k}$$

This leads to a minimum number of operations. An advantage of the polynomial is that the type limitation to a 32-bit integer makes the handling of a sliding window superfluous. Again, we hope to improve the overall runtime with this modification.

4. Evaluation

For our evaluation, we took the current ssdeep code as a basis and implemented various versions reflecting different combinations of our proposed modifications. The original source code is denoted with `-original`. Our modified implementations use a combination of similar hyphenated tokens indicating the modifications implemented in the corresponding implementation. For example, `-4b-nomax` is our implementation, which has the modifications `-4b` and `-nomax` included. Our implementations are publicly available on our GitHub repository (Jakobs et al., 2022). Due to space limitations, we cannot provide all results in this paper but focus on the most important ones. The full set of results is also provided in our repository.

Initial evaluations indicated that the removal of the last segment error (`-bugfix`) and the removal of the 32 character limitation of the second signature (`-no32lim`) improve the performance. In the following evaluation, we often use these two modifications together, so that we use the suffix `-refactored` to indicate the combined usage of `-bugfix` and `-no32lim`.

To assess the modifications, we consider the five characteristics mentioned in Section 2, particularly focusing on the most significant properties, which are runtime efficiency and accuracy.

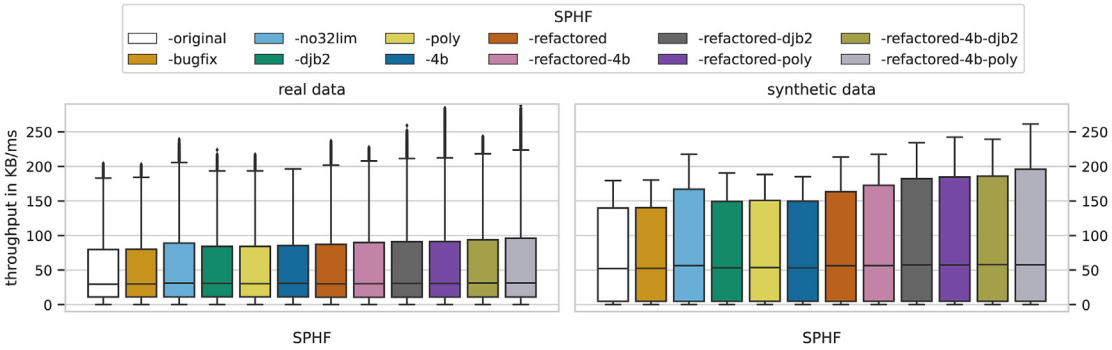

Fig. 1. Runtime results of the SPHF evaluation. The real data corpus consists of files derived from the GovDocs corpus, while the synthetic data corpus consists of randomly generated byte sequences.

4.1. Runtime

In our evaluation of the runtime, we differentiate between the runtime of the SPHF, the runtime of the CF and the overall runtime, i.e. SPHF + CF. We did this to be able to examine the modifications of the SPHF and the CF independently. A combined version is included because a prior hash generation with a subsequent comparison reflects the actual usage of similarity hashing in practice. Here, we evaluate if an efficient SPHF could generate hash values that hurt the efficiency of the CF.

4.1.1. SPHF runtime

Data corpus: The evaluation of the runtime is performed on a set of real data and a set of synthetic data. The real data is a set of files derived from the first 104 (000–103) directories of the GovDocs corpus (Garfinkel et al., 2009). The resulting corpus consists of 100000 files summing up to 58.2 GB of data. The synthetic data corpus consists of randomly generated byte sequences ranging from 10 bytes to 20 MB in size.

To measure the runtime, we hashed each input 10 times to account for variance. Fig. 1 displays the throughput of the implementations on the y-axis for the real data on the left and for the synthetic data on the right-hand side. Additionally, Table 1 shows the average throughput in the rightmost columns. Note that the table shows the mean values while the boxplots indicate the median and the distribution of the values as well as outliers.

With `-bugfix`, the throughput of the algorithm remains similar, because adding the last character requires no additional computations. The modification `-no32lim`, as expected, increases the throughput by approximately 14 % in the case of real data. This is due to the removal of additional FNV hash calculations.

