mathematics

MDPI

Article

An Improved Bytewise Approximate Matching Algorithm Suitable for Files of Dissimilar Sizes

Víctor Gayoso Martínez, Fernando Hernández-Álvarez and Luis Hernández Encinas *

Institute of Physical and Information Technologies (ITEFI), Spanish National Research Council (CSIC), Serrano 144, 28034 Madrid, Spain; victor.gayoso@iec.csic.es (V.G.M.); fernando.hernandez@iec.csic.es (F.H.-Á.)

* Correspondence: luis@iec.csic.es; Tel.: +34-91-561-88-06

Received: 21 February 2020; Accepted: 30 March 2020; Published: 2 April 2020

Abstract: The goal of digital forensics is to recover and investigate pieces of data found on digital devices, analysing in the process their relationship with other fragments of data from the same device or from different ones. Approximate matching functions, also called similarity preserving or fuzzy hashing functions, try to achieve that goal by comparing files and determining their resemblance. In this regard, ssdeep, sdhash, and LZJD are nowadays some of the best-known functions dealing with this problem. However, even though those applications are useful and trustworthy, they also have important limitations (mainly, the inability to compare files of very different sizes in the case of ssdeep and LZJD, the excessive size of sdhash and LZJD signatures, and the occasional scarce relationship between the comparison score obtained and the actual content of the files when using the three applications). In this article, we propose a new signature generation procedure and an algorithm for comparing two files through their digital signatures. Although our design is based on ssdeep, it improves some of its limitations and satisfies the requirements that approximate matching applications should fulfil. Through a set of ad-hoc and standard tests based on the FRASH framework, it is possible to state that the proposed algorithm presents remarkable overall detection strengths and is suitable for comparing files of very different sizes. A full description of the multi-thread implementation of the algorithm is included, along with all the tests employed for comparing this proposal with ssdeep, sdhash, and LZJD.

Keywords: approximate matching; context-triggered piecewise hashing; edit distance; fuzzy hashing; LZJD; multi-thread programming; sdhash; signatures; similarity detection; ssdeep

1. Introduction

Digital forensics is the branch of Mathematics and Computer Science in charge of identifying, recovering, analysing, and providing conclusions about digital evidence found on electronic devices. When inspecting the content of a computer or a mobile phone, the first step for the digital forensics expert is to reduce all the data available, extracting the important information that can be analysed more efficiently [1].

An initial strategy to obtain that reduction consists in using cryptographic hashing functions such as MD5 or SHA-1. Even though those algorithms are not recommended for cryptographic purposes, they are still valid for determining if two files are the same, considering that the probability for two files to have the same hash value is negligible. Precisely due to this reason, NIST (National Institute of Standards and Technology) developed a database in the early 2000s, called NSRL (National Software Reference Library), which contains hash values of sets of files of several trusted operating systems [2]. Nevertheless, these cryptographic hashing functions face a common problem when comparing files. If one of the files is modified even just in one byte, the final outcome of the comparison is negative.

In contrast to cryptographic hashing functions, approximate matching functions [3], also known as similarity preserving hashing (SPH) or fuzzy hashing functions, try to detect the resemblance between two files by linking similar inputs to similar outputs, indistinctly called in this context similarity signatures, fingerprints or digests [3]. These functions, which analyse files at byte level, are useful to compare a large variety of data and detect similar texts and even embedded objects (e.g., an image in a Word or OpenDocument text file) or binary fragments (e.g., a virus inside a file, a specific data packet in a network connection or similar content in audio files). By using a diversity of methods (for example, computations with a sliding window, as it will be described in next sections), approximate matching functions are able to identify if even a single byte is changed.

In computer forensics, ssdeep is the best-known bytewise approximate matching application, and it is considered by some researchers as the de facto standard in some cybersecurity areas [4]. The implementation of ssdeep allows the generation of the signature of files (where the generated signature depends on the actual content of the file) and to compare two signature files or one signature file with a data file. The results in both cases are the same, using one method or the other depends on the data available to the user. However, even though ssdeep represented an important achievement in similarity detection techniques, and it is relatively up to date (at the time of writing this text, the latest version is 2.14.1, released in November 2017 [5]), during the last years some limitations have been highlighted by several researchers, publishing different enhancements or alternative theoretical approaches (see, for example, References [6–13]).

In this article we have delved on the most important limitations of ssdeep, sdhash, and LZJD, which are mainly the impossibility of comparing files of very different sizes in the case of ssdeep and LZJD, the scarce relationship between the comparison score obtained and the actual content of the files when using the three applications in some cases, and the excessive size of sdhash and LZJD signatures. As a result of our research, we are presenting in this contribution a signature generation procedure based on the one implemented in ssdeep had but that overcomes its design limitations, together with an easy-to-implement algorithm for comparing the content of two files through their digital signatures. We can confirm that, based on the list of requirements that, in our opinion, any similarity search function should fulfil, our algorithm provides results better adjusted to different situations than ssdeep and is able to compare any pair of files regardless of their respective size. Besides, our proposal can manage some signatures that represent certain special cases that include different content swapping schemes.

In order to obtain meaningful results, in addition to ssdeep we have also considered sdhash and LZJD in our tests. sdhash is also very popular and is representative of a completely different theoretical approach in approximate matching algorithms, as it uses Bloom filters in order to find the features that have the lowest empirical probability of being encountered by chance [14]. On the other hand, LZJD is a recent alternative that uses the Lempel-Ziv Jaccard distance [15]. As a result of the comparison with sdhash and LZJD, we can state that our algorithm provides better results regarding the recognition of the proportion of a file which is present in another file while generating significantly smaller signatures in almost all the cases where those two algorithms are used.

This article represents an improved version of the information presented as a PhD thesis by one of the authors in 2015 [16], where SiSe (Similarity Search) was first described (this thesis has not been published but it is available as open access at http://oa.upm.es/39099/). Compared to that work, this contribution features a different signature comparison algorithm and new tests, some of them using FRASH (Framework to test Algorithms of Similarity Hashing), a reputed tool designed for comparing this type of applications. As another novelty, this article includes the design details of the multi-threading C++ implementation of the proposed algorithm (allowing researchers to inspect the code of our implementation, uploaded to GitHub [17]), compares the application to the latest version of ssdeep (which in the last years has evolved and fine-tuned its capabilities), and broadens the comparative spectrum by adding LZJD to the test set.

The rest of this paper is organised as follows—Section 2 discusses the related work. Section 3 reviews the most significant features of ssdeep. In Section 4, we provide a complete description of our proposed algorithm. Section 5 describes the C++ implementation of SiSe, including the multi-thread design. Section 6 contains the ad-hoc tests that we have performed with a selected group of files in order to compare the similarity detection capabilities of our proposal to those of ssdeep, sdhash, and LZJD. Section 7 contains the tests performed with the FRASH framework and a well-known dataset. Finally, Section 8 summarises our conclusions about this topic.

2. Related Work

SPH functions can be divided into four categories [18]—Block-based hashing (BBH) functions, context-triggered piecewise hashing (CTPH) functions, statistically-improbable features (SIF) functions, and block-based rebuilding (BBR) functions, as described in the following paragraphs.

BBH functions generate and store cryptographic hashes for every block of a chosen fixed size (e.g., 512 bytes). In the next step, the block-level hashes from two different inputs are compared counting the number of common blocks and giving a measure of similarity between them. An implementation of this kind of fuzzy functions was developed by Nicholas Harbour, who created a program called dcfldd [19]. This function divides the input data into several blocks of a fixed length and calculates the corresponding cryptographic hash value for each of them. Even though this approach is very efficient from a computational point of view due to its simplicity, a single byte insertion or deletion at the beginning of a file could change all the block hashes, making this scheme too vulnerable.

The technique behind CTPH was originally proposed by Andrew Tridgell [20], who implemented spamsum, a context-triggered piecewise hashing application devoted to the identification of spam e-mails [21]. Its basic idea consists in locating content markers, called *contexts*, within a binary data object, calculating the hash of each fragment of the document delimited by the corresponding contexts, and storing the sequence of hashes. Thus, the boundaries of the fragments are based on the actual content of the object and not determined by an arbitrarily fixed block size. In 2006, based on spamsum, Jesse Kornblum developed ssdeep [22], one of the first programs for computing context-triggered piecewise signatures. This algorithm generates a matching score in the range 0–100. This score must be interpreted as a weighted measure of how similar these files are, where a higher result implies a greater similarity of the files [23].

The SIF approach is based on the idea of identifying a set of features in each of the objects under study and then comparing the features, where a feature in this context is a sequence of consecutive bits selected by some criteria from the file that stores the object. As a practical implementation of the concept, Vassil Roussev decided to use entropy in order to find statistically-improbable features [24]. With this idea, he proposed a new algorithm called sdhash, whose goal was to pick object features that are least likely to occur by chance in other data objects [25]. This algorithm produces a score between 0 and 100 that, according to its author, must be interpreted as a confidence value indicating how certain the tool is that the two data objects under comparison have non-trivial amounts of commonality [26].

A more recent example of this approach is LZJD, a proposal made by Edward Raff and Charles K. Nichola [15]. LZJD uses the same command-line arguments as sdhash but implements a different distance function, the Lempel-Ziv Jaccard distance [15]. According to its author, LZJD scores range from 0 to 100 and can be interpreted as the percentage of bytes shared by two files, which makes it comparable to the other applications considered in this contribution.

In turn, BBR functions use external data, which are blocks chosen randomly, uniformly or in a fixed way, in order to rebuild a file. The process compares the bytes of the original file to the chosen blocks and calculates the differences between them, using the Hamming distance or any other metric. Two of the best known implementations of this approach are bbHash [10] and SimHash [27].

It is important to mention that, while there are other high-level information retrieval techniques based on the semantic analysis of the content (e.g., see References [28–30]), SPH tools work at a lower level by directly analysing the value of the content as a byte sequence.

When analysing the similarity between two files, it is important to distinguish the concepts of resemblance and containment. While resemblance indicates how much an object looks like another one, containment indicates the presence of an object inside another one [31]. SPH applications adjust their results to either one of these concepts or both. As it will become clear later, our proposal uses a combination of both concepts.

Other interesting approach developed in recent years is TLSH (Trend Micro Locality Sensitive Hash) [32], a Locality Sensitive Hashing (LSH) application which is related to work done in the data mining area. LSH algorithms classify similar input items into the same groups with high probability, maximizing hash collisions [33]. However, while ssdeep, sdhash, and LZJD provide a similarity score between two digests in the range [0–100], in TLSH a distance score of 0 represents that the files are identical and higher scores represent a greater distance between the analysed documents. As TLSH does not imposes a limit on the score (according to the authors it can potentially go up to over 1000), it is not possible to directly compare its results to those of the other applications, so we decided not to include it in our tests.

3. Review of Ssdeep

The kernel of the signature generation algorithm in ssdeep is a rolling hash very similar to the one employed in spamsum [21] and rsync [34]. The rolling hash is used to identify a set of trigger points in the file that depend on the content of a sliding window of 7 bytes (i.e., the number and location of the trigger points totally depend on the content that is processed by the application, as every byte is taken into account in the calculations). The algorithm hits a trigger point every time the rolling hash (based on the Adler-32 function [35]), produces a value that matches a predefined condition. A second hashing function based on the FNV (Fowler-Noll-Vo) algorithm [36] is consequently used to calculate the hash values of the content located between two consecutive trigger points. Then, the last 6 bits of each hash value is translated into a Base64 character. The final signature is formed by concatenating the single characters generated at all the trigger points (with a maximum of 64 characters per signature).

The number of characters of the signature is strongly determined by the frequency of appearance of the trigger points. Therefore, the first step that must be completed by the algorithm is to estimate the value of the block size that would produce a final signature of 64 characters. More precisely, the value of the initial block size is the smaller number of the format $3 \cdot 2^n$ (where n is a positive integer value) which is not lower than the value obtained after dividing by 64 the input size in bytes. Another condition is that if the length of the signature generated does not reach 32 characters, ssdeep modifies the block size and executes the algorithm another time to produce a new signature. This process is done repeatedly until the signature obtained is of at least 32 characters

The signature generation algorithm produces two signatures in order to be able to compare a higher number of files. The reason for that is that sometimes similar files have slightly different block sizes. The two signatures are known as the leading signature, which uses a certain value for the block size (the leading block size), and the secondary signature, whose secondary block size value is twice the value of the leading block size.

The signature format produced by ssdeep consists of a header followed by one hash per line. The content of the header used in its latest versions is the following:

```
ssdeep,1.1--blocksize:hash:hash,filename,
```

where the element `ssdeep` identifies the file type, `1.1` informs about the version of the file format (not to be confused with the version of the program), `--` acts as a separator, and the remainder of the line identifies the elements displayed below the header (block size, primary hash, secondary hash, and filename).

