mvHash - a new approach for fuzzy hashing

Knut Petter Åstebøl

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2012

This master thesis was written as a guest student at Center for Advanced Security Research Darmstadt (CASED) and Hochschule Darmstadt in Germany.

CASED

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Abstract

Effective investigation of crime is important to reduce crime in any society. During criminal investigation, the investigators often face digital evidences. As the amount of digital data seized in a case may be enormous, it's important that the investigators have effective and efficient methods to reduce the number of files subject to manual inspection. The number of files is reduced by identifying known files. It is important to identify files which are identical or similar to known files. One way to automatic identify similar files is by using fuzzy hashing. The leading fuzzy hashing algorithm by now, sdhash, has several weaknesses as it is slow and generates long hashes.

In this thesis, a new and really efficient fuzzy hashing algorithm named fuzzy hashing using majority vote (mvHash) is analysed, developed, implemented and tested. MvHash is based on a simple principle, for each byte in the input and its corresponding neighbourhood, it is calculated whether there is a plural of 0s and 1s. Based on this calculation, a new byte-string is created. The new byte string is then shortened by using run-length encoding (RLE). It is developed two versions of mvHash where the most successful one is named fuzzy hashing using majority vote and Bloom filters (mvHash-B). We used several tests, to ensure that mvHash is able to detect similar files and to evaluate the performance. Using code optimization techniques, it has been possible to make the prototype really fast, it generates its hashes almost as fast as SHA-1. This is way much faster than sdhash. Also noteworthy, the hash size of mvHash-B is significantly shorter than sdhash. MvHash-B, as sdhash, uses Bloom filter to store its hashes, making the comparison really fast.

Acknowledgments

I would like to thank my supervisor ph.d. student Frank Breitinger for excellent guidance. His knowledge within the topic fuzzy hashing and all our interesting discussions have been very valuable in this process and important for the achieved results.

Moreover, Professor Christoph Busch deserves my acknowledgements. He gave me the opportunity to do my thesis at CASED, an outstanding research institute within information security. Also, Professor Harald Baier really deserves to be mentioned, I would thank him for comments and interesting discussions.

Contents

List of Figures

List of Tables

Acronyms

CTPH context triggered piecewise hashing.

mvHash fuzzy hashing using majority vote.

mvHash-B fuzzy hashing using majority vote and Bloom filters.

mvHash-L fuzzy hashing using majority vote and Levenshtein distance.

RLE run-length encoding.

1 Introduction

1.1 Topic covered by the project

The modern society is totally dependent on information systems and information technology. Due to massive use of information technology, a very large proportion of criminal activity use information systems. When the police or other agencies investigate a case, they are often faced an enormous amount of digital data [3, sec. I] [4], which may contain critical evidences.

The main problem is the enormous amount of digital data that have to be considered in an investigation. Thus, an important issue is to quickly reduce the amount of files which need a manual inspection [5]. Both known-to-be-good and known-to-be-bad files may be a part of the reduction.

While there exist good methods for detecting files which are identical to known files [3, sec. 2], it is much more difficult to detect similar files. Today all tools for detecting similar files have some drawbacks. The topic for this master thesis is development of a new approach (fuzzy hashing algorithm) for detecting similar files.

1.2 Keywords

Digital forensics, fuzzy hashing, similarity preserving hash functions, run-length encoding, Bloom filter.

1.3 Problem description

As there is an increasing amount of data for the digital forensics investigators to deal with, it is essential to reduce the amount of data the investigator has to inspect manually. The traditional method for doing this is identifying known files using cryptographic hash functions. The files on the storage medium are hashed and the hashes are compared to hashes in a reference database [6, 3]. There exist reference databases for the purpose of identifying known-to-be-bad files (blacklisting) and known-to-be-good files (whitelisting).

One of the elementary properties of cryptographic hashing is called the avalanche property: If there is only a slight change in the input file, e.g. some few bits, approximately half of the bits in the output string are changed [7, p. 817]. Similar files will therefore have a complete different hash digest. A file on a storage medium may be similar, but not identical, to the file in a reference database. These files will not match, because the hash digest will be completely different compared to the hash digest of the similar file in the reference database. This property of cryptographic hash functions makes it easy for criminals to avoid blacklisting, by changing negligible bits in the file.

There has been some research to find methods for identifying similar files. It exists methods for detecting similar files, where the files have a specific content, such as perceptual hashing for images [8, sec. 1] and text fingerprinting for text documents [9]. Within digital forensics there is need for similar detection working independently of the content of the file. In digital Forensics, huge amounts of data should be expected [3, sec. I] [4].

Thus, it's only feasible to process these amounts if the hash size is small and the time to generate and compare hashes is very short. Another challenge for investigators are the increase in methods and tools for doing anti-forensics [10, 11].

Fuzzy hashing is a kind of hash function working independently of the content of the file, where similar files get a similar hash digests. By comparing hashes, it is possible to identify similar files. The most well-known proposals of fuzzy hashing are block based hashing [3, sec. III-A], context triggered piecewise hashing (CTPH) [5], and sdhash [12]. Sdhash may be considered the leading fuzzy hashing algorithm, as a paper [4] found it significantly better than CTPH. Thus, sdhash has a complex algorithm for generating hashes which makes it slow and its hashes are 2.6% of file size [12, sec. 5.2] which is quite large.

Fuzzy hashing is important to detect similar files within digital forensics, but current approaches have significant disadvantages and a new fuzzy hashing algorithm is necessary.

1.4 Justification, motivation, and benefits

The police and other agencies need robust methods and techniques in order to effectively investigate crime. It is important to improve the methods in line with the development of the society. Within digital forensics there is currently need for a better method to identify similar files [6, sec. VII].

Developing a new and efficient algorithm for detecting similar files will have major benefits. It will not only be possible for digital investigators to keep up with the development in the society, but to make digital forensics more efficient. Agencies will be able to investigate more cases in less time, this leads to more solved crime. More solved crime has a preventive effect, as it potentially leads to less crime.

Better methods for identifying similar files have also other applications. Methods for identifying similar files will be useful in malware detection [13], as well as biometrics and junk mail detection [14].

1.5 Research questions

This thesis is based on an idea of a new fuzzy hashing algorithm which first transform the input into a string of two states and then compress the string by using run-length encoding (RLE).

- Is it possible to build a fuzzy hashing algorithm based on this idea?
- How is the best design of the new algorithm?
- How well is the new algorithm able to distinguish between similar and non-similar files?
- Has the new algorithm the alignment robustness and non-propagation properties?
- How fast is the new algorithm compared to existing approaches?

1.6 Contributions

The main contribution of this master thesis is a new fuzzy hashing algorithm named fuzzy hashing using majority vote (mvHash) and particularly the version named fuzzy hashing using majority vote and Bloom filters (mvHash-B). Compared to leading fuzzy hashing algorithms, the main advantage of mvHash is computation time and hash size. MvHash has a very simple algorithm and generates the hashes very fast and its hashes are very short. Moreover, mvHash produces very few false positive and false negative results.

Another important contribution is the comprehensive analysis of mvHash, making it easy for interested researchers to review and improve mvHash. Source code of mvHash is provided as open-source.

1.7 Methodology

This section outlines the scientific methodology to be used in this thesis. A new suggestion of a fuzzy hashing algorithm must obviously be analysed, before implemented and tested. While a careful theoretical analysis increases the probability of success, it is not possible to know how well the algorithm works without practical testing. Therefore, the overall scientific methodology in this thesis is experimental methodology, and it is divided into the following steps:

Literature study: It has several purposes, first, to find the state of the art of fuzzy hashing. Second, to study how other fuzzy hashing algorithms are built.

Analysis: An analysis will be performed to identify the best design of the new algorithm. Based on the most promising design, a prototype will be implemented.

Practical tests and analyses: There will be performed several tests on the new algorithm. In the tests, the focus is how well the new algorithm is able to recognize similar files and a performance analysis.

2 Related work

This chapter presents and discusses relevant related literature, it has three sections: First, literature considering the concept of fuzzy hashing is described, second, fuzzy hashing within digital forensics is studied, third, properties of hash functions in general is considered.