An exchange of the rolling hash function using one of the modifications `-djb2` or `-poly` slightly increases the throughput. Although both rolling hash functions are less complex, the increase is unexpectedly low. This is because processing the input requires the computation of a single rolling hash while, at the same time, multiple FNV hash values have to be calculated. Additionally, the performance is directly linked to the occurrence of trigger points as

Table 1
Summary of the evaluation.

Modification	Properties of the hash database					Average throughput (KB/ms)	
SPHF	Compression ratio	$avg(len(s1))$	$avg(len(s2))$	Signature ratio $avg\left(\frac{len(s1)}{len(s2)}\right)$	Backward-compatibility	real data	synthetic data
`-original`	4548.82	50.51	22.65	3.01	*n/a*	48.629	69.619
`-bugfix`	4540.41	50.61	22.78	2.99	mostly	48.846	69.979
`-no32lim`	4517.36	50.51	23.54	2.98	partially	55.234	82.686
`-poly`	4611.33	54.95	16.97	14.12	no	52.117	73.701
`-djb2`	4502.57	50.57	23.86	2.39	no	51.640	73.483
`-refactored`	4508.91	50.61	23.68	2.95	partially	54.247	81.400
`-refactored-poly`	4566.10	50.05	18.12	13.90	no	58.504	89.319
`-refactored-djb2`	4446.77	50.56	25.49	2.33	no	57.608	89.068
`-4b`	3879.12	78.41	16.96	6.99	no	51.424	72.495
`-refactored-4b`	3851.10	78.52	17.94	6.84	no	55.513	83.577
`-refactored-4b-poly`	3569.33	94.75	14.08	30.81	no	61.136	94.083
`-refactored-4b-djb2`	3824.28	77.68	19.79	5.23	no	58.876	89.751

CF	Comparability & Resistance	Runtime (average) SPHF: **`-original`**	Runtime (average) SPHF: **`-refactored-djb2`**
`-original`	–	97.441	98.875
`-nocommonsub`	incr. resistance	115.676	117.024
`-nomax`	decr. fp-rate	99.544	99.007
`-4b`	incr. resistance & detection	128.120	132.535
`-nocommonsub-nomax`	incr. resistance & decr. fp-rate	116.050	115.703
`-4b-nocommonsub`	incr. resistance & detection	169.223	174.557
`-4b-nomax`	incr. resistance & detection & decr. fp-rate	126.688	131.755
`-4b-nocommonsub-nomax`	incr. resistance & detection & decr. fp-rate	164.286	170.443

they determine the number of FNV hashes to be calculated. The number of trigger points also explains the variance in throughput on the synthetic data corpus. The file sizes in this dataset are wider and more evenly distributed than the ones in the real dataset. While smaller files potentially lead to a better runtime due to fewer signature initializations, larger files tend to have more initializations.

The synthetic data also shows a greater throughput on average. This is due to the randomness provided by the input and the occurrences of trigger points. The minimum number of segments for double the block size $2b$ is reached faster than with the real dataset and the block size b and its corresponding signature are discarded. This leads to a higher throughput.

The increase in the throughput reached with the modification `-4b` is rather little. The average input size of the real data corpus and the synthetic data corpus is relatively small. For small files, the number of block sizes is already limited by the saltus function. Hence, the final block size of the similarity digest is found rapidly and this inhibits the effect of the increased progression constant of the `-4b` modification. Additionally, increasing the signature length to 128 characters leads to a delay in the discarding of block sizes and signatures.

Different independent modifications improve the runtime of the SPHF as shown by implementations where multiple modifications have been combined. The increased throughput of some modifications is lower than expected. This is caused by the average input size and the occurrence of trigger points. Since the modification `-refactored` only influences the generated hash value in very rare cases—and then just marginally—and optimizes the SPHF, in the remainder of this paper, it is often used as the basis on which further modifications are appended.