In Information Theory, the edit distance between two strings of characters generally refers to the number of operations required to transform one string into the other. There are several ways to define the edit distance, depending on which operations are allowed. The ssdeep edit distance algorithm is based on the Damerau-Levenshtein distance between two strings [37,38].

That distance compares two strings and determines the minimum number of operations necessary to transform one string into the other. The only operations allowed in this distance comparison are insertions, deletions, substitutions of a single character, and transpositions of two adjacent characters [39,40].

In ssdeep, insertions and deletions have a weight of 1, substitutions a weight of 3, and transpositions a weight of 5. For instance, using this algorithm, the distance between the strings "Saturday" and "Sundays" is 5, as it can be seen in the following sequence, where the elements in bold format represent the characters that are added or removed at each step:

$$\text{S}\textbf{a}\text{turday} \xrightarrow{\textit{deletion}} \text{S}\textbf{t}\text{urday} \xrightarrow{\textit{deletion}} \text{Su}\textbf{r}\text{day} \xrightarrow{\textit{deletion}} \text{Suday} \xrightarrow{\textit{insertion}} \text{Su}\textbf{n}\text{day} \xrightarrow{\textit{insertion}} \text{Sunday}\textbf{s}.$$

In practice, a consequence of assigning the value 3 to substitutions and 5 to transpositions is that the edit distance calculated only uses insertions and deletions. Therefore, substitutions and transpositions have a weight of 2 (a deletion followed by an insertion).

One of the limitations that this design has is that, for a specific string, a rotated version of it is assigned many insertion and deletion operations, when with regards to the content they are basically the same (i.e., the content is the same, although the order of the substrings is different). Consider for example the strings "abcd1234" and "1234abcd".

Finally, the score that ssdeep produces is normalised in the range $[0, 100]$, where 100 is associated to a perfect match and 0 to a complete mismatch. If the two signatures have different block sizes, then ssdeep automatically sets the score to 0 without performing further calculations.

In addition to the previous information, ssdeep defines the minimum length of the longest common substring as 7. If the longest common substring length detected during the procedure is less than that value, then ssdeep provides a score of 0.

The source code of ssdeep and implementations for both Windows and Linux are freely available [5], so interested readers can inspect the code and test the application.

4. Design of SiSe

4.1. Key Elements for Improvement

Frank Breitinger and Harald Baier presented in 2012 a list of four general properties for similarity preserving hashing functions [41], which they later extended to the following five characteristics [11]:

- *Compression*: The output must be several orders of magnitude smaller than the input for performance and storage reasons.
- *Ease of computation*: The processes of generating the hash value for a given file and making comparisons between files must be fast.

- *Similarity score*: The comparison function must provide a number representing a matching percentage value, so results for different files can be directly compared.
- *Coverage*: Every byte of the input data must be employed for calculating the similarity score.
- *Obfuscation resistance*: It must be difficult to obtain false negative and false positive results, even after manipulating the input data.

Once the analysis of the characteristics of ssdeep was completed, we were able to identify a list of additional practical features (which are closely related to ssdeep's limitations) that, in our consideration, any bytewise approximate matching function should provide (see Reference [16]):

- *Generation of results consistent with the content of the compared files*: This requirement implies the existence of a simple and clear definition of the concept of similarity, and how to express that concept as a number. In this context, this contribution uses a definition that states that similarity is the degree to which the contents of two files look like or are the same. This definition encompasses the concepts of *resemblance* and *containment* mentioned in Reference [3]. In order to measure it, a value that represents the percentage of the larger file which is also contained in the smaller file is provided.
- *Creation of signatures whose length is not affected by the input file size*: In order to avoid problems when storing the signatures of large files, the length of the signatures should be independent of the input file size.
- *Detection of content swapping*: This concept refers to the situation in which some portion of the document is taken from its original location and positioned in a different part, so from a graphical point of view it could be seen as moving data blocks inside the document. For a file where content swapping is the only operation performed, the result should be close to 100, as the content of the original and the modified files is basically the same.
- *Comparison of files of different sizes without limit*: In some cases, it is necessary to compare files of dissimilar size. If the only material available to the user are the signatures, then the files will be compared only if those signatures have a common block size. Implementations that allow the comparison of a signature file and a content file need to use the block size associated to the signature when processing the data file, which does not always produce the desired results, specially when the value of that block size is too big and does not generate a valid signature for the data file. This situation can happen, for instance, when comparing files five or ten times larger, so implementations need to provide a means for comparing such files.

As the reader may note, the previous requirements can be divided into two groups associated to the procedures of signature generation and signature comparison. For the sake of clarity, we provide below (see Table 1) the two differentiated sets of requirements as they will be referred to in the rest of the document:

Table 1. Requirements that should be fulfilled by any approximate matching application.

Signature Generation Procedure	Signature Comparison Procedure
(G1) Signature compression. (G2) Ease of computation. (G3) Coverage. (G4) Independence regarding the input file size.	(C1) Generation of a similarity score consistent with the content of the compared files. (C2) Ease of computation. (C3) Obfuscation resistance. (C4) Content swapping detection. (C5) Comparison of files of different sizes.

We have established a similarity definition that considers an important problem either not completely taken into account or implemented in an unsatisfactory way by other functions: We believe that the similarity comparison should detect the inclusion of the content of a file in another file, but at the same time it should also indicate in some way which parts of the larger file are contained in the smaller file. This feature would be very useful in many contexts, such as plagiarism detection. For example, if we consider three files extracted from the same book (one with the first chapter, another with the first ten chapters, and the last one with the whole book), the comparison should work in the following way—the result of comparing the first and the second file should be different from the one obtained when comparing the first and the third file. This outcome is possible with our similarity definition, as it provides more granularity about the results.

4.2. SiSe Signature Generation Procedure

As mentioned before, the SiSe signature generation procedure is based on that implemented in ssdeep. For example, the generation of the initial block size in SiSe and in ssdeep are the same (i.e., the minimum length of the leading signature is half the initially expected length), using the same versions of the Adler-32 and FNV (Fowler-Noll-Vo) functions, which were also selected by spamsum because their performance was better compared to other cryptographic functions such as MD5 or the SHA family.

However, SiSe includes important modifications in order to be aligned with the requirements described above. One of the more obvious differences between SiSe and ssdeep is the generation of two characters per trigger point instead of only one, using the last 12 bits of the FNV output instead of the last 6 bits. With this decision, the main drawback is that SiSe signatures require twice the number of characters than ssdeep signatures for the same file and block size. Nevertheless, in order to keep a low percentage of false positives, from our perspective it is essential to increase the number of characters generated with the output of the FNV function.

Even though SiSe does not impose a limit on the maximum length for the signatures when generating them, for performance reasons we have imposed in its implementation an arbitrary limit of 2560 double characters when comparing signatures. That value has been selected so, in most signatures, it is not exceeded, as the expected number of double characters in SiSe (and of single characters in ssdeep) is 64.

Another difference with respect to ssdeep is that SiSe computes three signatures in each execution loop, with block sizes b, $b/2$ and $b/4$ (in Reference [12] there is a similar technique, but Chen and Wang use b, $2b$, $4b$, and $8b$ instead, so our algorithm implements the opposite approach). The reason behind this decision is that the experimental work described in Reference [9] (which employed 15,036 files of several types totalling 3.84 gigabytes) demonstrated that 67.3% of the tested files needed the main loop of the algorithm (i.e., the loop that scans the file at byte level and processes its full content) to be executed once, while 26.2% needed to run the main loop twice, and the rest of the files needed at least three loop executions. In the case of our design, it is possible to choose as leading and secondary signature the ones related to block sizes b and $b/2$, or to block sizes $b/2$, and $b/4$, after any execution of the main loop. With this, it is not necessary to perform any additional loop execution and, consequently, the running time will decrease significantly in some cases. Figure 1 illustrates the difference between ssdeep and SiSe regarding the number of main loop executions for each case.

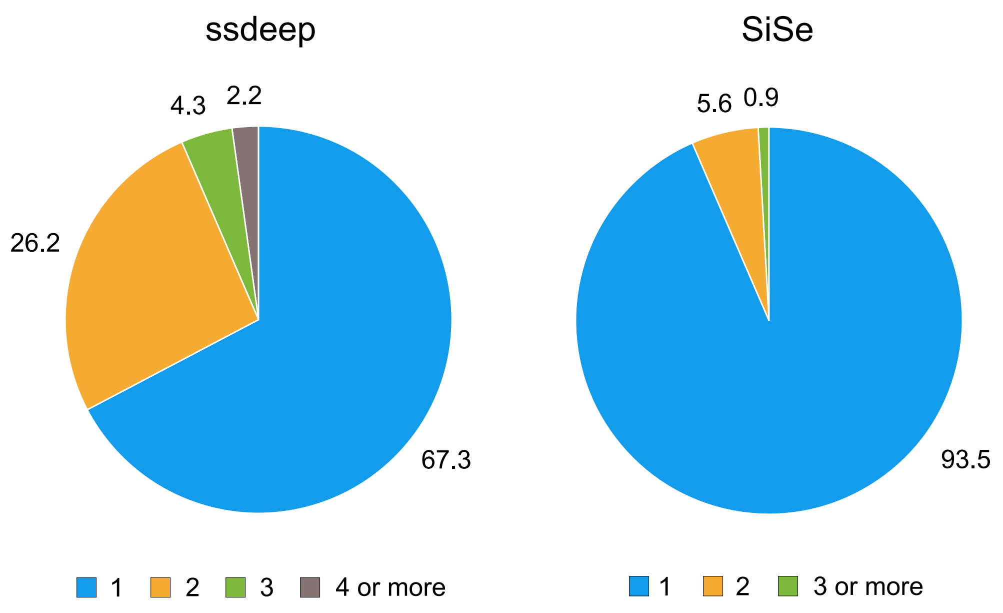

Figure 1. Percentage of files that need a different number of executions of the main loop.

While, in theory, calculating a third signature increments the complexity of the implementation and, in principle, would increment the running time, the difference is negligible in practice, since our algorithm checks if a certain byte is a trigger point when using block size b only if that byte is a trigger point for both $b/4$ and $b/2$. Besides, our approach allows the completion of only one pass most of the times (more precisely, in more than 90% of the cases) in comparison to ssdeep, where two or more passes are necessary in almost one third of the cases.

In our implementation we have decided to use b and $b/2$ (or $b/2$ and $b/4$) instead of b and $2b$ as the relationships between the block sizes of the leading and secondary signatures because, in this way, the length of both signatures surpasses the minimum length (32 double characters). This fact becomes important when using the secondary signature in a comparison: If, for the sake of precision, it is important for the length of the leading signature not to have less than 32 double characters, then the secondary signature should satisfy the same requirement. In our case, any signature selected for a comparison will surpass the minimum length, improving consequently the accuracy of the comparisons. It is obvious that the disadvantage of this design is that, in most files, the length of the secondary signature is larger than the length of the leading signature (the smaller the block size, the higher the number of trigger points that are likely to be detected), incrementing the size of the file containing both signatures.

Algorithm 1 presents the details of SiSe's signature generation procedure [16]. The meaning of the different functions included in the algorithm is as follows:

- `length(element)`: If the argument is a file, the function returns the length in bytes of the file. If it is a string, it returns the number of characters of the string.
- `floor(value)`: It computes the largest integer not greater than `value`.
- `subarray(array,start,end)`: It copies from `array` the subarray whose first byte is located at `start` and whose last byte is located just before the byte indicated by the position `end`.
- `updateWindow(byte)`: It updates the content of the sliding window taking into account the byte passed as the argument of the function.
- `Adler`*`(input)`: It computes the value associated to the byte array `input` using ssdeep's variant of the Adler-32 function.
- `FNV`*`(input)`: It computes the value associated to the byte array `input` using ssdeep's variant of the FNV function.

- `codify(value)`: It generates two Base64 characters (one using the last group of 6 bits of `value`, and the other one using the previous group of 6 bits).
- `append(string,chars)`: It appends two characters to `string`.