2.1 Fuzzy hashing

Having a good knowledge of how fuzzy hashing is described in the literature and the most known implementations of fuzzy hashing is important when developing a new fuzzy hashing algorithm. This section starts with studying the concept fuzzy hashing, continuing with the properties non-propagation and alignment robustness. The last part of this section is a presentation of the most well-known fuzzy hashing algorithms: Block-based hashing, CTPH and sdhash. All of them are presented on a very high level which gives a first impression of how fuzzy hashing could be solved.

2.1.1 The concept fuzzy hashing

Fuzzy hashing is a specific type of hashing, used to identify similar, but not necessarily identical files. This subsection presents two non-formal definitions of fuzzy hashing found in the literature.

In [2, p. 1], fuzzy hashing is described as:

> "Fuzzy hashing allows the discovery of potentially incriminating documents that may not be located using traditional hashing methods. The use of the fuzzy hash is much like the fuzzy logic search; it is looking for documents that are similar but not exactly the same, called homologous files. Homologous files have identical strings of binary data; however they are not exact duplicates. An example would be two identical word processor documents, with a new paragraph added in the middle of one".

Another description of fuzzy hashing [15, sec. 2.3]:

> "(...) the term fuzzy-hashing describes a process that can deal with blurred inputs, i.e. two inputs, which only have a few dissimilar bits, result in a nearly identical hash-value".

2.1.2 Non-propagation and alignment robustness

The properties non-propagation and alignment robustness were introduced by Tridgell in the manual of spamsum [16]. These properties are important because they are used as expectations of fuzzy hashing [15, sec. 2.3][13, p. 5]. According to [16] these properties are defined as:

Non-propagation: "In most hash algorithms a change in any part of a plaintext will either change the resulting hash completely or will change all parts of the hash after the part corresponding with the changed plaintext. In the spamsum algorithm only the part of the spamsum signature that corresponds linearly with the changed part of the plaintext will be changed. This means that small changes in any part of the plaintext will leave most of the signature the same. (...)"

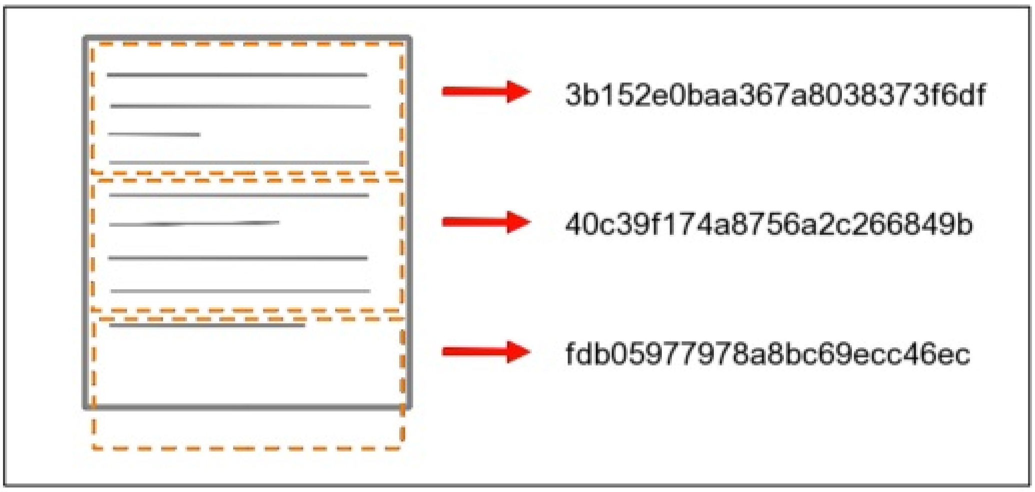


Figure 1: An overview of block-based hashing [1, p. 10].

Alignment robustness: "Most hash algorithms are very alignment sensitive. If you shift the plaintext by a byte (say by inserting a character at the start) then a completely different hash is generated. The spamsum algorithm is robust to alignment changes, and will automatically re-align the resulting signature after insertions or deletions. (...)"

2.1.3 Block-based hashing

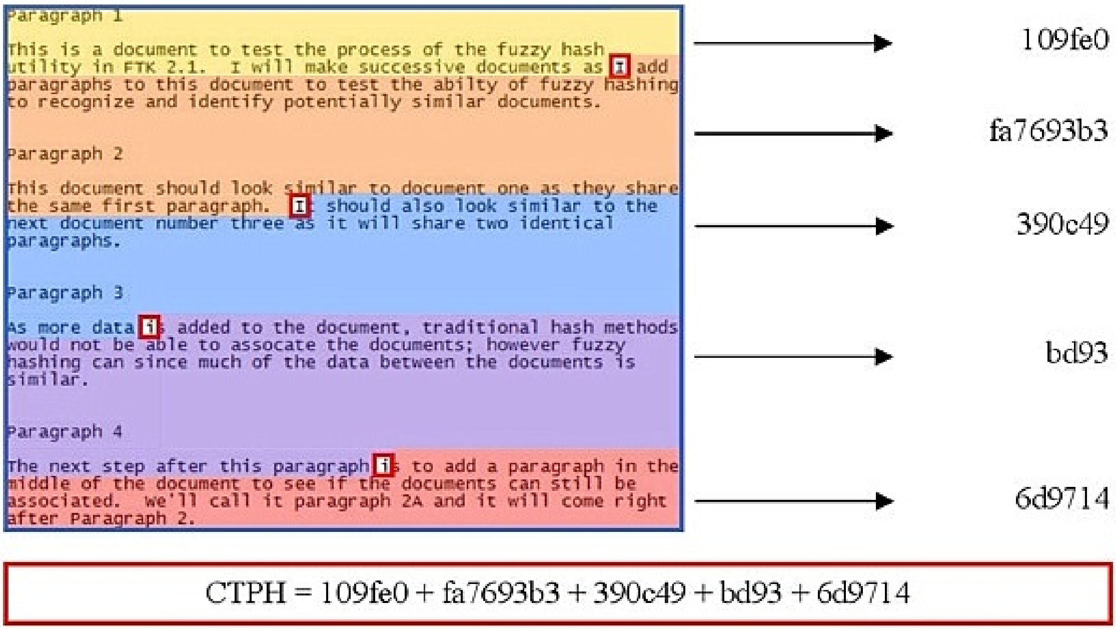
Block-based hashing is a simple fuzzy hashing technique developed by Nicholas Harbour in 2002 [17] [5, sec. 3.1], the technique is illustrated in figure 1. The algorithm separates a file into blocks of fixed size, then each block is hashed separately. These hashes are called piecewise hashes, because each of them only represents the hash of a small block (piece) of the file. Finally, all piecewise hashes are concatenated to a final hash. The length of the resulting hash string depends on the block size, the file size, and the size of the output of the used hash function. This is in contrast to the traditional usage of hash functions, where the entire file is hashed at once, and the output has fixed size regardless of the size of the input.

Using the block-based hashing technique, if one bit is changed, it only causes a change of the piecewise hash value of the corresponding block. This results in only a small change for the entire hash and the algorithm has the property *non-propagation* [16]. On the other hand, if a bit is inserted, this affects all following piecewise hashes. Therefore block-based hashing is not *alignment robust* [18, p. 288].

2.1.4 Context triggered piecewise hashing

The context triggered piecewise hashing (CTPH) algorithm was developed by Jesse Kornblum in 2006 and is implemented in the program ssdeep, CTPH is based on the algorithm Tridgell used in spamsum. CTPH is a complex algorithm, this subsection will introduce the main features of the algorithm as described in [5] and [15, sec. 2.4].

The basic idea is similar to block based hashing, divide an input into blocks and hash each block independently. The main difference is that the blocks are specified by a



Paragraph 1

This is a document to test the process of the fuzzy hash utility in FTK 2.1. I will make successive documents as I add paragraphs to this document to test the abilty of fuzzy hashing to recognize and identify potentially similar documents.

Paragraph 2

This document should look similar to document one as they share the same first paragraph. It should also look similar to the next document number three as it will share two identical paragraphs.

Paragraph 3

As more data is added to the document, traditional hash methods would not be able to assocate the documents; however fuzzy hashing can since much of the data between the documents is similar.