4.1.2. CF runtime

In the evaluation of the CF runtime, we did not simply measure the time required to perform the Levenshtein distance comparison. Instead, we opted to measure the actual use case of the algorithm, where a hash value is compared to a hash value database. For this matter, a hash value database consisting of 1 00 000 hash values has been created using the aforementioned corpus. The hash values to be compared with the database are extracted from either the first or the 105th directory of the GovDocs corpus. This ensures that we look up hashes that are in the database and hashes which are not. Again, we measured the mean execution time of 10 comparisons of an algorithm for both, the 100 hash values, which are in the database, and the ones which are not.

Fig. 2 shows the result of our evaluation. Each box indicates the average execution time of the comparisons for the respective algorithms. As mentioned in Section 3.3, we expect that the modification `-4b` increases the runtime of the Levenshtein distance comparison. Because of this, we show the results for both SPHF variants for each of the implementations.

The modifications `-nocommonsub` and `-nopa` significantly increase the runtime of the comparison with a hash value database. While the first optimization leads to a weaker obfuscation resistance and is expected to increase the false-negative rate, the calculation of the Levenshtein distance using a position array has no drawbacks and simply achieves a faster comparison.

Implementations modified by `-nomax` achieve only a moderate performance increase. There are two reasons for this: first, the modification influences only the comparison of hash values having two common block sizes. Second, in some cases, the algorithm generates hashes with a second signature of a very short length. Omitting the comparison between these short signatures only marginally improves the runtime. However, since the first signature of the similarity hash is larger, we expect to achieve a more accurate approximation. Therefore, the modification not only decreases the runtime of the comparison but also might improve the accuracy and reliability of the algorithm.

Implementations with the modification `-4b` have a larger range of possible comparisons and average signature length. Thus, a performance decrease of approximately 30 % is observed when compared to the original version. The real benefit of increasing the constant lies in the increased signature length and the range of possible comparisons. With the former an increased accuracy and with the latter, a decreased false-negative rate is anticipated.

Concluding, the current implementation of the Levenshtein distance using a position array successfully improves the efficiency. Increasing the trigger progression constant as well as removing the substring search optimization lead to a significantly increased runtime. However, these variants are expected to reduce the false-negative rate, strengthen the obfuscation resistance and increase the accuracy. So, depending on the use case, these modifications might still be beneficial (e.g. to find fragments or to cluster similar files). The removal of the selection of a maximum distance achieves a moderate improvement in the runtime with the assumption that the accuracy is also enhanced.

4.1.3. Combined SPHF and CF runtime

Since the choice of the SPHF influences the generated hash values and, hence, the performance of a database comparison, we conducted a third evaluation. Here, the previous evaluation of the CF is extended with the hash value generation using the corresponding SPHF. Since the hash calculation is performed only once, the average size of the inputs is relatively small and the subsequent database comparison consists of up to 100 000 distance calculations, an efficiency improvement of the SPHF is barely noticeable, as can be seen in Fig. 3.

Similar to the previous evaluation, `-nocommonsub` and `-4b` lead to a significantly increased execution time, while `-nomax` slightly improves the comparison time. The performance of the comparison function is not considerably affected by the choice of the rolling hash function. It can also be observed that the modification

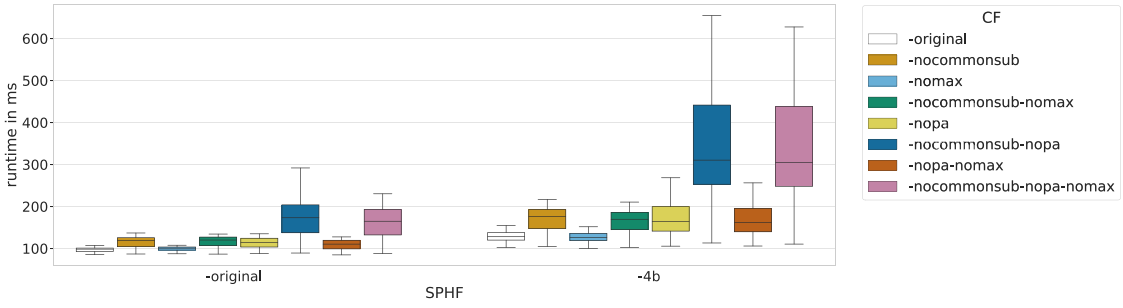

Fig. 2. Average runtime of hash lookups in a database to evaluate the CF performance.