Algorithm 1 SiSe signature generation.

```
inputSize ← length(inputFile)
blockSize ← 3
dec ← 1
quotient ← floor(inputSize/64 )
while (blockSize < quotient) do
    blockSize ← 2·blockSize
end while
sig1Len, sig2Len ← 0
blockSize1 ← blockSize
blockSize2 ← blockSize1/2
blockSize4 ← blockSize2/2
while ((sig1Len < 64) and (sig2Len < 64)) do
    index ← 0
    last1, last2, last4 ← -1
    while (index < inputSize) do
        currentByte ← subArray(baInput,index,index+1)
        baWindow ← updateWindow(currentByte)
        resultAdler ← Adler*(baWindow)
        if ((resultAdler % blockSize4) = (blockSize4 - dec)) then
            baBlock4 ← subArray(baInput,last4+1,index+1)
            resultFNV ← FNV*(baBlock4)
            pairChars ← codify(resultFNV)
            sig4 ← append(sig4, pairChars)
            last4 ← index
            if ((resultAdler % blockSize2) = (blockSize2 - dec)) then
                baBlock2 ← subArray(baInput,last2+1,index+1)
                resultFNV ← FNV*(baBlock2)
                pairChars ← codify(resultFNV)
                sig2 ← append(sig2, pairChars)
                last2 ← index
                if ((resultAdler % blockSize1) = (blockSize1 - dec)) then
                    baBlock1 ← subArray(baInput,last1+1,index+1)
                    resultFNV ← FNV*(baBlock1)
                    pairChars ← codify(resultFNV)
                    sig1 ← append(sig1, pairChars)
                    last1 ← index
                end if
            end if
        end if
    end while
    sig1Len ← length(sig1)
    sig2Len ← length(sig2)
    if (sig1Len < 64) and (sig2Len < 64) then
        blockSize1 ← blockSize1/4
        blockSize2 ← blockSize1/2
        blockSize4 ← blockSize2/2
    else
        if (sig1Len < 64) then
            return sig1,sig2
        else
            return sig2,sig4
        end if
    end if
end while
```

4.3. SiSe Signature Comparison

In addition to the changes in the signature generation process explained above and the multi-threading design, in this contribution we have implemented a new signature comparison algorithm which compares two signature strings.

Our work is related to other edit distance algorithms that allow moves (e.g., References [42–45]). However, those contributions are mainly focused on the performance of such algorithms, and in many cases they impose important restrictions during the comparison process. For example, in Reference [42] the strings under comparison must be of equal size and must contain exactly the same characters; Reference [43] makes his study assuming that each letter occurs at most k times in the input strings; in Reference [44] the original position (p_1) and the final position (p_2) of the substring to be moved must satisfy a limiting rule (namely, if l is the length of the substring, it is necessary that p_2 is either smaller than p_1 or larger than $p_1 + l$). Regarding Reference [45], it uses a different technique based on the embedding of strings into a vector space and using a parsing tree.

Compared to the previously mentioned contributions, our algorithm manages text units comprised of two characters, while the rest of related algorithms, to the best of our knowledge, manage single characters, which allows us to improve the detection rate, as the probability of two pairs of characters being the same is $1/4096$ (the two Base64 characters of a pair use 12 bits) instead of $1/64$, as it is the case when using only one Base64 character (and the 6 bits associated to it).

Algorithm 2 shows all the calculations performed to calculate numerically the similarity of the strings `sig1` (with length `sig1Len`) and `sig2` (with length `sig2Len`), where the first step consists in creating a two-dimensional array of size (`sig1Len` + 1) $\times$ (`sig2Len` + 1). That array, called `arrComp`, initially contains the value 0 in all of its positions, but is updated whenever a double character of the first string matches a double character of the second string (only if both double characters are located in an even position, considering that the first position of a string is 0). The arrays identified as `arrx` and `array` mark if a character has already been taken into account during the computation of the score (a value `0` means that the character has not be considered for the score up to that point, whilst a value `1` means that the character belongs to a substring included in both input strings that has already been used during the calculation of the score).

Algorithm 2 SiSe signature comparison.

```
for all $i$ such that $1 \leq i \leq$ sig1Len do
    for all $j$ such that $1 \leq j \leq$ sig2Len do
        if sig1[$i-1$] $=$ sig2[$j-1$] then
            if ($i$%2 $= 1$) and ($j$%2 $= 1$) then
                if sig1[$i$]=sig2[$j$] then
                    arrComp[$i$][$j$] = arrComp[$i-1$][$j-1$] + 1
                end if
            else
                if ($i$%2 $= 0$) and ($j$%2 $= 0$) then
                    if arrComp[$i-1$][$j-1$] > 0 then
                        arrComp[$i$][$j$] = arrComp[$i-1$][$j-1$] + 1
                        if $i$ <sig1Len and $j$ <sig2Len then
                            if sig1[$i$] $\neq$ sig1[$j$] then
                                list.add((arrComp[$i$][$j$],$i$,$j$))
                            end if
                        else
                            list.add((arrComp[$i$][$j$],$i$,$j$))
                        end if
                    end if
                end if
            end if
        end if
    end for
end for
index1 $\leftarrow 0$, index2 $\leftarrow 0$
for all item in list do
    if arrx[item.posRow]$= 0$ and arry[item.posCol]$= 0$ then
        points = points + item.val
        for all  index1 such that (item.posRow - item.val + 1) $\leq$ index1 $\leq$ item.posRow do
            for all  index2 such that (item.posCol - item.val + 1) $\leq$ index2 $\leq$ item.posCol do
                if ((arrx[index1]$= 1$) or (arry[index1]$= 1$)) then
                    points = points - 1
                end if
                arrx[index1] = 1
                arry[index2] = 1
            end for
        end for
    else
        dec = item.val;
        for all index1 such that item.posRow $\leq$ index1 $\geq$ item.posRow - item.val$+1$ do
            for all index2 such that item.posCol $\leq$ index2 $\geq$ item.posCol - item.val$+1$ do
                if arrx[index1]$= 1$ or arry[index2]$= 1$ then
                    dec $\leftarrow$ dec - 1
                else
                    break
                end if
            end for
        end for
        if reduced>0 then
            points = points + dec
            for all index1 such that (item.posRow-item.val$+1$) $\leq$ index1 $\leq$ (item.posRow-removed) do
                for all index2 such that (item.posCol-item.val$+1$) $\leq$ index2 $\leftarrow$ (item.posCol-(item.val-dec)) do
                    if arrx[index1]$= 1$ or arry[index2]$= 1$ then
                        points $\leftarrow$ points - 1
                    end if
                    arrx[index1] $\leftarrow 1$
                    arry[index2] $\leftarrow 1$
                end for
            end for
        end if
    end if
end for
return  (points*100)/max(sig1,sig2)
```

Table 2 shows an example of such an array when comparing the strings `A1B2C3D4F8` and `1A1BC3D4F7A1`. As it can be observed, the substring `1B` is not credited any point because in one string it starts at an odd position while in the other string it starts in an even position, so they do not produce a match.

Table 2. Comparison array example.

		A	**1**	**B**	**2**	**C**	**3**	**D**	**4**	**F**	**8**
		0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	2	0	0	0	0
D	0	0	0	0	0	0	0	3	0	0	0
4	0	0	0	0	0	0	0	0	**4**	0	0
F	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0
A	0	1	0	0	0	0	0	0	0	0	0
1	0	0	**2**	0	0	0	0	0	0	0	0

Once all the characters are compared and an array such as the one shown in Table 2 is completed, a list is created with elements that include three values: The highest value assigned to a series (displayed in bold in the table), and the row and column associated to that value. In the example, two elements would be added to the list, $(4, 8, 8)$ and $(2, 12, 2)$. Those elements, representing all the sequences of double characters (located at valid positions) are then processed by the second part of the algorithm in order to increase the number of points assigned to the comparison and remove those substrings from the comparison, so they are not taken into account more than once.

In the example shown, the final score would be 50, as 50% of string `1A1BC3D4F7A1` can be found in string `A1B2C3D4F8` (more specifically, the double characters `A1`, `C3`, and `D4` are shared by both strings, while the two strings contain three additional double characters that are not common).

5. SiSe Implementation

5.1. Interface

We have implemented our design of SiSe as a C++ multi-threaded command-line application that uses the features of the C++17 standard [46] and the OpenMP library [47]. The application implements a dual input command format: One adapted to the FRASH tests, and a proprietary one. Regarding the input format compatible with FRASH, SiSe can be used without modifiers with one or several files separated by spaces. This indicates SiSe that it must treat those files as content data and generate their signatures individually. In turn, if modifier `-x` is used along with the name of a file, it indicates the application that the file contains multiple signatures and that it must compare all signatures against the rest. Finally, if modifier `-r` is employed together with the path of a directory, SiSe will compute the signature of all the files contained in that folder.

In comparison, the proprietary input command format used by SiSe for creating a new signature is `sise -i input_file`, which allows the following additional optional elements:

`-o`: Output file name.
`-b`: Block size (if not indicated, the block size is computed by the application as in ssdeep).
`-d`: Decremental value, that is, the amount to be decremented to the block size in the trigger point computations (the default value is 1).
`-t`: Number of threads used by the application (if not indicated, the value employed is decided by SiSe based on the size of the input file).

Independently of the input command format used, the signature procedure creates a text string containing a header and two signatures (the leading and the secondary signatures). The header of the initial version is `SiSe-1.0--7:1--blocksize:hash,filename`, where the elements `7` and `1` are replaced by other numbers when non-default values selected by the user are employed for the window size and the decremental value, respectively.

The first signature that follows the header is the leading signature. Both signatures start with the block size and the value used in the modulus comparison to identify the trigger points (i.e., the block size minus the decremental value), the last one located between parenthesis and followed by a colon. The next item that appears is the actual signature string, corresponding to the previously stated block size. A double colon `::` separates both signatures. Finally, the complete path of the file that has been processed (written between quotation marks) appears separated from the previous information by a comma.

In order to illustrate the signature generation process, Table 3 shows the output provided by ssdeep and SiSe when processing the plain text file containing Robert Louis Stevenson's Treasure Island [48]. As it can be noted, the character that appears in the second position of each pair in the secondary signature of SiSe (the one with a block size of 3072) matches the corresponding character of the leading signature of ssdeep. This fact happens as the block size, windows size, and decremental value is the same for those two signatures. The coincidence in the series of characters ends at the 63th character of the ssdeep signature, as at that point ssdeep computes its final character using the rest of the document, independently of the length of that remaining part and the number of trigger points included in it. In contrast, SiSe continues computing all the characters derived from the presence of additional trigger points.

Table 3. ssdeep and SiSe signature examples.

```
> ssdeep pg27780.txt

ssdeep,1.1--blocksize:hash:hash,filename
3072:7ElbU+TnglXXzpfu2/8Fbo0ANtD3xKdeS/GVcO/bLe/sVcdJqZ/7y0O3FFh1g+Mp:27YpKwhKdeoD/JVjQT1J
+rOSfitp4pc7,"/media/victor/USB/pg27780.txt"
```

```
> sise pg27780.txt

SiSe-1.0--7:1--blocksize(modvalue):hash,filename
6144(6143):k2C7lYMp6KJw/hiKodceZoVDD//JfVrjOQVTT1GJA+Ir8O+SrfGiEtMpq4UpucU7::3072(3071):Q7
BEUlUbaUe+1TInKgvluXnXkzMpDfLu12J/j8ZFfbwou0YAxNXtXDU3NxiKodceLSf/gGxVCcvOi/PbSLDeD/EsZVuc
zd+Jyq/ZC/M7fyH0aOG3JFaF1hc1Kgh+VMZVbrXJR/HrT1GJA+xYR6cvz7a58OIfvpEBjAVAA+x3TuOB+gTTWOEtn3
pfQ2wLkqx9lFhovSu4uc1ePk,"/media/victor/USB/pg27780.txt"
```

The proprietary format of the command for comparing signatures and/or files is `sise -c input_file_1 input_file_2`, which allows these combinations:

- *Two signature files*: The comparison between signatures can be done only if they have at least one block size value in common. Additionally, the decremental value and the window size must be equal in both signature files. If the leading signatures use the same block size value, and the secondary signatures also employ the same block size value (but different from the block size of the leading signatures) at the same time, SiSe is able to apply the edit distance algorithm (see Algorithm 2 in Section 4.3) in both cases, and returns the highest score calculated.

- *A signature file and a data file*: The block size, decremental value, and window size are obtained from the signature file and used in the processing of the signature related to the data file.
- *Two data files*: In a first step, the leading and secondary signatures of both files are independently computed using the default parameters established for the block size, the decremental value, and the window size. Then, SiSe identifies the smallest block size value of the two leading signatures and processes again the signature of the other file with the new block size value.

5.2. Multi-Thread Design

The C++ implementation of SiSe uses the OpenMP library [47] both during the signature creation and the signature comparison procedures. Multi-threading is employed in the comparison phase when comparing digests using the `-x` option, that is, when requesting SiSe to compare all the signatures included in a file with the rest of signatures of that file. In that scenario, SiSe distributes the work between a number of threads that equals the number of logical cores of the processor, where the number of logical cores is the number of physical cores multiplied by the number of threads that can run on each core through the use of hyperthreading.

Multi-threading is also used during the signature generation process in two different ways. For any file larger than 1 megabyte, the strategy consists in dividing the file into several parts so each thread is assigned a portion of the file. Given the nature of the operations that each thread must perform, the multi-thread design tries to minimize the amount of bytes that are processed twice. Nevertheless, the double processing of some portions of the file cannot be avoided, as it will become clear with the description included below and Figure 2. Besides, when a command requests to process more than one file, SiSe creates a list of files whose size is less than 1 megabyte and assigns one thread per file up to the thread limit.