Paragraph 4

The next step after this paragraph is to add a paragraph in the middle of the document to see if the documents can still be associated. We'll call it paragraph 2A and it will come right after Paragraph 2.

Figure 2: An overview of how CTPH is dividing a file into blocks [2, p. 3].

pseudo random function based on the current context. The hash of a single block is called piecewise hash, when all piecewise hashes are concatenated to a hash of the entire file, it is called the ssdeep hash.

An example of CTPH is shown in figure 2. In this figure the end of each block is indicated using a red box, and the blocks are distinguished using different colors. As illustrated in the figure, the piecewise hash is computed for each block. The corresponding ssdeep hash is written at the bottom of the figure.

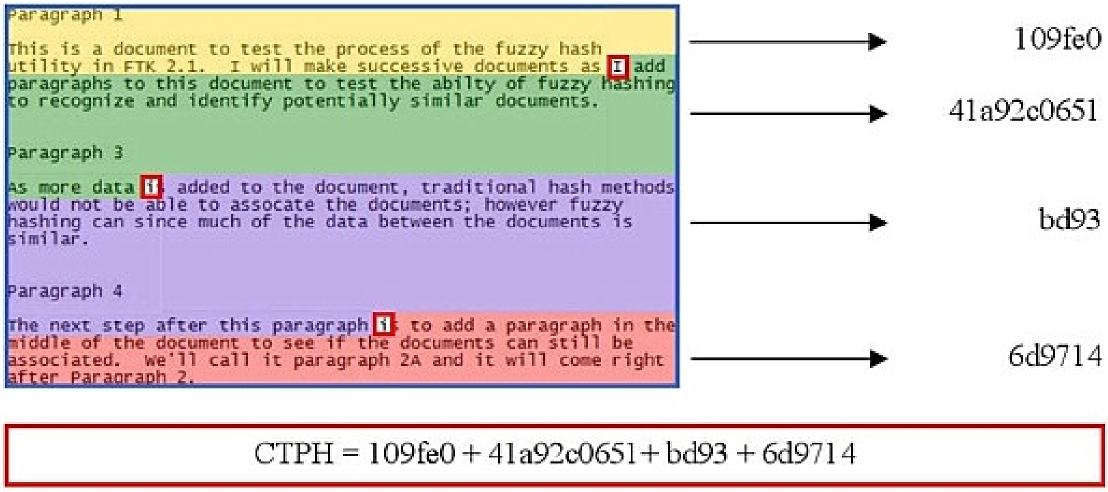
Figure 3 shows an altered version of the document, where paragraph 2 has been removed. Studying figure 3, it's clear that one of the blocks has been removed, while the others are kept. The number of blocks has been reduced from five to four. Still, three of the piecewise hash values from the altered document are among the five piecewise hash values in the original document. This example illustrates the advantage of CTPH when detecting similar files: A change in the input file only affects a few of the piecewise hashes and similar files get a similar hash.

2.1.5 Sdhash

Sdhash was presented by Vassil Roussev in 2010, this subsection gives a brief introduction to the theory of sdhash as described in [12].

An input consists probably of features which are rare and features which are very common, where a feature is a part of the input. Similar inputs will probably have the same rare features, while non-similar inputs will probably have different. The more common rare features, the more similar are probably the input. The general idea of sdhash is to identify statistical rare features in the input and make a hash based on these features.

A feature in sdhash is a 64-byte string, sdhash considers all 64-byte strings in the input and by using a complex algorithm is the rare features identified. The selected features



Paragraph 1

This is a document to test the process of the fuzzy hash utility in FTK 2.1. I will make successive documents as I add paragraphs to this document to test the abilty of fuzzy hashing to recognize and identify potentially similar documents.

Paragraph 3

As more data is added to the document, traditional hash methods would not be able to assocate the documents; however fuzzy hashing can since much of the data between the documents is similar.

Paragraph 4

The next step after this paragraph is to add a paragraph in the middle of the document to see if the documents can still be associated. We'll call it paragraph 2A and it will come right after Paragraph 2.

CTPH = 109fe0 + 41a92c0651 + bd93 + 6d9714

Figure 3: An overview of how the CTPH works after the document is altered [2, p. 3].

are stored using Bloom filters which is an efficient way to store data. A maximum of 128 features are stored per Bloom filter and it is used as many Bloom filters as required to store all the features. The hash of sdhash is a serie of one or more Bloom filters. Comparison of hashes is performed using Hamming distance which makes the comparison fast.

2.2 Fuzzy hashing and digital forensics

The primary usage of fuzzy hashing is within digital forensics. Therefore, it's crucial to know how fuzzy hashing is used within forensics, and it's also important to know relevant attacks against fuzzy hashing. This section starts by studying how fuzzy hashing is used within forensics and concludes with studying the attacks anti-whitelisting and anti-blacklisting.

2.2.1 Usage of fuzzy hashing within forensics

Digital forensics investigators face enormous amounts of data, therefore they try to reduce the amount needing manual inspection by identifying known files [6, sec. II]. The techniques to reduce the amount of data are blacklisting and whitelisting, these are defined as [15, sec. 2.2.3]:

Whitelisting: "A whitelist is a file or database containing known-to-be-good files. An investigator tries to reduce the amount of data by filtering out known-to-be-good files, e.g. files of the operation system. The remaining files are unknown and must be investigated manually. The usage of a whitelist is referred to as whitelisting."

Blacklisting: "A blacklist is a file or database containing known-to-be-bad files and works the other way round. An investigator tries to find known-to-be-bad files e.g. pictures of child abuse. The usage of a blacklist is referred to as blacklisting."

To identify known files, the files are hashed and compared to hash values in a reference database. Traditionally it is used cryptographic hash functions. Within digital forensics, fuzzy hashing will be used for the same purpose as cryptographic hash functions.

2.2.2 Anti-whitelisting and anti-blacklisting

Relevant attacks against a fuzzy hashing algorithm are attacks which prevent the fuzzy hashing algorithm to work correctly. When implementing a new fuzzy hash algorithm, it's crucial to know about relevant attacks. As the task of fuzzy hashing within forensics is whitelisting and blacklisting, relevant attacks are anti-whitelisting and anti-blacklisting. These attacks are defined in the following way [15, sec. 2.6]:

Anti-Whitelisting / false positive: "Let $F1$ be a known-to-be-bad file. What we like to have is a software yielding a file $F2$ which is semantically different but whitelisting identifies a match. In this case a known-to-be-bad file would be identified as a known-to-be-good file.

If the similarity algorithm identifies a match even though both files have nothing in common, this will be denoted as a false positive."

Anti-Blacklisting / false negative: "Let $F1$ be a known-to-be-bad file. What we like to have is a software yielding a file $F2$ which is semantically the same, i.e. a human being will not see a difference between $F1$ and $F2$, but blacklisting will fail.

If the similarity algorithm doesn't identify a match even though both files have the same origin, this will be denoted as a false negative."

2.3 Hash functions

Properties of cryptographic hash functions are among the most well-known properties of hashing and it's important to know them when studying hash functions. This section starts by studying properties of hash functions in general, before it continues with studying security properties of cryptographic hash functions.

2.3.1 Properties of hash functions

This subsection studies general properties of hash functions. A hash function h has the following properties [19, sec. 9.2]:

Ease of computation: Given h and an input x, $h(x)$ is easy to compute.

Compression: h maps an input x of arbitrary finite bit length, to an output $h(x)$ of fixed bit length n. In mathematical terms [20, p. 56]:

$$h : \{0, 1\}^* \rightarrow \{0, 1\}^n,\ n \epsilon N$$

The compression property is very strict and requires that the hash function maps the input to an output of fixed length, regardless of the size of the input file. Several well-known implementations of fuzzy hashing are not strictly following the compression property. Using block-based hashing the hash digest will vary with the size of the file as discussed in section 2.1.3, and using sdhash the size of the digest will be about 2.6% of the size of the file [12, sec. 5.2].

2.3.2 Security properties of cryptographic hash functions

Due to the extensive usage of cryptographic hash functions, their security properties are basic knowledge in all contexts dealing with hash functions, the properties are listed in the following list according to [19, sec. 9.2.2]:

Preimage resistance: "For essentially all pre-specified outputs, it is computationally infeasible to find any input which hashes to that output, i.e., to find any preimage x' such that $h(x') = y$ when given any y for which a corresponding input is not known".