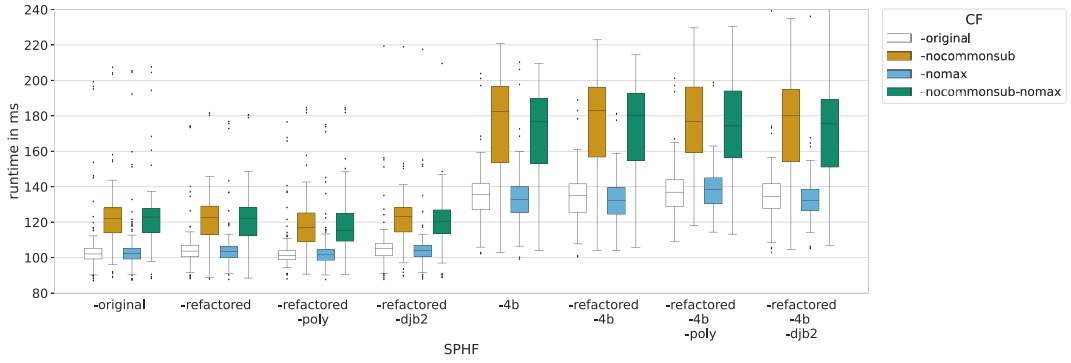

Fig. 3. Runtime of hash lookups with prior computation of the target hash.

`-refactored` leads to a slightly increased execution time. This is due to the larger hash value lengths on average when removing the 32-character limitation. The same can be observed for the modification `-refactored-djb2`, in which the moderate increase in runtime is a result of generating fewer signatures of short size, that are unfit for comparison, as we will show in the subsequent Section 4.2.

Concluding, for typically sized user files (some MB of size), the database lookup accounts for most of the time taken. Parallelization of the lookups or similar means to improve them seems like a worthwhile future work.

4.2. Compression

We evaluated the compression property employing the database of 100 000 hash values, which we previously generated for the evaluation of the efficiency of the CF.

Table 1 shows the compression ratio for each implemented modification as well as the average length of signatures. The compression ratio is calculated as the overall size of the 100 000 files divided by the size of the hash value database. In addition to those values, the average ratio between the two signatures for the block sizes b and $2b$ is listed. A suitable rolling hash function is expected to have a ratio close to the value 2 because the signature corresponding to the block size b should be close to double the size of the signature corresponding to the block size $2b$. A value close to 2, thus indicates a favored, regular distribution of trigger points. Implementations modified with `-4b` are expected to have a value close to 4.

The modification `-refactored` slightly decreases the average compression ratio. This is due to the removal of the 32 character limitation of the second signature as can also be seen on the average signature lengths of the modification `-no32lim`. The removal of the minor error (`-bugfix`) is noticeable as well, leading to an increased average length for both signatures, due to the occasional absence of the last segment.

The implementations using the modification `-4b` have a larger hash length on average due to the increased limitation to at most 128 base64 characters. The difference to the original is not as weighty as expected.

For the compression ratio, there is no noteworthy difference between the different rolling hash functions. Yet, when inspecting signature lengths and ratios, the modification `-djb2` outperforms the `-original`. With a ratio of 2.39 for `-djb2` and 2.33 for `-refactored-djb2`, the value is close to the value 2. This confirms, that djb2 leads to hash values with relatively evenly sized segments.

In addition, Fig. 4 illustrates that modifications using the djb2 algorithm generate fewer hashes that are unfit for comparison. We

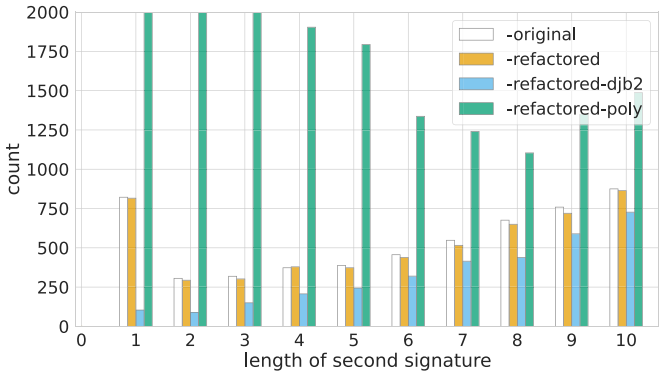

Fig. 4. Signature lengths.