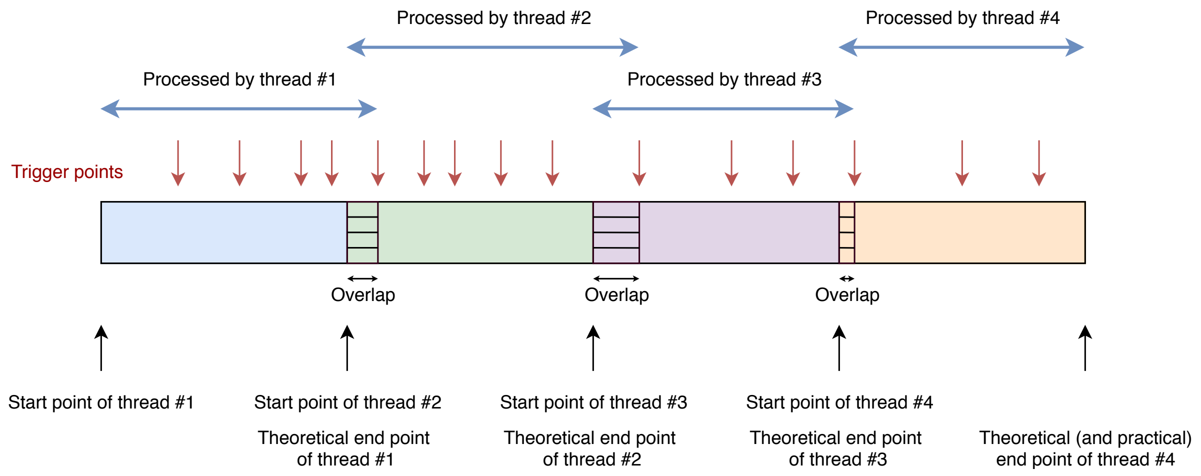

Figure 2. Representation of the multi-thread design.

Coming back to the multi-threading scheme for a file larger than 1 megabyte, when using n threads that file is divided into n parts, so each thread starts processing bytes at a different offset inside the file. If the working thread is thread #1, the first triggered point found by the thread provokes the computation of the first pair of characters of the signature. However, if the working thread is not thread #1, when the thread detects the first trigger point it adds no pair of characters to the signature, as at that point the thread lacks the knowledge of where the previous trigger point was located. From that moment on, all the threads work as expected and, whenever they find a trigger point, they compute the corresponding two-character element of the signature. Threads continue to work in that way and, except the last thread (which finishes its execution when it reaches the end of the file, computing at that moment the pair of characters associated to the end of the signature), they only stop when finding the first trigger point located after the byte that marks the theoretical end point of their respective file partitions.

This scheme brings as a consequence a certain overlapping of parts of the file processed both by threads $i-1$ and i, where the overlapped portion is read by thread $i-1$ at the end of its execution and by thread i at the beginning of its execution. The extent of the overlap depends on the content of the file, its size, and the number of threads. Figure 2 provides an example that illustrates the multi-thread scheme when four threads are used.

As the detection of trigger points depends on the value of the byte being processed and the value of the previous 7 bytes, before processing the first byte each thread fills in the sliding window array with the proper 7 bytes, so no false trigger points appear and thus the signatures are guaranteed to be the same independently of the number of threads used during the process.

Regarding the optimal number of threads to be employed by the application, after several tests it was decided to use one thread with files whose size is less than 1 megabyte and a number of threads that equal the number of logical cores in the rest of cases. The reason for doing this is that, for small files, the cost of setting up the threads and aggregating the results provided by them does not compensate the benefit of a reduced workload per thread. As for the buffer size used when reading the files, during the tests a buffer of one megabyte proved to be the best option.

6. Ad-Hoc Tests

This section shows the results of the comparisons performed with SiSe, ssdeep, sdhash, and LZJD in different scenarios designed by ourselves to verify the features associated to the characteristics described in Section 4.1.

We have classified the tests in four categories: Similarity tests, dissimilarity tests, special signatures tests, and suitability tests. For the first two groups, we have tested plain text files, Word documents, and BMP images. The tests with special signatures are intended to check the behaviour of the four algorithms in some special cases. The suitability tests try to determine the practicability of the applications in real-world scenarios by measuring the performance and signature size using files whose length span from less than 1 megabyte to several gigabytes.

As mentioned in Section 4, after performing several tests modifying the values of the block sizes, window sizes, and decremental values, we concluded that the results did not show appreciable differences. For this reason, we decided to use the default parameters (sliding window of 7 bytes, block size computed by the algorithm, and decremental value equal to 1) in all the tests that follow. Moreover, as they are the same values that ssdeep use, it is easier to derive conclusions from the results.

It is important to notice that, in all the tables related to tests in which two files are employed, the identifier linked to the row represents the first input argument and the identifier linked to the column represents the second input argument.

Tables associated to SiSe include four values in each cell. Those values are obtained when comparing two hash files, a content file and a hash file (in that order), a hash file and a content file (in that order), and two content files, respectively.

The values associated with the tests with ssdeep can be obtained either by comparing two signature files by means of the command `ssdeep -ak sigfile1 sigfile2` or by comparing a signature file and a content file (in that order) using the command `ssdeep -am sigfile contentfile`. As in all the tests both methods provide the same numerical value, in the tables related to ssdeep we have included each value once, even though all the tests have been performed separately with both commands.

Results can be obtained with sdhash and LZJD through two procedures. In the first one, a signature file with the hashes of all the files has been generated with the command `sdhash * > sigfile.sdbf` and, as a second step, that file has been used as input for the command `sdhash -c sigfile.sdbf` that compares all the hashes included in the file. In the second case, the results have been directly generated by using the command `sdhash -g *`. When using LZJD, the command `sdhash` must be replaced with the command `LZJD.jar`, as the author of LZJD recommends using the Java version of its algorithm [49]. In the tables with the sdhash results only one value appears in each cell, as the result obtained with the two procedures is exactly the same. In comparison, LZJD provides different values when using the two procedures with the same files, so in that case we have preferred to include both values in the corresponding table.

It is interesting to note that, while the ssdeep matrices are always symmetric, that is not always the case with the matrices related to SiSe. The reason for that fact is that the second score of each cell represents a test where a signature file (pertaining to the file identified by the row of the table) is compared against the content of a data file (whose identifier is at the top of the column that contains the cell with the score). According to its predefined behaviour, SiSe imposes the leading block size of the file associated to the row when processing the signature of the file associated to the column. If the leading block size of the file associated to the row is smaller than the theoretical leading block size of the file associated to the column, a signature (which theoretically is longer than the original signature that would have been generated with SiSe) is produced by the application. On the other hand, when the leading block size of the file associated to the row is larger than the theoretical leading block size of the file associated to the column, it could happen that, when that block size is forced on the file associated to the column, no signature of the minimum length could be generated. In this case, SiSe is not able to perform the comparison. In the rest of the cases there might be small differences in the outcome produced by SiSe for the second test (i.e., the second number displayed in every cell) in both comparisons (i.e., file A compared to file B, and file B compared to file A), depending on the leading block size used in each case. The same situation appears in the tests associated to the third score. However, when using two signature files or two data files (i.e., the first and fourth scenarios) the results are always symmetric for each comparison.

As a final comment, for better legibility in the tables, we have discarded the results related to the comparison of any file with itself.

6.1. Similarity Tests

6.1.1. Plain Text Documents

In this test, plain text files with the first 20 chapters of Miguel de Cervantes' Don Quijote, in the version offered by The Project Gutenberg [50], have been used. These files are the following: `Q01.txt` (10,887 bytes, Quijote's chapter 1), `Q02.txt` (23,877 bytes, chapters 1, 2), `Q03.txt` (37,342 bytes, chapters 1–3), `Q04.txt` (51,374 bytes, chapters 1–4), `Q05.txt` (60,526 bytes, chapters 1–5), `Q10.txt` (125,217 bytes, chapters 1–10), `Q15.txt` (204,198 bytes, chapters 1–15), and `Q20.txt` (305,527 bytes, chapters 1–20). Table 4 shows the percentage of the smaller file as a part of the larger file using the byte size as the comparison value, as in this test the smaller files are totally contained in the larger ones.

Table 4. Percentage of the larger file representing the smaller file with content in plain text format.

	Q01	**Q02**	**Q03**	**Q04**	**Q05**	**Q10**	**Q15**	**Q20**
Q01	-	45.6%	29.2%	21.2%	18.0%	8.7%	5.3%	3.6%
Q02	45.6%	-	63.9%	46.5%	39.4%	19.1%	11.7%	7.8%
Q03	29.2%	63.9%	-	72.7%	61.7%	29.8%	18.3%	12.2%
Q04	21.2%	46.5%	72.7%	-	84.9%	41.0%	25.2%	16.8%
Q05	18.0%	39.4%	61.7%	84.9%	-	48.3%	29.6%	19.8%
Q10	8.7%	19.1%	29.8%	41.0%	48.3%	-	61.3%	41.0%
Q15	5.3%	11.7%	18.3%	25.2%	26.6%	61.3%	-	66.8%
Q20	3.6%	7.8%	12.2%	16.8%	19.8%	41.0%	66.8%	-

Tables 5–9 present the outcomes of the tests using SiSe, ssdeep, sdhash, and LZJD. As it can be seen, SiSe is the application that offers results closer to the percentages included in Table 4, providing values better adapted to the similarity definition presented in Section 4.1. In addition to that, it is important to note that SiSe provides results (with two of its four operation modes) in several cases where ssdeep and LZJD return a value of 0, as for example in the comparison between `Q01.txt` and `Q04.txt`, the comparison between `Q02.txt` and `Q10.txt` or the comparison between `Q05.txt` and `Q15.txt`. The values generated by sdhash in those cases are 95, 99, and 100, respectively, which inform that the smaller file is definitely included in the larger file, but do not provide details about how much content of the larger file is replicated in the smaller file.

Table 5. Test results for similar plain text files with SiSe, part 1 (hash vs. hash/file vs. hash/hash vs. file/file vs. file).

	Q01	**Q02**	**Q03**	**Q04**
Q01	-	40/42/43/43	0/26/28/28	0/20/21/21
Q02	40/43/42/43	-	61/61/63/63	46/46/47/47
Q03	0/28/26/28	61/63/61/63	-	74/74/74/74
Q04	0/21/20/21	46/47/46/47	74/74/74/74	-
Q05	0/18/17/18	40/40/40/40	64/64/64/64	85/85/86/86
Q10	0/8/0/8	0/22/20/22	32/35/32/35	44/46/44/47
Q15	0/5/0/5	0/14/0/14	0/23/0/23	0/30/0/31
Q20	0/4/0/4	0/9/0/9	0/15/0/15	0/20/0/20

Table 6. Test results for similar plain text files with SiSe, part 2 (hash vs. hash/file vs. hash/hash vs. file/file vs. file).

	Q05	**Q10**	**Q15**	**Q20**
Q01	0/17/18/18	0/0/8/8	0/0/5/5	0/0/4/4
Q02	40/40/40/40	0/20/22/22	0/0/14/14	0/0/9/9
Q03	64/64/64/64	32/32/35/35	0/0/23/23	0/0/15/15
Q04	85/86/85/86	44/44/46/47	0/0/30/31	0/0/20/20
Q05	-	54/54/54/54	0/0/36/36	0/0/24/24
Q10	54/54/54/54	-	56/56/66/66	0/0/44/44
Q15	0/36/0/36	56/66/56/66	-	58/58/62/62
Q20	0/24/0/24	0/44/0/44	58/62/58/62	-

Table 7. Test results for similar plain text files with ssdeep.

	Q01	**Q02**	**Q03**	**Q04**	**Q05**	**Q10**	**Q15**	**Q20**
Q01	-	66	0	0	0	0	0	0
Q02	66	-	75	66	66	0	0	0
Q03	0	75	-	90	90	47	0	0
Q04	0	66	90	-	99	55	0	0
Q05	0	66	90	99	-	65	0	0
Q10	0	0	47	55	65	-	60	41
Q15	0	0	0	0	0	60	-	77
Q20	0	0	0	0	0	41	77	-

Table 8. Test results for similar plain text files with sdhash.

	Q01	**Q02**	**Q03**	**Q04**	**Q05**	**Q10**	**Q15**	**Q20**
Q01	-	95	95	95	95	95	95	95
Q02	95	-	99	99	99	99	99	99
Q03	95	99	-	100	100	100	100	100
Q04	95	99	100	-	100	100	100	100
Q05	95	99	100	100	-	100	100	100
Q10	95	99	100	100	100	-	100	100
Q15	95	99	100	100	100	100	-	100
Q20	95	99	100	100	100	100	100	-

Table 9. Test results for similar plain text files with LZJD (signature comparison/file comparison).