2nd-preimage resistance: "It is computationally infeasible to find any second input which has the same output as any specified input, i.e., given x, to find a 2nd-preimage $x' \neq x$ such that $h(x) = h(x')$".

Collision resistance: "It is computationally infeasible to find any two distinct inputs x, x' which hash to the same output, i.e., such that $h(x) = h(x')$. (Note that here there is free choice of both inputs.)"

The avalanche effect is ensured by these three security properties [15, sec. 2.2.1]. The key of the avalanche effect is that regardless of how many input bits that are changed, the output behaves pseudo randomly and therefore approximately 50% of the output bits are changed [7, p. 817].

3 Basics of fuzzy hashing

In this chapter, it is discussed basic aspects and expectations of fuzzy hashing. This chapter has three sections. First section discusses use-cases for fuzzy hashing, second discusses the term similarity, while the third section outlines some properties of fuzzy hashing.

3.1 Use-case of fuzzy hashing

As mentioned in previous chapters, there are lots of use-cases of fuzzy hashing, where in the literature the main focus is digital forensics.

Fuzzy hashing is very useful for blacklisting as it identifies files similar to known-to-be bad. On the other hand, it is an ongoing discussion in the fuzzy hashing community whether fuzzy hashing is useful for whitelisting[1]. Consider a large whitelisted file, where an active adversary inserts some few lines of code that are used as a backdoor. Using fuzzy hashing, the altered file is detected as similar to the original whitelisted file and will falsely be classified as “white”. Thus, its relevance for whitelisting is unclear.

Therefore, when developing a fuzzy hashing algorithm in this thesis, the use-case is blacklisting within digital forensics.

3.2 Similarity

As fuzzy hashing is all about detecting similar files, it is important to clarify the terminology similarity in the context of fuzzy hashing. Similar files may have two different meanings[2]:

Similar syntax: The bit- and byte-representation of the files are similar.

Similar semantic: When the files are viewed by a program suitable for parsing that file type, e.g. viewing jpg-files in an image viewer or pdf-files in a pdf-reader, a human being will by visual inspection consider the files as similar.

Previous implementations of fuzzy hashing, described in the related work chapter, consider similar syntax and not similar semantic. It is unambiguous in the literature that the purpose of fuzzy hashing is to detect similar syntax, in this thesis similarity always means similar syntax and the focus of the new algorithm will be to detect similar syntax.

It is important to understand the consequence of the fact that fuzzy hashing algorithms are working on syntax level. Files are considered similar by the fuzzy hashing algorithm if they have a similar byte-representation.

On the other hand, two pictures may be considered identical for humans, but one may be stored as a gif-file and the other as a jpg-file. As these files are stored with different file formats, their byte level representation is totally different, they are not similar from a syntax point of view and they will not be considered similar by a fuzzy hashing algorithm.

[1] According to discussion with supervisor Frank Breitinger.

[2] These definitions are based on a discussion with and draft paper by supervisor Frank Breitinger.

3.3 Properties of fuzzy hashing

This section defines some properties of fuzzy hashing, these properties are important as criteria when designing a new algorithm and to be used when the new algorithm will be evaluated.

The properties are designed based on the use-case of section 3.1. Thus, a fuzzy hashing algorithm satisfying the properties, should be suitable for the use-case.

3.3.1 Similarity detection properties

Properties describing the fuzzy hashing algorithm's ability to detect similar files:

Indicating the degree of similarity: Similarity is no yes or no-question. It is important to know how similar the files are and the fuzzy hashing algorithm should be able to indicate the degree of similarity between two files.

Non-propagation and alignment robustness: Non-propagation and alignment robustness consider the case when a file is altered by an edit operation, such that a byte is either inserted, deleted or substituted. The altered file should get a hash similar to the original file, and the fuzzy hashing algorithm should detect them as similar.

Low false negative and low false positive rate: False positive and false negative results may occur, but under normal circumstances these rates should be low. False negative may occur both by chance and due to an active adversary (anti-blacklisting). The low false negative rate refers to cases that occur by chance. Cases including an active adversary are handled by the security properties presented in later sections.

3.3.2 Performance properties

Within digital forensics, large data sets are expected. Large amounts of data require all parts of the fuzzy hashing process to be fast. These properties describe the performance of a fuzzy hashing algorithm:

Ease of computation: Generation of hashes should be fast.

Compression: A short hash will make file operations faster and then faster computation and comparison of hashes. Short hashes are also important when storing huge amounts of hashes. Thus, it is not expected a fixed hash length within fuzzy hashing, but is must be short. In the development of a new algorithm in this thesis, a hash size of 2% of the file size is defined as a maximum.

Ease of comparison: This property describes the ability to fast compare hashes.

3.3.3 Security properties

There is defined one security property of fuzzy hashing:

Difficulty of anti-blacklisting: To do an anti-blacklisting attack should in practice be difficult.

4 Design of mvHash

This thesis is based on an idea of a new fuzzy hashing algorithm which first transform the input into a string of two states (a string of only two states may easily be compressed) and then compress the string by using run-length encoding (RLE), that idea was presented in [14, p. 8].

Based on this overall idea, a new fuzzy hashing algorithm is developed within this thesis. Moreover, it is developed two versions of the algorithm. The algorithm is named fuzzy hashing using majority vote (mvHash) and this name is used as a common name for both versions, the two versions are named fuzzy hashing using majority vote and Levenshtein distance (mvHash-L) and fuzzy hashing using majority vote and Bloom filters (mvHash-B).

Figure 4 gives an overview of mvHash-L and mvHash-B. As shown in this figure, the basic structure is the same, majority vote and run-length encoding. The difference is that mvHash-B uses one more phase, Bloom filter, to make a hash. Moreover, both versions have their own comparison function.

This chapter presents the design of mvHash using three sections: First section presents the basic structure of mvHash, second section presents mvHash-L and third section presents mvHash-B.

4.1 Basic structure

This section presents the basic structure of mvHash, the part which is common for both mvHash-L and mvHash-B.

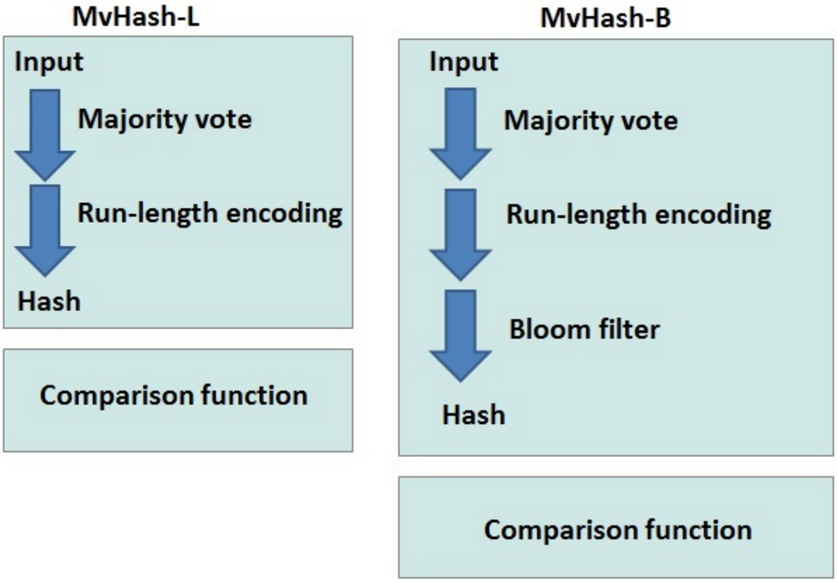


Figure 4: An overview of mvHash-L and mvHash-B.

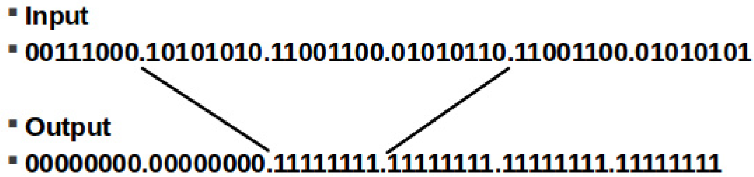


Figure 5: A simple example of majority vote.