consider hash values with a second signature of short size to be unfit to represent an input. For this matter, Fig. 4 displays the number of such hash values found in the aforementioned hash value databases. The x-axis indicates the length of the signature corresponding to the block size $2b$ and the y-axis the corresponding count. Since the determination of trigger points is directly linked to the choice of the rolling hash function, we show only such modifications in the figure. The `-original` generates some signatures of length 0 due to the minor error, which has been removed with the modification `-bugfix`.

We also evaluated the roughly 21 000 000 hashes provided in the NSRL ssdeep dataset (National Institute of Standards and Technology, 2019). The results are consistent with the results we obtained with our dataset. Here, we found 7740 signatures with a length of 0, 233 664 with a length ≤ 5 and 709 322 with a length ≤ 10.

The polynomial hash function (`-poly`) leads to a signature ratio of 14.12 indicating that it produces hash values with a poor and irregular segmentation of the input.

To conclude, the modifications `-no32lim` and `-bugfix` slightly increased the hash length as expected. For a more reliable output, the 4b variants can provide a more suitable approximation depending on the use case, due to the larger hash length. This is the case if false negatives are of concern, e.g. in the case of detecting malicious software. If the runtime performance of the comparison is prioritized and false negatives are not of concern, the original variant using a progression of 2 is more suitable. Exchanging the rolling hash function with djb2, increases the reliability of the output too, because the signatures for double the block size 2b are more suitable for comparisons. In contrast, the polynomial turned out to be unsuitable at all.

4.3. Accuracy

Data corpus: The evaluation of the accuracy is performed on a set of synthetic data and a set of real data. For the synthetic data corpus, randomly generated byte sequences of the sizes 1, 10 and 100 KB as well as 1 and 10 MB have been created. Each input has been modified using three different types of modifications: the deletion, insertion and substitution of byte sequences. Additionally, each type of modification is performed in one out of three different sections of the input, i.e. in the first, second, or last third of the byte sequence. The size of the modification increases step-by-step from 0 to 100 % of the size of the original input leading to a total of 900 modified versions for each size. The real data corpus consists of fragments of real data derived from the t5 corpus (Roussev, 2011). Here, we copied fragments from one out of three sections of the original file. This reflects the fragment search case of an application as well as the deletion of byte sequences. Since we cover two use cases and the deletion of bytes represents one of the worst-case scenarios for the algorithm (insertion and deletion have a high probability of changing the block size), we opted to show this example in the paper, while the other evaluations can be found on GitHub. Overall, the real data corpus in total consisted of 66 870 fragments of 4458 original input files.

There are different ways to define the accuracy of a similarity hashing algorithm. In our evaluation, we focus less on the meaning of the similarity score, but rather on its variance. Ideally, the similarity score should not vary too much after a certain type of modification has been applied to the test files. The more it does, the more unreliable and inaccurate we consider the actual values to be.

A detailed evaluation of the synthetic data is provided in our GitHub repository. The key takeaway from the evaluation is that due to the randomness of the synthetic data, the different rolling hash implementations exhibit very similar behavior.

Fig. 5 shows the results of the evaluation of the real data corpus. The upper x-axis labels indicate the size of the fragments relative to the size of the original file. The y-axis depicts the similarity score output by the ssdeep algorithm. Finally, the x-axis labels on the bottom describe the SPHF configuration for the various CF variants.

The modifications `-djb2` and `-poly` alter the average accuracy only a little. For the 50 % and 75 % fragments, all variants of the ssdeep algorithm lead to similar scores on average. Still, it appears that the range of similarity scores generated by the polynomial hash function is larger than with the Adler32 checksum or the djb2 algorithm. The reason for the variance in the similarity score is the poor occurrence of trigger points (cf. Section 4.2).