	Q01	**Q02**	**Q03**	**Q04**	**Q05**	**Q10**	**Q15**	**Q20**
Q01	-	25/35	20/22	0/0	0/0	0/0	0/0	0/0
Q02	25/35	-	59/54	45/38	37/31	0/0	0/0	0/0
Q03	20/22	59/54	-	66/63	53/50	20/21	0/0	0/0
Q04	0/0	45/38	66/63	-	77/77	25/28	0/0	0/0
Q05	0/0	37/31	53/50	77/77	-	34/34	0/0	0/0
Q10	0/0	0/0	20/21	25/25	34/34	-	46/49	29/29
Q15	0/0	0/0	0/0	0/0	0/0	46/49	-	53/53
Q20	0/0	0/0	0/0	0/0	0/0	29/29	53/53	-

There are two situations in which SiSe does not provide a comparison result. The first one happens when comparing two files with incompatible signatures. The second situation appears when comparing a small content file with the hash of a large file, as in that case the block size used in the comparison is too large for generating a valid signature in the smaller file. When the two other situations appear (comparing two content files or a large content file with the hash of a small file), valid results are always produced.

6.1.2. Word Documents

In this second test, the same book as in the previous test was used, although the chapters of Don Quijote have been saved as Microsoft Word 2016 using the identifiers `Q01.docx` (17,637 bytes), `Q02.docx` (24,140 bytes), `Q03.docx` (30,596 bytes), `Q04.docx` (37,422 bytes), `Q05.docx` (41,895 bytes), `Q10.docx` (68,691 bytes), `Q15.docx` (105,228 bytes), and `Q20.docx` (150,969 bytes).

Table 10 shows the percentage of the smaller file as a part of the larger file using the byte size as the comparison value, as in this test the smaller files are totally contained in the larger ones. The values included in this table are slightly higher than the values presented in Table 4 due to the internal format used by Microsoft Word.

Table 10. Percentage of the larger file representing the smaller file with content in Microsoft Word format.

	Q01	**Q02**	**Q03**	**Q04**	**Q05**	**Q10**	**Q15**	**Q20**
Q01	-	73.1%	57.6%	47.1%	42.1%	25.7%	16.8%	11.7%
Q02	73.1%	-	78.9%	64.5%	57.6%	35.1%	23.0%	16.0%
Q03	57.6%	78.9%	-	81.8%	73.0%	44.5%	29.1%	20.3%
Q04	47.1%	64.5%	81.8%	-	89.3%	54.5%	35.6%	24.8%
Q05	42.1%	57.6%	73.0%	89.3%	-	61.0%	39.8%	27.8%
Q10	25.7%	35.1%	44.5%	54.5%	61.0%	-	65.3%	45.5%
Q15	16.8%	23.0%	29.1%	35.6%	39.8%	65.3%	-	69.7%
Q20	11.7%	16.0%	20.3%	24.8%	27.8%	45.5%	69.7%	-

Tables 11–15 present the results obtained with SiSe, ssdeep, sdhash, and LZJD, respectively. With the information included in those tables, it is obvious that SiSe can compare more files than ssdeep. The values provided by SiSe in this test are lower than the values generated in the previous test, which is once again due to the internal structure of Word documents and the metadata that is contained in each file.

In the case of sdhash, it can be said that the values shown in Table 14 do not reflect the trend that can be visualised with SiSe where, for a specific file (e.g., `Q01.docx`) the comparison score decreases as the other file used in the comparison contains a larger portion of the book (`Q02.docx`, `Q05.docx`, `Q10.docx`, etc.), as the content of the initial file is diluted in the larger files.

Finally, it is worth mentioning that LZJD fails to produce results above zero in many cases.

Table 11. Test results for similar Word documents with SiSe, part 1 (hash vs. hash/file vs. hash/hash vs. file/file vs. file).

	Q01	**Q02**	**Q03**	**Q04**
Q01	-	32/32/32/32	29/29/30/30	23/23/23/23
Q02	32/32/32/32	-	32/32/32/32	24/24/24/24
Q03	29/30/29/30	32/32/32/32	-	52/53/52/53
Q04	23/23/23/23	24/24/24/24	52/52/53/53	-
Q05	21/21/21/21	22/22/22/22	47/47/48/48	49/49/49/49
Q10	0/14/0/14	0/15/12/15	11/15/11/15	11/15/11/15
Q15	0/9/0/9	0/10/8/10	6/9/6/10	6/10/6/10
Q20	0/7/0/7	0/8/0/8	0/7/0/7	0/7/0/7

Table 12. Test results for similar Word documents with SiSe, part 2 (hash vs. hash/file vs. hash/hash vs. file/file vs. file).

	Q05	**Q10**	**Q15**	**Q20**
Q01	21/21/21/21	0/0/14/14	00/9/9	0/0/7/7
Q02	22/22/22/22	0/12/15/15	0/8/10/10	0/0/8/8
Q03	47/48/47/48	11/11/15/15	6/6/9/10	0/0/7/7
Q04	49/49/49/49	11/11/15/15	6/6/10/10	0/0/7/7
Q05	-	11/11/16/16	6/6/9/9	0/0/7/7
Q10	11/16/11/16	-	45/45/45/45	27/27/32/32
Q15	6/9/6/9	45/45/45/45	-	65/65/65/65
Q20	0/7/0/7	27/32/27/32	65/65/65/65	-

Table 13. Test results for similar Word documents with ssdeep.

	Q01	**Q02**	**Q03**	**Q04**	**Q05**	**Q10**	**Q15**	**Q20**
Q01	-	40	0	0	0	0	0	0
Q02	40	-	0	0	0	0	0	0
Q03	0	0	-	63	49	0	0	0
Q04	0	0	63	-	50	0	0	0
Q05	0	0	49	50	-	0	0	0
Q10	0	0	0	0	0	-	52	32
Q15	0	0	0	0	0	52	-	68
Q20	0	0	0	0	0	32	68	-

Table 14. Test results for similar Word documents with sdhash.

	Q01	**Q02**	**Q03**	**Q04**	**Q05**	**Q10**	**Q15**	**Q20**
Q01	-	44	36	34	31	35	33	32
Q02	44	-	38	37	22	36	37	36
Q03	36	38	-	71	66	18	21	19
Q04	34	37	71	-	51	17	15	16
Q05	31	22	66	51	-	13	13	13
Q10	35	36	18	17	13	-	77	76
Q15	33	37	21	15	13	77	-	93
Q20	32	36	19	16	13	76	93	-

Table 15. Test results for similar Word documents with LZJD (signature comparison/file comparison).

	Q01	**Q02**	**Q03**	**Q04**	**Q05**	**Q10**	**Q15**	**Q20**
Q01	-	0/0	0/0	0/0	0/0	0/0	0/0	0/0
Q02	0/0	-	0/0	0/0	0/0	0/0	0/0	0/0
Q03	0/0	0/0	-	48/43	32/35	0/0	0/0	0/0
Q04	0/0	0/0	48/43	-	34/37	0/0	0/0	0/0
Q05	0/0	0/0	32/35	34/37	-	0/0	0/0	0/0
Q10	0/0	0/0	0/0	0/0	0/0	-	41/33	26/25
Q15	0/0	0/0	0/0	0/0	0/0	41/33	-	45/48
Q20	0/0	0/0	0/0	0/0	0/0	26/25	45/48	-

6.1.3. BMP Images

This test uses the three images shown in Figure 3. The first image is the Lenna's classic greyscale test image [51]. For the second and third images, although the image is the same, we have rearranged half of it horizontally and vertically, respectively. The three images are bitmaps of the same resolution, therefore their size is the same (786,486 bytes). Readers should note that, in those individual BMP images, each pixel is represented by 24 bits, and that the pixel content is stored sequentially in the file. Pixels are read processing rows from top to bottom and, for any given row, from left to right (which means that the first pixel stored in the file is the one located at the upper left corner of the image, and the last one is the pixel located at the bottom right corner of the file).

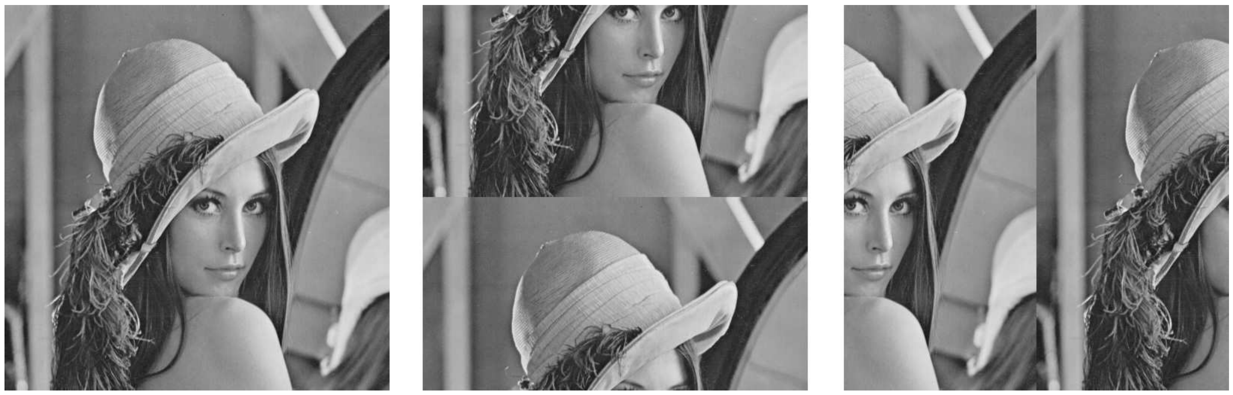

Figure 3. BMP test images.

The results obtained with the four applications are displayed in Tables 16–19, where we have used the identifiers BMP 1, BMP 2, and BMP 3 for the three images.

Table 16. Test results for similar BMP images with SiSe (hash vs. hash/file vs. hash/hash vs. file/file vs. file).

	BMP 1	**BMP 2**	**BMP 3**
BMP 1	100/100/100/100	98/98/98/98	20/20/20/20
BMP 2	98/98/98/98	100/100/100/100	20/20/20/20
BMP 3	20/20/20/20	20/20/20/20	100/100/100/100/

Table 17. Test results for similar BMP images with ssdeep.

	BMP 1	**BMP 2**	**BMP 3**
BMP 1	100/100	52/52	0/0
BMP 2	52/52	100/100	0/0
BMP 3	0/0	0/0	100/100

Table 18. Test results for similar BMP images with sdhash.

	BMP 1	**BMP 2**	**BMP 3**
BMP 1	-	69/69	75/75
BMP 2	69/69	-	50/50
BMP 3	75/75	50/50	-

Table 19. Test results for similar BMP images with LZJD (signature comparison/file comparison).

	BMP 1	**BMP 2**	**BMP 3**
BMP 1	-	37/44	43/50
BMP 2	37/44	-	36/43
BMP 3	43/50	36/43	-

Using the results obtained, we can conclude that ssdeep is not able to match the image vertically rearranged with the two other images. In contrast, SiSe obtains results in both cases. Additionally, SiSe assigns a larger similarity percentage when comparing the first and second images (98%) than ssdeep (52%) which, in our opinion, is closer to reality given our similarity definition and the content of the files. On the other hand, the results obtained when processing the third image with sdhash are better adapted to our similarity definition, but sdhash fails to identify the first two images as having basically the same content. Regarding LZJD, its results follow the trend of sdhash but with scores significantly lower that those of sdhash. The reason for SiSe providing a lower result when comparing the third file to the other ones than when comparing the second file to the third one, is that due to the way a BMP file stores its pixels, a lot more trigger points are shared by the first and second images.

6.2. Dissimilarity Tests

The aim of this group of tests is to detect undesirable false positives by comparing files of the same format but with different content. Given the results obtained, no file is interpreted as strongly related to the other files included in the same test, as the maximum value obtained through those tests is 10 (produced by sdhash).

6.2.1. Plain Text Documents

The plan text files used in this test contain the following books as offered by The Project Gutenberg: H. G. Wells' The Time Machine [52] (201,875 bytes), Miguel de Cervantes' Don Quijote [50] (2,198,927 bytes), Robert Louis Stevenson's Treasure Island [48] (397,415 bytes), and Jules Verne's Voyage au Centre de la Terre [53] (460,559 bytes).

In this test ssdeep and LZJD did not provide any positive results when comparing the different files. The results obtained with SiSe and sdhash are offered in Tables 20 and 21.

Table 20. Test results for dissimilar plain text files with SiSe (hash vs. hash/file vs. hash/hash vs. file/file vs. file).