This section has two subsections, one for each of the phases of the basic structure: First subsection explains majority vote, second explains RLE.

4.1.1 Majority vote

The concept majority vote is developed for this thesis. It was chosen to use majority vote to transform the input string into a string of two states, due to its advantages: It is a simple algorithm which may be computed fast. Moreover, majority vote has two parameters which may be configured, making it easy to tune the algorithm. Majority vote will be explained in this subsection.

Majority vote makes an output string of the same size as the input string. In the output string is each byte set to either all 0s (byte value 0x00), or all 1s (byte value 0xFF). To determine the value of an output byte, the number of bits of 1s in the corresponding input byte and its neighbourhood is counted. If there is a majority of 1s within the neighbourhood, the output byte is set to 0xFF, otherwise 0x00.

Figure 5 shows a simple example of majority vote, in this figure it's used a neighbourhood of 3 bytes. The neighbourhood used to calculate the output for output byte number 3 is sketched.

The size of the neighbourhood may be adjusted. If using a very small neighbourhood, e.g. only the corresponding input byte itself, then the neighbourhood will change for every output byte, and the majority may change for every output byte. In this case, the granularity will be high, but there will not be long runs in the output of majority vote and the compression by RLE will not be that efficient. If using a very wide neighbourhood like the entire file, then all output bytes will be identical, as all have the same neighbourhood. In this case the RLE-encoded string will be very short, but there will not be any granularity.

The wider neighbourhood, the longer runs, shorter RLE encoded string, but less granularity. The optimal neighbourhood makes long runs and makes it possible to distinguish between similar and non-similar files. The only way of identifying the optimal neighbourhood is by using a test file corpus.

Majority vote limit

To decide whether an output byte will be set to 0x00 or 0xFF, equation 4.1 will be used, where nb is the number of bits of 1 within the neighbourhood, and ns is the neighbourhood size in bytes.

$$nb >= \frac{(ns) * 8}{2} \tag{4.1}$$

If the inequality in equation 4.1 is correct, then 50% of the bits or more is 1, and the output byte will be set to 0xFF. If this inequality is not correct, there is a plural of 0s in the neighbourhood and the output byte will be set to 0x00. In this equation, it's not possible to adjust the minimum proportion of bits which has to be 1 to set the output to 0xFF, the proportion is always 50%. Depending on the distribution of 0s and 1s within the file type, another proportion may be reasonable, therefore majority vote limit, mvl, is introduced and the equation is altered:

$$nb >= \frac{(ns) * (mvl)}{2} \tag{4.2}$$

Studying equation 4.2, if mvl is 8, this is the same as equation 4.1 and at least half of the bits in the neighbourhood must be 1, to set and output byte to 0xFF. By adjusting mvl, the proportion of the neighbourhood which has to be 1, before the output is set to 0xFF may be adjusted. E.g. by adjusting the value to 7, if at least 43.75% of the bits in the neighbourhood is 1, then the output is set to 0xFF.

Neighbourhood

In the implementation, it was decided to always use a symmetric neighbourhood. How the exact neighbourhood is calculated will be explained by an example. If a neighbourhood of 200 is specified, mvHash divides this value by 2 and gets 100. Then it's used a neighbourhood of 100B on each side of the byte in focus, in total a neighbourhood of 201B. If a neighbourhood of 201 is specified, 201 is divided by 2 and rounded down to 100. Once again the neighbourhood is 100B on each side of the byte in focus and in total a neighbourhood of 201B.

Close to the start and end of the file, the neighbourhood will go outside the file, any part of the neighbourhood outside the file is ignored.

4.1.2 Run-length encoding

The output of majority vote will be shortened by using an adjusted version of RLE. How RLE is implemented is described using the listing below. RLE counts the number of consecutive bytes of each type.

- String: 00.00.FF.FF.FF.FF.00.00.00.00.00.00.00.00.00.00.00.00.FF.FF

- RLE: 2|4|12|2

To know whether the first integer in the RLE encoded string represents the number of bytes of 0x00 or 0xFF, it was decided that the first element in RLE encoded string always describes the number of bytes of 0x00. If the first byte in the string is 0xFF, then the first integer in the RLE encoded string should be a 0, describing that the string starts with 0 bytes of 0x00.

4.2 Design of mvHash-L

Figure 4 showed an overview of mvHash-L. The hash of mvHash-L is the RLE encoded string. Thus, there are several ways to configure the RLE encoded string. Moreover, to compare the hashes, a comparison algorithm is required.

This section has four subsections: First subsection describes how the RLE encoded string may be configured, second subsection is an analysis to find a suitable comparison algorithm. Third and fourth describe the chosen comparison algorithm.

4.2.1 Configuration of run-length encoding

In mvHash-L, it must be decided how much space should be used to store each element in the RLE encoded string. This space is named the RLE-width, it may be e.g. 1B or 2B. A problem occurs if the run-length exceeds the value it's possible to store within the RLE-width. This will be handled by using modulus. E.g. if using 1 byte to store each value, the maximum value to be stored is 255. Then, a suitable modulus is 256.

Which RLE-width that is most suitable will be identified when mvHash-L is evaluated in chapter 6 Evaluation.

4.2.2 Analysis and choice of comparison algorithm

The aim of this section is to identify a comparison algorithm suitable for mvHash-L. As there are lots of potential comparison algorithms, it is not feasible to study all of them with respect to mvHash-L. Therefore, the analysis works the other way round. It starts by describing what a hash of mvHash-L looks like and typical difference between hashes which represent similar files. Based on these descriptions, a suitable comparison algorithm will be identified.

A hash of mvHash-L is an RLE encoded string. Thus, the hash is a sequence of numbers, where the numbers represent the number of consecutive bytes of either 0x00 or 0xFF. Similar files will get a similar hash. The difference between two similar hashes may be one or more of the following:

- Substution of a value.
- Insertion or deletion of a value.

Levenshtein distance is a comparison algorithm which counts the minimum number of edit operations between two strings. The edit operations in Levenshtein are insertion, deletion or substitution. As Levenshtein counts the number of edit operations between two strings, and the difference two between hashes is a number of edit operations, Levenshtein seems to fit well. On the other hand, Levenshtein just compares the numbers of edit operations, it doesn't count the significance of an edit operation. The significance describes the significance of the change performed by the edit operations, if 1 is substituted with 55, it is more significant than if 1 is substituted with 2. Not measuring the significance of an edit operation is a disadvantage of Levenshtein distance in this context.

To count the number of edit operations and not the significance is a simplification. On the other hand, as there will be very many values in a hash, the significance of each edit operation is not that important and this simplification is an acceptable disadvantage.

Ease of comparison is an important property of fuzzy hashing. According to [21], the fastest possible algorithm for calculating the Levenshtein distance between s_1 and s_2, $|s_1| \geq |s_2|$ runs in:

$$O\left((|s_1|) * \max\left(1, \frac{|s_2|}{\log(|s_1|)}\right)\right) \tag{4.3}$$

Studying equation 4.3, it means that the Levenshtein distance has a potential of being slow, particularly at large strings. On the other hand, the hashes should be small and when Levenshtein are used at small strings, this is not expected to be any issue.

The hashes are distinguished by a number of edit operations, therefore Levenshtein distance seems as a suitable method to measure the similarity between the hashes and will be used in mvHash-L.

4.2.3 Levenshtein distance

In this subsection, the main aspects of Levenshtein distance is described, the description of Levenshtein is based on [22, ch. 4] and [23]. Readers familiar with Levenshtein distance may skip this subsection.

Basics of Levenshtein distance

Levenshtein distance computes the minimum number of edit operations between two strings. An edit operation is either insertion, deletion or substitution. Let s_1 and s_2 be two strings. To compute the minimum number of edit operations, count the minimum number of edit operations required to transform s_1 into s_2.

Below is an example where the Levenshtein distance between Tuesday and Monday is computed, by counting the number of operations required to transform Tuesday into Monday.

Operation	**Transformation**
Delete t.	tuesday => uesday
Substitute u with m.	uesday => mesday
Substitute e with o.	mesday => mosday
Substitute s with n.	mosday => monday

In the example above, four edit operations were required and the Levenshtein distance between those strings is four. In the expression “minimum number of edit operations”, the word minimum implies that there are several ways to transform one string into another. Levenshtein distance is the way requiring fewest edit operations.