Surprisingly, the selection of the maximum similarity score is noticeable even for smaller fragments (50 % and less). In this case, the majority of similarity hash values are expected not to have two common block sizes. Yet, the selection leads to a slightly increased output on average. This indicates that, if two similarity hash values have two common block sizes, the calculated similarity score often results from the comparison of the second signatures, leading to a higher output.

The increase of the trigger progression constant leads to more hash values with common block sizes and, thus, a higher detection rate. This becomes particularly obvious for the fragments of the size of 25 %.

For larger fragments, the removal of the common substring optimization is not noticeable because most of the fragments have common substrings of length 7. Only for very small fragments, the optimization shows different results. With the removal, more comparisons are carried out and fragments of smaller size remain detectable.

Summarizing the accuracy evaluation, all ssdeep variants often calculate a similarity score of 100 %, even for smaller fragments. This shows a susceptibility for a high false-positive rate of the ssdeep algorithm. One reason for this is the limitation of the length of the similarity hash, which can lead to a merging of multiple segments. Modifying the merged segments will only affect a single character of the similarity hash independent of the size of modification and this can, by chance of $\frac{1}{64}$, be the same character as before, so that the modification goes unnoticed. Another reason for the higher similarity scores is the poor segmentation of the input caused by the selection of the trigger points. This can be seen from the modification using the polynomial hash function, which showed great variance in the output caused by an irregular segmentation.

As in Section 4.2, the polynomial hash function, again, is considered to be not suitable as a rolling hash function. Having said that, the evaluation showed that for any of the three rolling hash

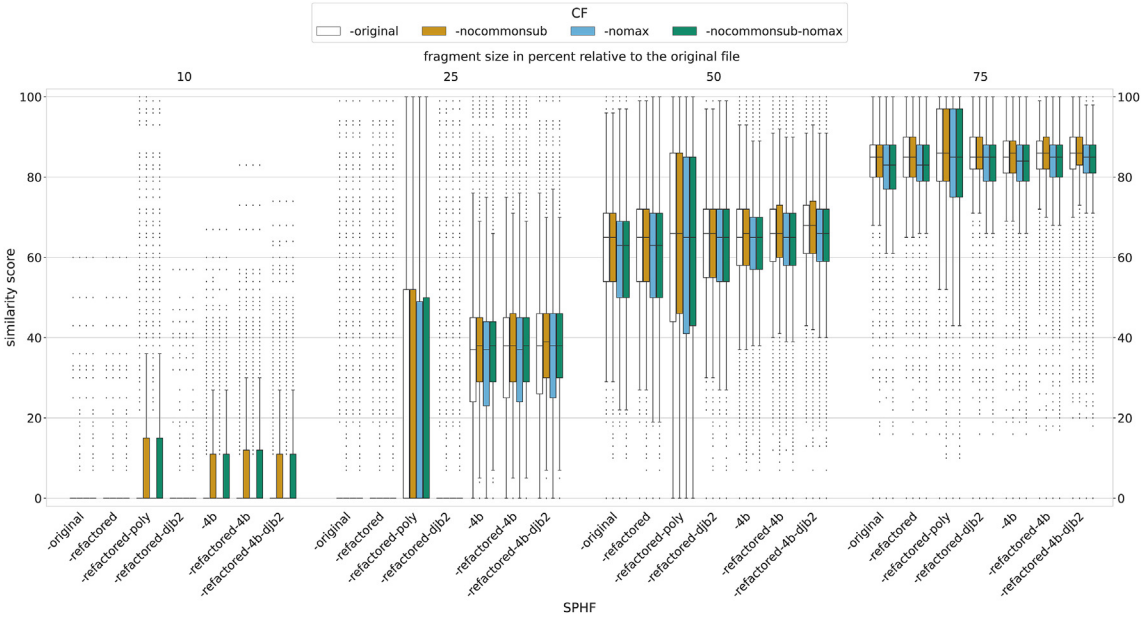

Fig. 5. Accuracy evaluation on real data fragments.

functions, ssdeep can be considered only moderately accurate due to a limited hash length and its dependency on well-selected trigger points.