	Machine	**Quijote**	**Treasure**	**Voyage**
Machine	-	0/0/2/2	5/5/5/5	1/1/4/4
Quijote	0/2/0/2	-	0/3/0/3	0/3/0/3
Treasure	5/5/5/5	0/0/3/3	-	3/3/3/3
Voyage	1/4/1/4	0/0/3/3	3/3/3/3	-

Table 21. Test results for dissimilar plain text files with sdhash.

	Machine	**Quijote**	**Treasure**	**Voyage**
Machine	-	10/10	9/9	7/7
Quijote	10/10	-	5/5	5/5
Treasure	9/9	5/5	-	5/5
Voyage	7/7	5/5	5/5	-

SiSe obtains a maximum value for dissimilar files of 5, which implies that in order to discard false positives a threshold could be established. In the case of ssdeep, it returns a value of 0 for every comparison without processing its edit distance algorithm. The reason for that is that the corresponding signatures do not have a substring with at least 7 characters common, and hence it directly returns a value of 0. The comparisons performed by sdhash show values that are slightly higher than the ones provided by SiSe, but without a significant difference.

6.2.2. Word Documents

For this test, new Microsoft Word 2016 (Microsoft Corporation, Redmond, WA, USA) files have been created with the same content of the books used in the previous test. The size in bytes of each file is 118,169 (`Machine.docx`), 1,918,754 (`Quijote.docx`), 372,941 (`Treasure.docx`), and 263,316 (`Voyage.docx`).

While ssdeep and LZJD did not provide any positive results when comparing the different files, SiSe and sdhash returned the residual values included in Tables 22 and 23.

Table 22. Test results for dissimilar Word files with SiSe (hash vs. hash/file vs. hash/hash vs. file/file vs. file).

	Machine	**Quijote**	**Treasure**	**Voyage**
Machine	-	0/0/1/1	2/2/2/2	1/1/2/2
Quijote	0/1/0/1	-	0/2/0/2	0/2/0/2
Treasure	2/2/2/2	0/0/2/2	-	1/1/1/1
Voyage	1/2/1/2	0/0/2/2	1/1/1/1	-

Table 23. Test results for dissimilar Word files with sdhash.

	Machine	**Quijote**	**Treasure**	**Voyage**
Machine	-	2/2	2/2	5/5
Quijote	2/2	-	3/3	2/2
Treasure	2/2	3/3	-	1/1
Voyage	5/5	2/2	1/1	-

As in the case of dissimilar plaint text files, the values returned by both SiSe and sdhash are very low and could be avoided by establishing a small threshold.

6.2.3. BMP Images

In this test, three well-known test colour images obtained from Reference [54] have been used. The first one is Lenna's portrait (`lenna.bmp`), the second one is the photograph of a combat jet (`airplane.bmp`), and the third one displays several vegetables (`pepper.bmp`) as it can be seen in Figure 4. The resolution of the three files is 512 $\times$ 512 pixels, and they use 24 bits per pixel, so their size is 786,486 bytes.

Figure 4. Color BMP test images.

While ssdeep and LZJD did not provide any positive results, sdhash returned a value of 1 when comparing `pepper.bmp` to the other two files and SiSe also returned a value of 1, but only when comparing `lenna.bmp` and `pepper.bmp`. Thus, the results are aligned with the expected behaviour, as the content of those files is clearly different.

6.3. Special Signatures

The aim of this test is to verify the behaviour of SiSe when it has to process some special signatures which represent cases that cannot be directly translated to regular input data files. Even so, we believe that it would be worthwhile to test SiSe and ssdeep in this scenario, as it represents different degrees of content modification. This test was not carried out with sdhash and LZJD as it was impossible to verify if the forged signatures truly reflected the nature of the tests, since their signature generation process is completely different.

Once again, it is important to point out that these special signatures do not represent actual files, instead they have been developed ad-hoc taking into account that the minimum length had to be 32 characters so they could be used with ssdeep. The special signatures for ssdeep are as follows:

```
S01: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
S02: ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ
S03: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
S04: 12345678901234567890123456ABCDEFGHIJKLMNOPQRSTUVWXYZ
S05: BADCFEHGJILKNMPORQTSVUXWZYbadcfehgjilknmporqtsvuxwzy
S06: CDABGHEFKLIJOPMNSTQRWXUVabYZefcdijghmnklqropuvstyzwx
S07: EFGHABCDMNOPIJKLUVWXQRSTcdefYZabklmnghijstuvopqrwxyz
S08: IJKLMNOPABCDEFGHYZabcdefQRSTUVWXopqrstuvghijklmnwxyz
S09: QRSTUVWXYZabcdefABCDEFGHIJKLMNOPwxyzghijklmnopqrstuv
S10: ghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZabcdef
```

The base element for this set is the first signature, `S01`. The second signature, `S02`, duplicates the first half of `S01` in the second half. Signature `S03` swaps the two blocks of `S01`. Additionally to this change, signature `S04` replaces the first half of `S03` with a string of digits. The remaining signatures, from `S05` to

S10, have been created taking S01 and then applying transpositions of blocks whose sizes in characters are is 1, 2, 4, 8, 16, and 32, respectively.

Bearing in mind that SiSe signatures use two characters per trigger point, the special signatures of ssdeep have been adapted accordingly. In that sense, each character have been doubled (e.g., A and a have been transformed into AA and aa, respectively) in the signatures employed with SiSe.

The results obtained when comparing these special signatures are included in Tables 24 and 25.

Table 24. Test results with ad-hoc signatures and SiSe.

	S01	**S02**	**S03**	**S04**	**S05**	**S06**	**S07**	**S08**	**S09**	**S10**
S01	100	50	100	50	100	100	100	100	100	100
S02	50	100	50	50	50	50	50	50	50	50
S03	100	50	100	50	100	100	100	100	100	100
S04	50	50	50	100	50	50	50	50	50	50
S05	100	50	100	50	100	100	100	100	100	100
S06	100	50	100	50	100	100	100	100	100	100
S07	100	50	100	50	100	100	100	100	100	100
S08	100	50	100	50	100	100	100	100	100	100
S09	100	50	100	50	100	100	100	100	100	100
S10	100	50	100	50	100	100	100	100	100	100

Table 25. Test results with ad-hoc signatures and ssdeep.

	S01	**S02**	**S03**	**S04**	**S05**	**S06**	**S07**	**S08**	**S09**	**S10**
S01	100	50	50	50	0	0	0	55	63	63
S02	50	100	50	50	0	0	0	47	50	50
S03	50	50	100	50	0	0	0	36	44	90
S04	50	50	50	100	0	0	0	32	32	50
S05	0	0	0	0	100	0	0	0	0	0
S06	0	0	0	0	0	100	0	0	0	0
S07	0	0	0	0	0	0	100	0	0	0
S08	55	47	36	32	0	0	0	100	32	32
S09	63	50	44	32	0	0	0	32	100	32
S10	63	50	90	50	0	0	0	32	32	100

From the results obtained, it can be concluded that SiSe provides meaningful results in all the comparisons, while this is not the case for ssdeep. For instance, the comparison between S01 and S07, which provides a score of 0 in ssdeep, is considered to have a similarity degree of 100% by SiSe. A similar situation occurs when comparing for example S02 and S06, where the results provided by SiSe and ssdeep are 0 and 50, respectively.

Besides, the results provided by SiSe are significantly better adapted to our similarity definition. For instance, when comparing S01 to S03 and S04, it is clear that S03 is almost the same string as S01, whilst S04 only shares with S01 half of its content. However, ssdeep is not able to detect that difference and assigns a value of 50% in both cases. In comparison, SiSe identifies the similarity degree as 100% and 50%, respectively.

6.4. Suitability

The goal of this test is to compare the running time and the signature size of the four applications when processing the following files:

- `lshort.pdf` (613.131 bytes, File 1): The Not so short introduction to LaTeXversion 6.3 [55].
- `bctls-jdk15on-160.zip` (11,933,996 bytes, File 2): One of The Legion of Bouncy Castle's files [56].
- `inkscape-0.92.4-x64.exe` (66,983,821 bytes, File 3): An Inkscape 0.92.4 installer [57].
- `incubating-netbeans-10.0-source.zip` (113,256,313 bytes, File 4): The NetBeans version 10 source code [58].
- `VirtualBox-6.0.4-128413-Win.exe` (219,650,560 bytes, File 5): A Virtual Box installation file [59].
- `HD_Earth_Views.mov` (594,546,566 bytes, File 6): A NASA video taken from their digital archive [60].
- `ubuntu-17.10-desktop-amd64.iso` (1,502,576,640 bytes, File 7): One of the Ubuntu 17.10 distribution files [61].
- `debian-9.6.0-amd64-DVD-1.iso` (3,626,434,560 bytes, File 8): One of the Debian 9.6.0 distribution files [62].

The previous files were selected in order to cover a wide range of file size values, ranging from less than a megabyte to several gigabytes. Table 26 shows the running time in seconds when computing the signature of those files in a desktop with Ubuntu 18.04 operating system, an Intel i7-4790 processor (Intel Corporation, Santa Clara, CA, USA) at 3.60 GHz, and 16 gigabytes of RAM memory. All the applications have been executed using the Linux command `time`, and the output has been redirected to a file in all the cases in order not to penalize sdhash, as its signatures are very long and would provoke an important delay if they were to be printed on the screen.

Table 26. Signature generation time in seconds with the set of eight files tested.

	SiSe	**ssdeep**	**sdhash**	**LZJD**
File 1	0.012	0.005	0.025	0.499
File 2	0.078	0.087	0.326	1.275
File 3	0.394	0.464	0.352	6.066
File 4	0.723	0.821	0.671	10.535
File 5	1.333	1.573	1.158	20.869
File 6	2.680	3.992	3.054	-
File 7	6.529	9.280	8.042	-
File 8	16.048	24.791	18.657	-

As it can be observed in Table 26, SiSe is only slightly slower than ssdeep with very small files, and clearly faster than the other applications with medium and large-sized files. It was not possible to process the larger files with LZJD, so it seems that that application is not well-optimized for files of that size.

Regarding the study on the signature length, Table 27 shows the size of the files that store the signatures, where in each signature file only the hashes for that file have been included.

Table 27. Signature size in bytes with the set of eight files tested.

	SiSe	**ssdeep**	**sdhash**	**LZJD**
File 1	405	154	21,553	5476
File 2	457	172	405,902	5476
File 3	373	150	1,423,030	5476
File 4	283	170	2,405,783	5476
File 5	313	178	4,665,767	5476
File 6	367	152	12,628,901	-
File 7	577	179	31,915,871	-
File 8	343	136	77,028,165	-

It is clear that ssdeep provides the shortest signatures, followed by SiSe. Both applications generate signatures of less than 1 kilobyte. On the other hand, in most of the cases, the size of the signature files of sdhash is several orders of magnitude larger, which could lead to a storage problem when processing volumes with large files. This fact is due to the proportional relationship between the signature length and the input size in sdhash [41], something that does not happen in the case of ssdeep and SiSe. In a test performed by Breitinger and Baier, they found that the signature size when using sdhash is in average 3.3% the size of the original file. Regarding LZJD, the signatures generated by that application always have the same size, but the application was not able to process the three larger files.

Finally, it should also be noted that the complete file path is included in the signatures of SiSe and ssdeep, while sdhash and LZJD only include the file name in their signatures.

7. FRASH Tests

As mentioned in Reference [4], one of the major difficulties when comparing approximate matching algorithms is the diversity of approaches and files tested by each proposal. With that problem in mind, Breitinger et al. designed and implemented a test framework, called FRASH, that could be used for comparing different proposals.

In order to complement the results offered previously with another set of tests used by other authors, we have included in this section the results obtained with FRASH and the t5 dataset [63], a well-known set of 4457 files (1.8 gigabytes) derived from the GovDocs corpus designed to help test approximate matching algorithms [13].

The FRASH framework is implemented in Ruby 1.9.3 and requires a Linux environment to be run. In our evaluation of SiSe, ssdeep, sdhash, and LZJD we have used the same testing environment as in Section 6 (a desktop with Ubuntu 18.04 operating system, an Intel i7-4790 processor at 3.60 GHz, and 16 gigabytes of RAM memory). Instructions for installing the FRASH framework and compiling the four applications are included in SiSe's GitHub page [17].

FRASH allows us to perform five types of tests: efficiency tests, single-common-block correlation tests, fragment detection tests, alignment robustness tests, and random-noise-resistance tests. Information regarding how to lauch those tests can be found in References [4,17].

FRASH includes the Ruby files associated to ssdeep and sdhash. In order to include SiSe and LZJD in the tests, it is necessary to extend FRASH so that it supports both applications. The necessary files for that task are available in Reference [17]. Those files include the headers used by SiSe and LZJD as well as the methods for identifying the signature strings and retrieving the numerical result of a comparison.