Lower bound

Based on the length of the strings, the minimum possible Levenshtein distance between the strings s_1 and s_2 may be computed. This value is named lower bound and it is computed using equation 4.4.

$$\text{Lower bound} = ||s_1| - |s_2|| \tag{4.4}$$

As a brief explanation of lower bound, assume two strings where the length of the strings are known, but the content of the strings are unknown. The shortest possible Levenshtein distance between the strings occurs if the shortest is a fragment of the longest. Then, to transform the shortest string into the longest, just add the missing characters to the shortest strings. The number of characters to be added is the difference between the lengths of the strings, lower bound.

Upper bound

Based on the length of the strings, it’s possible to compute the maximum possible Levenshtein distance between the strings s_1 and s_2. This value is named upper bound, it is the length of the longest string and it is computed using equation 4.5.

$$\text{Upper bound} = \text{Maximum}(|s_1|, |s_2|) \tag{4.5}$$

As a brief explanation of upper bound, assume two strings where the length of the strings are known, but the content of the strings are unknown. To transform the shortest

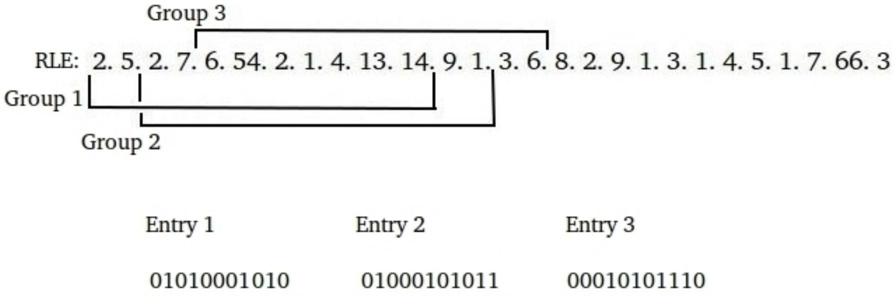

Group 3
RLE: 2. 5. 2. 7. 6. 54. 2. 1. 4. 13. 14. 9. 1. 3. 6. 8. 2. 9. 1. 3. 1. 4. 5. 1. 7. 66. 3
Group 1
Group 2
Entry 1
Entry 2
Entry 3
01010001010
01000101011
00010101110

Figure 6: Explanation of how mvHash-B uses Bloom filter.

string into the longest will at least require to insert characters corresponding to the difference in length (lower bound) and it may require to substitute all characters in the shortest string. If it also is required to substitute all characters in the shortest string, then the total number of edit operations will correspond to the longest string, upper bound.

4.2.4 Mapping the Levensthein distance to a scale

Levenshtein distance is the minimum number of edit operations between two strings, but it doesn't consider the hash size. If the compared hashes are large, a small Levenshtein distance means they are almost similar. The other way round, a small Levenshtein distance between two small hashes indicates they are totally different.

Therefore, the Levenshtein distance is mapped to a scale from 0 to 100 using an equation which considers the hash size. A score of 100 means very similar, 0 means not similar. Equation 4.6 is used to map the Levenshtein distance to the scale, the outcome for one specific comparison will be named score.

$$\text{Score} = 100 - 100 * \frac{\text{Levensthein distance}}{\text{Upper bound}} \tag{4.6}$$

In this equation the Levenshtein distance is considered relative to the upper bound. Thus, the difference between the hashes is considered relative to the maximum possible difference between the hashes. In mvHash-L, this equation is used to calculate the score when two hashes are compared.

4.3 Design of mvHash-B

In this section is the design of mvHash-B explained. MvHash-B is an extension of the basic structure of mvHash, it's added one phase, Bloom filter. This section has two subsections: First is the usage of Bloom filter explained, second is the comparison algorithm explained.

4.3.1 Bloom filter

Bloom filter is a way to store values, it was first described in [24]. In mvHash-B is Bloom filter used as it has a potential of making the comparison algorithm really fast. In a Bloom filter, the components are stored independent of the ordering in the original string. Thus, the comparison algorithm may use Hamming distance and calculation of Hamming distance is really fast as it only requires the low-level operations XOR and bit count.

How mvHash-B uses Bloom filters will be explained in the following. MvHash-B uses Bloom filters to store the RLE encoded string. A Bloom filter is 256 bytes, an empty Bloom filter is a string where all bits are 0. How Bloom filters are used to store the RLE encoded string will be explained by using figure 6. The RLE encoded string is grouped into groups of 11 elements, a new group starts for every second element in the RLE encoded string. From group 1, it's created entry 1, from group 2, entry 2, etc. To create entry 1 from group 1, each element in group 1 is processed by using modulo 2. Thus, an entry is an 11 bit long bit string. As the Bloom filter is 2048 bits, 11 bits are required to address one bit within the Bloom filter. For each entry, the bit in the Bloom filter which is addressed by the value of the entry is set to one. Thus, 1 bit in the Bloom filter will represent each group/entry. The maximum number of entries per Bloom filter must be selected, if there are more entries than the maximum number, a new Bloom filter is created. That way, a hash of mvHash-B is a serie of one or more Bloom filters.

As a new group/entry starts for every second element in the RLE encoded string, the groups will be overlapping. In the following paragraph, it will be described why it's required to start a new group for every second element in the RLE encoded string. Readers not interested in this detail may skip the rest of this subsection.

To understand this issue, it must first be studied how the RLE encoded string differs between similar files. The RLE encoded string is made of the output of majority vote, and the majority vote changes based on changes in the input file. In the following it's studied how a change in the output of majority vote changes the RLE encoded string and thus how the entries of the Bloom filter are changed. A change in the RLE encoded string will be divided into two types.

Change 1

The following listing describes this change:

```
Majority vote: 00.00.00.00.FF.FF.FF.FF.FF.FF.00.00.00.00.00.FF
RLE: 4.6.5.1

Majority vote: 00.00.00.00.FF.FF.FF.FF.FF.FF.FF.00.00.00.00.FF
RLE: 4.7.4.1
```

In this listing, it's shown two examples from the output of majority vote and their corresponding RLE encoded string. From the upper to the lower has majority vote changed one byte from 0x00 to 0xFF, this byte is in the end of a serie of 0x00-bytes, so it only affects two elements in the RLE encoded string. One element in the RLE encoded string is increased by 1, and one element has been 1 value lower, no elements are added or removed from the RLE encoded string. This change type also includes the other way round, where a 0xFF byte is changed to 0x00.

Change 2

The following listing describes this change:

```
Majority vote: 00.00.00.00.FF.FF.FF.FF.FF.FF.FF.00.00.00
RLE: 4.7.3

Majority vote: 00.00.00.00.FF.FF.00.FF.FF.FF.FF.00.00.00
RLE: 4.2.1.4.3
```

In this listing, from the upper to the lower string, a 0xFF-byte in the middle of a string of 0xFF-bytes has been altered to 0x00. The run of 7 0xFF-bytes has been split into 3 runs and element 7 in the RLE encoded string is changed to 2.1.4. It means that 2 new elements are introduced in the RLE encoded string. It may also be the other way round, if the bottom byte string was the original string, in this case 2 elements would have been removed from the RLE encoded string.

When a change in this category occurs, 2 elements in the RLE encoded string are either introduced or removed and all subsequent elements are moved 2 positions. Thus, if a new group/entry for the Bloom filter is picked more seldom than for every second byte, all subsequent entries after the change will be changed and similar files will get a different hash.

4.3.2 Comparison algorithm

A hash of mvHash-B consists of one or more Bloom filters, when comparing two hashes, name the hashes A and B. The easiest way to perform the comparison would be to compare each Bloom filter in A against the corresponding Bloom filter in B, such that the first Bloom filter in A is compared against the first in B, etc. The disadvantage of this method occurs e.g. if there is inserted some bytes in the beginning of the file, then the entries which originally was stored in first Bloom filter is moved to second Bloom filter, the entries from second is moved to third, etc. Therefore, the comparison algorithm is designed such that there is performed an all-against-all comparison between all Bloom filters in A and B. That way each Bloom filter may identify which Bloom filter in the other hash which is most similar. The details of the algorithm are explained in the following.