4.4. Security properties

The general security properties, obfuscation resistance and coverage, are mostly predetermined by the original ssdeep algorithm. Our modifications do not alter the core algorithm, but still have minor effects on the security properties, which will be discussed in the following sections.

4.4.1. Obfuscation resistance

The proposed modifications to the ssdeep algorithm do not cause any particular improvement of the obfuscation resistance. As the limited hash length is wanted in ssdeep to have an efficient distance calculation, a modification to enforce false negatives by inserting segments at the beginning of the input remains possible. Removing the substring search optimization using `-nocommonsub` or increasing the hash value length with the modification `-4b` strengthens the obfuscation resistance but by no means leads to a secure algorithm.

4.4.2. Coverage

In addition, segments are merged to limit the hash length to 64 base64 characters (128 characters in the case of `-4b`), for the first and 32 base64 characters for the second signature. The last base64 character of a signature produced by ssdeep can, therefore, be a result of a very large part of the input. Modifying this part however only alters a single character, independent of the size of modification. This means that some bytes influence the hash value more than others do. With the removal of the 32-character limitation of the second signature, the coverage is more evenly distributed. Further, with `-bugfix` a missing segment is appended to ensure full coverage. Implementations using the combined modification `-refactored`, therefore, ensure full coverage for both signatures and provide a second signature that is expected to represent evenly sized segments.

To demonstrate the discussed security issues, we constructed several examples files, which are provided in the GitHub repository supplementing this paper. For instance, we created modified versions of a PDF document, where we only introduced minor changes in the header, which have a low or zero similarity score when compared with the original. We also created versions with minor changes, which are not comparable at all with the original document. Lastly, we constructed example files highlighting issues concerning the coverage as a result of the error documented above.

5. Conclusion

In this paper, we assessed the current implementation of ssdeep, version 2.14.1, as well as the underlying algorithm. We highlighted inconsistencies between the implementation and the original publication and unfavorable design choices. In addition, we proposed several modifications to improve the algorithm in terms of runtime and accuracy, which we assessed in an extensive evaluation.

For the generation of hash values, we suggested various optimizations, which increase the runtime performance by up to 25 %. The most promising candidate is the variant `-refactored-djb2`. Here, we removed the 32-character limitation of the second signature, fixed an implementation bug and replaced the rolling hash function with a modified version of the djb2 algorithm. In our evaluation, this leads to a significantly faster calculation and the construction of more reliable similarity digests. Especially, this reduced the number of very short signatures. For a higher detection rate, but at the cost of slower comparisons, the variant `-refactored-4b-djb2` can be used. The compare function has also been assessed and improved. With the removal of a maximum distance calculation, we were able to enhance the runtime and the reliability of the outputs.

During our evaluation, we observed that the results varied depending on the file type. This is in line with prior works and should be addressed in more detail with a larger and more contemporary corpus covering more different file types and sizes.

The backward compatibility with previously generated hash databases for our modifications is given in Table 1. The term “mostly” means that the hashes differ in very rare cases, while the term “partially” describes a somewhat more likely deviation from the hashes generated with the original version.

To conclude, the best combination of modifications depends on the use case. When detecting modified suspicious files, `-refactored-4b-djb2-nocommonsub` is a suitable choice. The process of filtering and minimizing uninteresting, extraneous data requires a low false-positive rate, which is provided by `-refactored-djb2-nomax`. As mentioned in the Introduction, we believe that further improvements are likely only possible by modifying the core algorithm. In this case, however, one could argue that such modifications result in a new algorithm that cannot be called ssdeep anymore.

All implementations, the evaluation data corpus, examples files highlighting issues with ssdeep and further detailed analyses and figures are published on GitHub and Zenodo respectively (Jakobs et al., 2022).

On a final note, during the profiling of our modifications, we observed at least one obscure effect, which we pin on the compiler: adding a counter variable into a loop body sped up the execution of the entire algorithm by roughly 20 %. We did not explicitly focus on pure code optimizations here, but this observation suggests that general profiling of the ssdeep code may be advisable.

Lastly, we argue that future research should also focus on more efficient database lookups of similarity hashes like Wallace did in his Virus Bulletin article (Wallace, 2015).