Of all the tests implemented by FRASH, it has only been possible to perform (with the four applications) the tests related to efficiency, alignment, and fragmentation exactly as they are provided. The obfuscation test could only be completed with SiSe, ssdeep, and sdhash, due to the excessive time taken by LZJD (more than 3 days for processing just the first file). In comparison, the single-common-block test could not be completed with any application except ssdeep, as the other algorithms got stuck for days processing the first files of the batch being tested. For that reason, we have included the limited comparison regarding the obfuscation test, but have decided not to include the single-common-block test in our results.

7.1. Efficiency Tests

This test is composed of three parts suitably called runtime efficiency, fingerprint comparison, and compression [4]. Runtime efficiency measures the time needed by each application for reading the input files and generating the corresponding fingerprints. Those fingerprints are stored by FRASH in a temporary file so they can be used in the fingerprint comparison, which measures the time needed by the algorithms to complete an all-against-all comparison of the fingerprints. Finally, compression measures the ratio between input and output of each algorithm and returns a percentage value.

Table 28 shows the summary of the efficiency test when using all the files of the t5 corpus as input to the test (the screenshot and text file with the results of the test can be found in Reference [17]). This table contains the digest generation and all-pairs comparison time as well as the average digest length.

Table 28. Efficiency test results.

	SiSe	ssdeep	sdhash	LZJD
Digest generation (s)	8.83	13.50	11.07	22.87
All-pairs comparison (s)	45.17	7.19	137.38	2.26
Average digest length (bytes)	298	57	10,368	4098
Average digest ratio (%)	0.069	0.013	2.417	0.955

As the data included in the table shows, SiSe is the fastest application for digest generation thanks to its double multi-threading capabilities. As expected due to its design (usage of two-character elements, secondary digest larger than the primary digest and lack of digest limits), the average length for the digests generated by SiSe is approximately six times the length of ssdeep digests. It must be noted that FRASH does not compute the size of the digest as the size of the file generated: It removes both the header and the path of the file, and applies a factor of 3/4 to the Base64 string which forms each signature (which is a different approach from the one used in the tests described in Section 6.4, where the total length of the file was taken into account).

Finally, the all-pairs-comparison shows that LZJD clearly outperforms the other algorithms, especially sdhash and SiSe. In the case of SiSe, the fingerprint comparison is slow because of the complex method used in the comparison of digests and its implementation, though it is approximately three times faster than sdhash.

7.2. Alignment Tests

This test analyses the impact of inserting byte sequences at the beginning of an input by means of fixed and percentage blocks [4]. Regarding the interpretation of the results, in the case of percentual addition, a test associated to the value 100% means that new content of equal size to the original size has been added to the file, so if comparing the amount of the larger file (i.e., the modified file) that contains data included in the smaller file (i.e., the original file), in the case of that specific test the result of the comparison should be around 50 (following our similarity definition). In another example, adding 300% of new content means that the result of the comparison should be roughly 25% using the same definition of similarity.

Table 29 includes an extract of the alignment test performed by FRASH over a random selection of 100 files taken from the t5 corpus (the list of files tested and the screenshots and text file with the results of the test can be found in Reference [17]). In each cell, the first element represents the average score and the second one the number of matches (where the maximum is 100, the number of files tested).

Table 29. Alignment robustness test with percentage blocks showing the score (first value) and the number of matches (second value) when testing 100 files taken from the t5 corpus.

	SiSe	**ssdeep**	**sdhash**	**LZJD**
20%	78.41 100	85.67 97	80.08 100	33.50 100
40%	65.55 100	75.14 95	76.75 100	24.75 100
60%	56.69 100	68.66 91	79.12 100	19.98 100
80%	50.07 100	63.48 86	78.58 100	16.86 100
100%	44.83 100	60.68 71	78.76 100	14.74 100
200%	29.08 100	48.21 24	75.68 100	8.91 100
300%	21.79 100	31.00 5	78.68 100	6.52 100
400%	17.53 100	22.00 1	77.52 100	5.14 100
500%	14.45 100	- -	77.69 100	4.35 100

While sdhash is able to provide a match in all the cases, its results are located in the narrow band 70–80%, which does not permit to differentiate between the different testing scenarios. In comparison, LZJD is also able to match all the cases, but it produces values significantly lower than expected in all the tests (e.g., in the 100% test, the result is 14.74%). Both SiSe and ssdeep provide results better adapted to those expected according to our definition of similarity, but SiSe clearly outperforms ssdeep in the number of matches.

7.3. Fragment Detection Tests

Fragment detection identifies the minimum correlation between an input and a fragment (i.e., it determines what is the smallest fragment for which the similarity tool reliably correlates the fragment and the original file). It sequentially cuts a certain percentage (which varies along the test) of the original input length and generates the matching score. This test includes two modes: Random cutting (the framework randomly decides whether to start cutting at the beginning or at the end of an input and then continues randomly) and end side cutting (it only cuts blocks at the end of an input).

Table 30 includes an extract of the fragmentation tests performed by FRASH over the same random selection of 100 files belonging to the t5 corpus (the complete results can be found in Reference [17]). While SiSe and LZJD are able to match all the files, the scores of LZJD deviate significantly more from what is expected in several tests. In comparison to SiSe, both ssdeep and sdhash provide higher outputs, which from our point of view makes SiSe the best option for providing a value according to our similarity definition.

Table 30. Fragment detection test with two cutting modes (random start/end only) and number of matches in each case.

File Size	SiSe	ssdeep	sdhash	LZJD
95%	94.52/94.96 100/100	96.63/97.36 100/100	90.92/99.78 100/100	73.21/92.51 100/100
75%	76.54/79.97 100/100	83.49/88.54 99/100	75.96/99.58 100/100	40.12/65.97 100/100
50%	52.44/54.79 100/100	66.18/71.34 90/90	76.79/99.67 100/100	25.51/39.06 100/100
25%	26.69/29.35 100/100	48.36/60.44 11/25	81.81/99.39 100/98	13.57/19.02 100/100
5%	5.18/5.67 100/100	-/66.00 -/1	68.92/86.75 85/88	3.97/4.60 100/100

7.4. Single-Common-Block and Obfuscation Tests

Unfortunately, it has not been possible to complete the single common block and obfuscation tests in their original form with the four applications. Due to the design of the tests and the characteristics of the four algorithms evaluated, FRASH gets in a loop when processing most of the files used in the previous sections.

Both tests evaluate the algorithms for each of the files entered as an argument and change the files so the score gradually decreases until it reaches the value 0 (which is the halting condition). However, even with files whose size is less than 1 megabyte, LZJD gets active in the loop for several days with just one file, and the same happens (though to a lesser extent) with sdhash and SiSe. The result is that, after several days, there was no progress in the tests.

As an alternative, we were able to perform the obfuscation test with SiSe, ssdeep, and sdhash using a selection of the seven files (each one of a different file type) with minimum size from the t5 batch. The obfuscation test tries to determine what is the maximum number of changes if the match score is requested not to be lower than a certain value. This allows an estimation of how many bytes need to be changed all over the input to receive a non-match. The results of that test are offered in Table 31.

Table 31. Obfuscation resistance test showing the score (first value) and the number of matches (second value) when testing 7 files.

	SiSe	**ssdeep**	**sdhash**
≥90	8.89 7	6.86 7	5.71 7
≥70	33.71 7	26.00 7	21.86 7
≥50	84.57 7	40.50 5	38.86 7
≥30	183.14 7	- -	73.00 7
≥10	526.57 7	- -	182.71 7

The results of this test show that sdhash is the most resistant of the three algorithms. While SiSe and ssdeep offer similar values for the first three cases displayed in Table 31, ssdeep does not produce any output for the rest of the cases and SiSe is able to provide a match in all the cases.

8. Conclusions

In this contribution, a signature generation procedure based on ssdeep and a new algorithm for comparing two file signatures have been presented. We have implemented our tool, SiSe, to fulfil the requirements that any bytewise approximate matching application should satisfy, as indicated in Section 4.1:

(G1) Signature compression: The signature obtained is much smaller than the data file used as input. In this aspect ssdeep provides shorter signatures than SiSe, with both applications clearly outperforming sdhash and LZJD in the vast majority of cases (see Sections 6.4 and 7.1).

(G2) Ease of computation: The signature generation procedure is fast due to the use of the Adler-32 and FNV hashing functions, and the multi-threading design that allows distribution of the workload among the available processor logical cores (see Sections 4.2 and 5.2). As a result, SiSe outperforms ssdeep, sdhash, and LZJD in most of the signature generation tests, specially those with large files.

(G3) Coverage: Every byte of the input file is used by SiSe to process the hash value (see Sections 4.2 and 5.2). As there is no signature maximum length established in our design, it is guaranteed that all the data file is analysed under the same conditions. In this regard, it can be concluded that SiSe outperforms ssdeep, since if ssdeep reaches the maximum number of trigger points (63), the remaining content of the file is managed as a single trigger point, which consequently leads to potentially distinctive information loss.

(G4) Independence regarding the input file size: Thanks to the proposed signature generation procedure, the signature length does not depend on the file size. In this way, it allows us to obtain signatures of less than 1 kilobyte in the majority of cases (see Section 6).

(C1) Generation of a similarity score consistent with the content of the compared files: The score returned by SiSe is ranged between 0 and 100, representing the percentage of a file contained in another file (see Sections 4.2 and 6.1). Based on the set of tests performed, we have shown that, according to the definition of similarity managed in this contribution, SiSe generates results better adapted to that similarity definition than ssdeep, sdhash, and LZJD for most of the tests (see Sections 6 and 7). The results obtained with SiSe are adequate and show low values when comparing data files with different content, reducing the risk of false positives.

(C2) Ease of computation: Due to the design of the complex signature comparison procedure used in SiSe, its performance is not as good as the performance of ssdeep and LZJD, though SiSe clearly outperforms sdhash in tests with multiple files (see Sections 4.2 and 5.2).

(C3) Obfuscation resistance: The possibility (available to the user) to change parameters such as the decremental value, the length of the sliding window or the block size prevents attackers from figuring out which byte string would generate a trigger point. Additionally, as there is no limit in the size of the signature generated by SiSe, it is irrelevant if attackers insert at the beginning of the file 63 (or more) trigger points (see Sections 4.2 and 5.2).

(C4) Content swapping detection: In case data blocks are moved within the file, SiSe readily detects those movements, as shown in the ad-hoc signature tests (see Section 4.3).

(C5) Comparison of files of different sizes: Thanks to the modes made available in the implementation (comparison of two content files, two signature files, and one content file and one signature file), SiSe can compare files of very different sizes (see Sections 6 and 7). However, it must be noted that, given that the score provided by SiSe ranges from 0 to 100, in practice comparing files where the length proportion is greater that 100 (e.g., 1 megabyte and 100 megabytes) will in most cases produce a comparison score of 0, as the content of the larger file also found in the smaller file is expected to be less than 1%.

In the implementation of SiSe we have limited the maximum length to 2560 double characters for performance reasons, though given that the algorithm computes the initial block size with the goal to obtain a leading signature of 64 double characters, the number of files potentially affected by that limitation is practicable negligible.

Our algorithm, which can be applied to forensic operations in different types of digital media, shows results appropriately adapted to the similarity definition that we have used along this contribution. Our implementation is only slower than ssdeep when managing very small files, and faster than the other three algorithms in the rest of cases, specially when computing the signature of large files. Additionally, the signatures generated by our algorithm are much smaller than the ones of sdhash and, to a lesser extent, LZJD, making it more suitable for processing sets of large files.

From a practical standpoint, SiSe is able to compare files without imposing any limitation to their respective size, which consequently allows us to use it in many different situations (e.g., detecting plagiarism, searching for a list of items inside a much larger document, looking for malware hidden in executable files, etc.). Besides, it is able to detect the resemblance in some special cases where the other algorithms considered in this contribution fail.

In our opinion, according to the previously mentioned features, our proposal is a good alternative to the most common tools used nowadays (i.e., ssdeep, sdhash, and LZJD) for evaluating files, producing a clear value representing the similarity degree, specially when the goal is to obtain accurate information about the percentage of a file included in another file.

Author Contributions: Elaboration, validation, reviewing, edition, and resource allocation, V.G.M., F.H.-Á., L.H.E. Funding acquisition, V.G.M., L.H.E. All authors have read and agreed on the published version of the manuscript.