The comparison algorithm outputs the results to a scale from -1 to 100, 100 means very similar and 0 means not similar. If the difference between the sizes of the hashes is significant, it is unlikely that their corresponding files are similar and they get the score -1 without being compared.

When two hashes are compared, name the shortest hash A and the longest B. To compare two hashes, each Bloom filter in A is compared against each Bloom filter in B. For each Bloom filter in A, it is calculated which Bloom filter in B that is most similar and the value representing this similarity is kept. Thus, there exists one similarity value for each Bloom filter in A and these similarity values are used to map the similarity between A and B to a scale from 0 to 100.

When a Bloom filter of A is compared to a Bloom filter of B, it is calculated a value representing the similarity between these Bloom filters. This value is calculated by calculating the Hamming distance at bit-level between the Bloom filters. The Hamming distance represents the number of bits which differ between the Bloom filters and the lower Hamming distance, the more similar.

5 Basics of testing and evaluation

No well-known and renowned evaluation methodology for fuzzy hashing exists. In this chapter, it is designed a methodology which will be used in the evaluation of mvHash. Moreover, other important aspects related to the testing and evaluation of mvHash will be described.

First section of this chapter describes the evaluation methodology, second describes the test file corpuses to be used. Section three describes how execution time is measured, while section four describes which other fuzzy hashing algorithms mvHash is compared to and why.

5.1 Evaluation methodology

In chapter 3, Basics of fuzzy hashing, there was defined some properties of fuzzy hashing and the evaluation methodology is based on these properties.

Note, even there was defined one security property of fuzzy hashing, to evaluate the security is not a part of the evaluation methodology and the security of mvHash will not be evaluated in this thesis. The reason is that a security analysis of a fuzzy hashing algorithm is very comprehensive and it is a master thesis itself.

The evaluation methodology consists of a serie of tests which are separated into three groups and described in the following subsections.

5.1.1 Similarity detection tests

The purpose of these tests is to determine whether the fuzzy hashing algorithm has ability to detect similar files, the purpose is not to state how well it detects similar files. Following tests are within this category:

The alignment robustness test: This test is used to test whether the fuzzy hashing algorithm has the alignment robustness property. To do this test, one file from the test corpus is chosen and copied. Using a hex-editor one of the first bytes in the copy is removed. If the fuzzy hashing algorithm has the alignment robustness property, then the original file and the copy get a high score when compared against each other. To compare the files against each other, the files are hashed and they are compared using the comparison function.

The non-propagation test: This test is used to test whether the fuzzy hashing algorithm has the non-propagation property. To do this test, one file from the test corpus is chosen and copied. Using a hex-editor one of the first bytes in the copy is substituted with another byte. If the fuzzy hashing algorithm has the non-propagation property, then the original file and the copy get a high score when compared against each other.

General similarity detection test: An essential aspect of a fuzzy hashing algorithm is the ability to detect how similar the files are and to give a higher score the more similar the files are. To test this aspect, it is used a method similar to the method used in [22, sec. 6.6.2]. It is chosen one file and there is made 1000 copies of the selected file. In copy 1 there is performed 1 edit operation, in copy 2, 2 edit operations, etc. An edit operation is either insert, delete or substitute a byte. The edit operations are chosen randomly and there is equal probability for each edit operation. For the edit operations substitute and insert, the new byte is randomly chosen. Each of the copies is compared to the original file using the fuzzy hashing algorithm. The fuzzy hashing algorithm works correctly if the lower number of edit operations, the higher score is the outcome.

5.1.2 Measure the capability to detect similar files

The purpose of the tests in this subsection is to measure how well the fuzzy hashing algorithm is able to detect similar files.

The comparison function makes a score from 0 to 100, when using the algorithm in practical work, a threshold is needed, any score equal to or above the threshold, means that the files are similar, when two files get a score below the threshold, they should be non-similar. An optimal way to evaluate the fuzzy hashing algorithm would be to calculate the false match rate vs false non-match rate for different thresholds. To do this calculation would require a test corpus where each file pair is classified as either similar or non-similar. Such a test corpus doesn't exist and making it would be very time consuming, therefore another method is used.

In this method, it is first identified a threshold which make very few false positive results. When this threshold is identified, there is measured a value named the edit operation ratio, the edit operation ratio tells how much a file may be edited and still be detected as similar to the original file. The edited file is detected as similar if the score is equal to or above the threshold when compared to the original file. A higher edit operation ratio means better capability to detect similar files and the edit operations ratio will be used to measure how well the fuzzy hashing algorithm is able to detect similar files. The details about how a threshold is identified and how the edit operations ratio is calculated are described below.

To identify a suitable threshold, the following method is used:

All-against-all comparison: A randomly collected test corpus is used and there is performed an all-against-all comparison between the files in the test corpus. Due to the high amount of files in the comparison, it's not feasible to inspect all file pairs manually to determine whether or not they are similar. As the files are randomly collected, it's reasonable to assume that most file pairs are non-similar. By analysing all the file pairs which got a high score, it is possible to identify how high score it is likely that two non-similar files will get.

In some cases there may be identified two or more thresholds, the upper threshold is a threshold where very few false positive results are expected, while using the lower threshold, few false positive results are expected.

When a threshold is identified, the edit operation ratio is calculated. The edit operation ratio tells in average how many percent of the bytes in the file may be edited and the edited file will still be detected as similar to the original file. The edit operation ratio is calculated by the following algorithm:

Edit operation ratio calculation: To calculate the edit operation ratio, a test corpus is used. The edit operation ratio is the average of the individual edit operation ratio for all files in the test corpus. To calculate the individual edit operation ratio for a specific file, the following method is used: It is made a copy of the file and performed a specified number of random edit operations on the copy. Then the original file and the copy are compared to each other. The number of edit operations on the copy is slight increased until the score is below the threshold. When the score is below the threshold, the individual edit operation ratio is calculated by dividing the number of edit operations by the size of the original file in bytes.

5.1.3 Performance

The tests in this section are used to measure the performance of the fuzzy hashing algorithm.

Hash size: The compression is evaluated by measuring the hash size. The hash size is measured in percent of the file size. For each of the files in the test corpus, the hash size is measured in percent of the corresponding file size and the average calculated.

Computation time - reference test: 500 files from the test corpus are selected and the time to hash them is calculated.

Computation time - graph presentation: It is generated 50 random files from 100kB to 5MB. For each file, the time to hash the file 1000 times is measured. The result will be presented as a graph of computation time vs file size.

Comparison time - reference test: 50 files from the test corpus are selected and hashed, the time to do an all-against-all comparison between them 1000 times is measured.

5.2 Test file corpuses

When testing mvHash it will be used two test file corpuses, the corpuses are named configuration corpus (c-corpus) and t5-corpus. The c-corpus will be used for configuration and testing, and the t5-corpus will be used to verify that the results also apply for another corpus than the corpus used for the configuration. There is no intended overlap between these corpuses and they will be considered independent.

Configuration corpus (c-corpus)

This corpus consists of pdf, txt and doc files from a corpus sampled by Georg Ziroff [22, p. 51] and jpg-files which are sampled for this thesis. The corpus of Ziroff also contains jpg-files, but they are the same as the jpg-files in the t5-corpus and they are therefore not used in this corpus to make it independent of the t5-corpus.

Ziroff collected the pdf, txt and doc-files by using various key words in a search engine and the jpg-files of the c-corpus are also collected in this way. Table 1 shows the statistic of the c-corpus.

	jpg	doc	pdf	txt
Number of files	2067	2000	1724	2000

Table 1: Statistic of the c-corpus.

t5-corpus

The t5-corpus is collected by Roussev based on the GovDocs corpus [4, sec. 4.1]. Statistic of the t5-corpus is shown in table 2.

	jpg	doc	pdf	txt
Number of files	362	533	1073	711

Table 2: Statistic of the t5-corpus.

5.3 Measuring time

When testing the performance, the following platform will be used:

Laptop:	Acer Aspire 5738Z, 32 bit
Operating system:	Ubuntu 11.10
Memory:	4GB
Processor:	Pentium(R) Dual-Core CPU T4300 @ 2.10GHz

The time is measured by using the /usr/bin/time function. It is measured the sum of CPU-time the system used behalf of the algorithm and the CPU-time used directly by the algorithm.