Acknowledgements

We would like to thank our shepherd, Simson Garfinkel, and the anonymous reviewers for their valuable feedback and help to improve our paper.

References

Breitinger, F., Baier, H., 2012. Performance issues about context-triggered piecewise hashing. In: Digital Forensics and Cyber Crime: Proceedings of the 3rd International Conference on Digital Forensics and Cyber Crime (ICDF2C). Springer, pp. 141–155.

Breitinger, F., Baier, H., 2013. Similarity preserving hashing: eligible properties and a new algorithm MRSH-v2. In: Digital Forensics and Cyber Crime: Proceedings of the 4th International Conference on Digital Forensics and Cyber Crime (ICDF2C). Springer, pp. 167–182.

Breitinger, F., Guttman, B., McCarrin, M., Roussev, V., White, D., 2014. Approximate Matching: Definition and Terminology (NIST Special Publication 800-168).

Chen, L., Wang, G., 2008. An efficient piecewise hashing method for computer forensics. In: First International Workshop on Knowledge Discovery and Data Mining (WKDD 2008). IEEE, pp. 635–638.

Fowler, G., Noll, L.C., Vo, K.P., Eastlake, D., Hansen, T.. The FNV non-Cryptographic hash algorithm. https://datatracker.ietf.org/doc/html/draft-eastlake-fnv-17.

Garfinkel, S., Farrell, P., Roussev, V., Dinolt, G., 2009. Bringing science to digital forensics with standardized forensic corpora. Digit. Invest. 6, S2–S11.

Jakobs, C., Lambertz, M., Hilgert, J.N., 2022. Github: fkie-cad/ssdeeper. https://github.com/fkie-cad/ssdeeper.

Kornblum, J., 2006. Identifying almost identical files using context triggered piecewise hashing. Digit. Invest. 3, 91–97.

Kornblum, J., Grohne, H., 2013. Merging in helmut's branch for thread safety · ssdeep-project/ssdeep@9d51afb. https://github.com/ssdeep-project/ssdeep/

commit/9d51afb8be01db4a2532bdce80cb0500991b447e.
Kornblum, J., Grohne, H., Oi, T., 2017a. Merge "tsukasa" branch · ssdeep-project/ssdeep@7a62735. https://github.com/ssdeep-project/ssdeep/commit/7a627351e23449ddba2ed1b27d7e4c9bb60dcf52.
Kornblum, J., Grohne, H., Oi, T., 2017b. ssdeep Project — ssdeep - fuzzy hashing program. https://ssdeep-project.github.io/ssdeep/.
Meta, 2022. /similar_malware - ThreatExchange - documentation - meta for developers. https://developers.facebook.com/docs/threat-exchange/reference/apis/similar-malware/v12.0.
National Institute of Standards and Technology, 2019. ssdeep Datasets. https://www.nist.gov/itl/ssd/software-quality-group/ssdeep-datasets.
OASIS Open, 2021. STIX Specification Version 2.1. https://oasis-open.github.io/cti-documentation/resources.html.
Roussev, V., 2010. Data fingerprinting with Similarity digests. In: Chow, K.P., Shenoi, S. (Eds.), Advances in Digital Forensics VI. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 207–226.
Roussev, V., 2011. An evaluation of forensic similarity hashes. Digit. Invest. 8, S34–S41. https://doi.org/10.1016/j.diin.2011.05.005. https://www.sciencedirect.com/science/article/pii/S1742287611000296 (The Proceedings of the Eleventh Annual DFRWS Conference).
Roussev, V., Richard III, G.G., Marziale, L., 2007. Multi-resolution similarity hashing. Digit. Invest. 4, 105–113.
Tridgell, A., 2002. Spamsum README. http://samba.org/ftp/unpacked/junkcode/spamsum/README.
VirusTotal. VirusTotal Developer Hub - ssdeep. https://developers.virustotal.com/reference/ssdeep.
Wallace, B., 2015. Optimizing ssDeep for use at scale. Virus Bull. https://www.virusbulletin.com/virusbulletin/2015/11/optimizing-ssdeep-use-scale.