Funding: This work was supported in part by the Ministerio de Economía, Industria y Competitividad (MINECO), in part by the Agencia Estatal de Investigación (AEI), in part by the Fondo Europeo de Desarrollo Regional (FEDER, UE) under Project COPCIS, Grant TIN2017-84844-C2-1-R, and in part by the Comunidad de Madrid (Spain) under Project reference P2018/TCS-4566-CM (CYNAMON), also cofunded by European Union FEDER funds. Víctor Gayoso Martínez would like to thank CSIC Project CASP2/201850E114 for its support.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

BBH	Block-Based Hashing
BBR	Block-Based Rebuilding
BMP	Bitmap
CTPH	Context-Triggered Hashing
FNV	Fowler-Noll-Vo
FRASH	Framework to test Algorithms Of Similarity Hashing
LSH	Locality Sensitive Hashing
LZJD	Lempel-Ziv Jaccard Distance
NIST	National Institute of Standards and Technology
NSRL	National Software Reference Library
SIF	Statistically-Improbable Features
SISE	Similarity Search
SPH	Similarity Preserving Hashing
TLSH	Trend Micro Locality Sensitive Hash

References

1. Stamm, M.C.; Wu, M.; Liu, K.J.R. Information forensics: An overview of the first decade. *IEEE Access* **2013**, *1*, 167–200. [CrossRef]
2. NIST. National Software Reference Library. Available online: http://www.nsrl.nist.gov (accessed on 21 March 2020).
3. NIST. Approximate Matching: Definition and Terminology, SP 800-168. 2014. Available online: https://csrc.nist.gov/publications/detail/sp/800-168/final (accessed on 21 March 2020).
4. Breitinger, F.; Stivaktakis, G.; Baier, H. Frash: A framework to test algorithms of similarity hashing. *Digit. Investig.* **2013**, *10*, 50–58. [CrossRef]
5. Jesse Kornblum. Ssdeep Project. 2017. Available online: https://ssdeep-project.github.io/ssdeep/index.html (accessed on 21 March 2020).
6. Astebøl, K. mvHash—A New Approach for Fuzzy Hashing. Master's Thesis, Gjøvik University College, Gjøvik, Norway, 2012.
7. Baier, H.; Breitinger, F. Security aspects of piecewise hashing in computer forensics. In *Sixth International Conference on IT Security Incident Management and IT Forensics (IMF 2001)*; Morgenstern, H., Ehlert, R., Frings, S., Goebel, O., Guenther, D., Kiltz, S., Nedon, J., Schadt, D., Eds.; IEEE Computer Society: Los Alamitos, CA, USA, 2011; pp. 21–36.
8. Breitinger, F.; Astebøl, K.; Baier, H.; Busch, C. mvHash-B—A new approach for similarity preserving hashing. In *Seventh International Conference on IT Security Incident Management and IT Forensics (IMF 2013)*; IEEE Computer Society: Washington, DC, USA, 2013; pp. 33–44.
9. Breitinger, F.; Baier, H. Performance issues about context-triggered piecewise hashing. In *Third ICST Conference on Digital Forensics & Cyber Crime (ICDF2C)*; Gladyshev, P., Rogers, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 141–155.
10. Breitinger, F.; Baier, H. A fuzzy hashing approach based on random sequences and Hamming distance. In *7th Annual Conference on Digital Forensics, Security and Law (ADFSL 2012)*; Dardick, G., Ed.; Association of Digital Forensics, Security and Law: Ponce Inlet, FL, USA, 2012; pp. 89–100.
11. Breitinger, F.; Baier, H. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In *Digital Forensics and Cyber Crime*; Rogers, M., Seigfried-Spellar, K., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 167–182.
12. Chen, L.; Wang, G. An efficient piecewise hashing method for computer forensics. In *First International Workshop on Knowledge Discovery and Data Mining (WKDD 2008)*; IEEE Computer Society: Los Alamitos, CA, USA, 2008; pp. 635–638.

13. Roussev, V. An evaluation of forensic similarity hashes. *Digit. Investig.* **2011**, *8*, 34–41. [CrossRef]
14. Roussev, V. Sdhash Home. 2013. Available online: http://roussev.net/sdhash/sdhash.html (accessed on 21 March 2020).
15. Raff, E.; Nicholas, C. Lempel-ziv jaccard distance, an effective alternative to ssdeep and sdhash. *Digit. Investig.* **2018**, *24*, 34–49. [CrossRef]
16. Hernández Álvarez, F. Biometric Authentication fo Users through Iris by Using Key Binding and Similarity Preserving Hash Functions. Ph.D. Thesis, Universidad Politécnica de Madrid, Madrid, Spain, 2015.
17. Gayoso, V. SiSe (Similarity Search). 2019. Available online: https://github.com/victor-gayoso/sise (accessed on 21 March 2020).
18. Gayoso Martínez, V.; Hernández Álvarez, F.; Hernández Encinas, L. State of the art in similarity preserving hashing functions. In *Proceedings of WorldComp 2014—International Conference on Security & Management—SAM'14*; Daimi, K., Arabnia, H., Eds.; CSREA Press: Athens, GA, USA, 2014; pp. 139–145.
19. Harbour, N. dcfldd—Defense Computer Forensics Lab. 2002. Available online: http://dcfldd.sourceforge.net (accessed on 21 March 2020).
20. Tridgell, A. Efficient Algorithms for Sorting and Synchronization. Master's Thesis, The Australian National University, Canberra, Australia, 1999.
21. Tridgell, A. Spamsum Readme. 1999. Available online: http://samba.org/ftp/unpacked/junkcode/spamsum/README (accessed on 21 March 2020).
22. Kornblum, J. Identifying almost identical files using context triggered piecewise hashing. *Digit. Investig.* **2006**, *3*, 91–97. [CrossRef]
23. Kornblum, J. Ssdeep Usage. 2017. Available online: https://ssdeep-project.github.io/ssdeep/usage.html (accessed on 21 March 2020).
24. Roussev, V. Building a better similarity drap with statistically improbable features. In *Proceedings of the 42nd Hawaii International Conference on System Science*; IEEE Computer Society: Washington, DC, USA, 2009; pp. 1–10.
25. Roussev, V. Data fingerprinting with similarity digests. In *Advances in Digital Forensics VI*; Chow, K.-P., Shenoi, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 207–226.
26. Roussev, V. Sdhash Tutorial. 2013. Available online: http://roussev.net/sdhash/tutorial/03-quick.html (accessed on 21 March 2020).
27. Sadowsky, C.; Levin, G. Simhash: Hash-Based Similarity Detection. 2007. Available online: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.473.7179&rep=rep1&type=pdf (accessed on 21 March 2020).
28. Blanco, R.; Lioma, C. Graph-based term weighting for information retrieval. *Inf. Retr. J.* **2012**, *15*, 54–92. [CrossRef]
29. Brosseau-Villeneuve, B.; Nie, J.-Y.; Kando, N. Latent word context model for information retrieval. *Inf. Retr. J.* **2014**, *17*, 21–51. [CrossRef]
30. Goyal, P.; Behera, L.; McGinnity, T.M. A novel neighborhood based document smoothing model for information retrieval. *Inf. Retr. J.* **2013**, *16*, 391–425. [CrossRef]
31. Broder, A.Z. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997 (SEQUENCES '97)*; IEEE Computer Society: Washington, DC, USA, 1997; pp. 21–29.
32. Oliver, J.; Cheng, C.; Chen, Y. TLSH—A locality sensitive hash. In *Proceedings of the 2013 Fourth Cybercrime and Trustworthy Computing Workshop*; IEEE Computer Society: Piscataway, NJ, USA, 2013; pp. 7–13.
33. Leskovec, J.; Rajaraman, A.; Ullman, J.D. *Mining of Massive Datasets*, 3rd ed.; Cambridge University Press: Cambridge, UK, 2020.
34. Davison, W. rsync. 1996. Available online: http://rsync.samba.org/ (accessed on 21 March 2020).
35. Deutsch, P.; Gailly, J.-L. Zlib Compressed Data Format Specification Version 3.3. IETF RFC 1950. 1996. Available online: http://tools.ietf.org/html/rfc1950 (accessed on 21 March 2020).
36. Fowler, G.; Noll, L.C.; Vo, K.-P.; Eastlake, D. The FNV Non-Cryptographic Hash Algorithm. IETF Internet Draft. 2012. Available online: http://tools.ietf.org/html/draft-eastlake-fnv-03 (accessed on 21 March 2020).
37. Damerau, F.J. A technique for computer detection and correction of spelling errors. *Commun. ACM* **1964**, *7*, 171–176. [CrossRef]

38. Levenshtein, V.I. Binary codes capable of correcting deletions, insertions, and reversals. *Sov. Phys. Dokl.* **1966**, *10*, 707–710.
39. Karpinski, M. On approximate string matching. *Lect. Notes Comput. Sci.* **1983**, *158*, 487–495.
40. Wagner, R.A.; Fischer, M.J. The string-to-string correction problem. *J. ACM* **1974**, *21*, 168–173. [CrossRef]
41. Breitinger, F.; Baier, H. Properties of a similarity preserving hash function and their realization in sdhash. In *Information Security for South Africa (ISSA 2012)*; Institute of Electrical and Electronics Engineers: Piscataway, NJ, USA, 2012; pp. 1–8.
42. Shapira, D.; Storer, J.A. Edit distance with move operations. *Lect. Notes Comput. Sci.* **2002**, *2373*, 85–98.
43. Chrobak, M.; Kolman, P.; Sgall, J. The greedy algorithm for the minimum common string partition problem. *Lect. Notes Comput. Sci.* **2004**, *3122*, 84–95.
44. Kaplan, H.; Shafrir, N. The greedy algorithm for edit distance with moves. *Inf. Process. Lett.* **2006**, *97*, 23–27. [CrossRef]
45. Cormode, G.; Muthukrishnan, S. The string edit distance matching problem with moves. *ACM Trans. Algorithms* **2007**, *3*, 85–98. [CrossRef]
46. ISO. Programming Languages—C++, ISO Std. 14 882:2017. 2017. Available online: https://www.iso.org/standard/68564.html (accessed on 21 March 2020).
47. OpenMP. The OpenMP API Specification for Parallel Programming. 2018. Available online: https://www.openmp.org/ (accessed on 21 March 2020).
48. Stevenson, R.L. Treasure Island. 2009. Available online: http://www.gutenberg.org/ebooks/27780 (accessed on 21 March 2020).
49. Raff, E. Java Implementation of the Lempel-Ziv Jaccard Distance. 2019. Available online: https://github.com/EdwardRaff/jLZJD (accessed on 21 March 2020).
50. Cervantes Saavedra, M. Don Quijote. 2010. Available online: http://www.gutenberg.org/ebooks/2000 (accessed on 21 March 2020).
51. Wakin, M. Standard Test Images—Lenna. 2003. Available online: https://www.ece.rice.edu/~wakin/images/ (accessed on 21 March 2020).
52. Wells, H.G. The Time Machine. 2011. Available online: http://www.gutenberg.org/ebooks/35 (accessed on 21 March 2020).
53. Verne, J. Voyage au Centre de la Terre. 2011. Available online: http://www.gutenberg.org/ebooks/4791 (accessed on 21 March 2020).
54. Levkin, H. Set of Classic Test Still Images. 2013. Available online: http://www.hlevkin.com/TestImages/classic.htm (accessed on 21 March 2020).
55. Oetiker, T.; Partl, H.; Hyna, I.; Schlegl, E. The Not so Short Introduction to LaTeX Version 6.3. 2018. Available online: http://tobi.oetiker.ch/lshort/lshort.pdf (accessed on 21 March 2020).
56. Legion of the the Bouncy Castle Inc. JCE with Provider and Lightweight API. 2018. Available online: https://www.bouncycastle.org/download/lcrypto-jdk15on-160.zip (accessed on 21 March 2020).
57. Inkscape. Inkscape 0.92.4. 2019. Available online: https://inkscape.org/gallery/item/13318/inkscape-0.92.4-x64.exe (accessed on 21 March 2020).
58. The Apache Software Foundation. Apache NetBeans 10.0 Source Code. 2018. Available online: https://archive.apache.org/dist/incubator/netbeans/incubating-netbeans/incubating-10.0/incubating-netbeans-10.0-source.zip (accessed on 21 March 2020).
59. Oracle, Index of Virtual Box 6.0.4 Downloads. 2019. Available online: http://download.virtualbox.org/virtualbox/6.0.4/VirtualBox-6.0.4-128413-Win.exe (accessed on 21 March 2020).
60. NASA. Earth in HD. 2013. Available online: http://s3.amazonaws.com/akamai.netstorage/HD_downloads/HD_Earth_Views.mov (accessed on 21 March 2020).
61. Canonical Ltd. Ubuntu 17.10 (Artful Aardvark). 2017. Available online: http://old-releases.ubuntu.com/releases/17.10/ (accessed on 21 March 2020).

62. Software in the Public Interest. Debian 9.6. 2018. Available online: https://cdimage.debian.org/mirror/cdimage/archive/9.6.0/amd64/iso-dvd/debian-9.6.0-amd64-DVD-1.iso (accessed on 21 March 2020).
63. Roussev, V. The t5 Corpus. 2011. Available online: http://roussev.net/t5/t5.html (accessed on 21 March 2020).