5.4 Relevant comparison

As a new fuzzy hashing algorithm is developed in this thesis, the best way to evaluate how well it works is by comparing it to existing approaches. In the literature there is described two important fuzzy hashing approaches, CTPH and sdhash. An evaluation which compares CTPH and sdhash found sdhash significantly better [4], and it is therefore chosen to compare mvHash to sdhash. Sdhash is downloaded from [25] and it is used the most recent version from time when the testing started, sdhash version 1.7.

When mvHash is compared to sdhash, the same tests must be performed for both algorithms. For sdhash, the hash size and threshold are not calculated in this project, as it was calculated by others in [12, sec. 5.2] and [4].

Sdhash has a complex algorithm for generating hashes and is expected to be slow. To prove the speed of mvHash, the time to compute a hash is compared to the cryptographic hashing algorithm SHA-1 which is known to be very fast.

6 Evaluation

In this chapter, mvHash-L and mvHash-B will be evaluated. Note, to evaluate the algorithms, the tests which were described in section 5.1 Evaluation methodology will be used. During the evaluation each of the two versions of mvHash will be compared against the leading fuzzy hashing algorithm sdhash.

This chapter has three sections: First, initial analysis, second and third, evaluation of mvHash-L and mvHash-B, respectively.

6.1 Initial analysis

MvHash has several configuration options. Some initial tests were performed to get an idea of which configurations that most likely will succeed. The tests in the initial analysis focus on mvHash-L, but some of the results are also important in the context of mvHash-B.

This section has two subsections: First, the distribution of run-lengths for different configurations is calculated, second, the average hash size for different configurations of mvHash-L is calculated.

6.1.1 Distribution of run-lengths

First phase of mvHash, majority vote, creates long runs which later are compressed using RLE. The length of a run will obviously vary with the neighbourhood and probably with the file type. In this section the distribution of the run-lengths is calculated for different neighbourhoods and file types. Some different neighbourhoods were chosen and the calculation is performed by using the following specification:

File type:	jpg, pdf, txt and doc.
Neighbourhood:	20, 50, 100, 200.
Majority vote limit:	8, default.

For each of the four file types, the test is performed by using the four listed neighbourhoods, in total 16 tests. For each test, it is used all the files of that type from the c-corpus, the files are processed by majority vote and there is made statistic of the run-lengths. The test was implemented such that run-lengths exceeding 65535 was counted as 65535. It's assumed that only some few run-lengths will exceed 65535, so this disadvantage is accepted.

The results of the test[1] show that, for all configurations, the upper quartile is very low, while the maximum length and the average is high. As the upper quartile is low, most of the runs are short. The average is high due to some few very long runs. Another important aspect which was disclosed by these tests; it is significant difference between the average run-length for the different file types.

Recall that in mvHash-L the hash is the RLE encoded string and the space used to store each element in the RLE encoded string is named RLE-width. The shorter RLE-width, the shorter hash, but a smaller modulus has to be used. E.g. an RLE-width of

[1] The results are written in table 17 in appendix A Tables.

1B requires modulus of 256, while an RLE-width of 2B requires modulus of 65536. The disadvantage of modulus is that it makes different values identical, e.g. a modulus of 256 makes 1 and 257 the same value. Therefore, it is an advantage if the modulus does not affect too many values. Due to the low upper quartile, it is possible to identify a modulus which not affects many run-lengths. By using a modulus of 256, the modulus will only affect some very few run-lengths, as 256 is way much higher than the upper quartile for all cases. 256 is also chosen as it is suitable with an RLE-width of 1B. Therefore, for mvHash-L, there will in this thesis be used modulus 256 and RLE-width of 1B.

6.1.2 Hash size of mvHash-L

In this subsection the hash size for different neighbourhoods of mvHash-L is computed and analyzed. In the previous subsection, it was disclosed that for a specific neighbourhood, the average run-length varies between the file types. Longer runs imply fewer elements in the RLE encoded string and a shorter hash. As the distribution of the run-lengths differs for each file type, it's expected that the hash size also differ for different file types and the calculation of hash size in this subsection will be performed per file type. It is used the same neighbourhoods as in previous subsection. The following configuration is used:

File type:	jpg, pdf, txt and doc.
Neighbourhood:	20, 50, 100, 200.
RLE-width:	1B.
Modulus:	256.
Majority vote limit:	8, default.

Studying the configuration, it is four file types and four neighbourhoods, which mean in total 16 tests. For each test, mvHash-L is used to hash all the files of that file type in the c-corpus and the average hash size is calculated. The results[2] show that for each neighbourhood, the average hash size varies significantly with the file type.

A longer hash has higher granularity than a smaller hash. For a specific neighbourhood, the difference of hash size, and then granularity, between the file types is so significant, that it's unlikely the same neighbourhood will fit all file types. Therefore, the subsequent evaluation of mvHash-L will be performed per file type.

The hash of mvHash-B is based on the RLE encoded string which is the hash of mvHash-L. Thus, as it is unlikely the same neighbourhood will fit all file types of mvHash-L, it is unlikely the same neighbourhood will fit all file types of mvHash-B and the evaluation of mvHash-B will also be performed per file type.

6.2 Evaluation of mvHash-L

In previous section, it was decided to test mvHash-L for each file type individually, the tests will be performed at jpg- and doc-files. These file types are chosen such that mvHash-L is tested for one compressed file type (jpg) and one uncompressed file type (doc). This section has two subsections: First, mvHash-L is evaluated for jpg-files, second, mvHash-L is evaluated for doc-files.

[2] The results are written in table 18 in appendix A Tables.

6.2.1 Evaluation for jpg-files

The evaluation is performed using five steps: First, a suitable configuration is chosen, second, we tested if mvHash-L is able to detect similar jpg-files, third, the threshold and edit operation ratio are calculated, fourth, the performance is evaluated and fifth, a summary.

Choice of configuration

In chapter 3, Basics of fuzzy hashing, it was defined that when designing a new algorithm, the hash size should be below 2%. The hash size for different configurations of mvHash-L was computed in a previous section, the results are shown in table 18 in appendix A Tables. For jpg, the only configuration with a hash size below 2% is when using a neighbourhood of 200B. Thus, to test mvHash-L for jpg-files, the following configuration will be used:

Neighbourhood:	200.
RLE-width:	1B.
Modulus:	256.
Majority vote limit:	8, default.

Similarity detection

The purpose of this paragraph is to test whether mvHash-L for jpg-files is able to detect similar files. The alignment robustness and non-propagation tests are used and showed that mvHash-L has the properties alignment robustness and non-propagation.

The general similarity detection test was performed by choosing a jpg-file from the c-corpus and the result is presented in table 3. A short explanation about how to read this table follows. In the table, it's written that if there was a maximum of 10 edit operations, all scores were above 99. Thus, for all copies of the file where it was performed 10 or less edit operations, the hash of the copy got a score of 99 or higher when compared to the hash of the original file. When there was performed maximum 50 edit operations, the score was 91 or higher. Studying the table, few edit operations give a high score and the more edit operations, the lower score. This implies that mvHash-L is able to detect similar files and gives similar files a high score.

File: book-and-pen.jpg.	
File size: 110kB.	
Hash size: 1,8kB.	
Maximum number of edit operations	All scores above
10	99
50	91
100	87
150	79

Table 3: The general similarity detection test for mvHash-L for jpg-files.

Threshold and edit operations ratio

In this paragraph, a threshold is identified and the edit operations ratio is calculated. To identify the threshold, we performed an all-against-all comparison between the jpg-files in the c-corpus. The results are summarized in table 4. By analysing the results, 63 may be a suitable threshold, as it seems that non-similar files will get a score below 63. 35 may also be a usable threshold, where few false positive results are expected.

Score	Non-similar	Similar
63-100:	0	0
35-62:	6	0
0-34:	Ca 2 million	0

Table 4: The results of an all-against-all comparison of the jpg-files in the c-corpus using mvHash-L.

To test whether mvHash-L with the specified configuration makes similar results for another file corpus, the all-against-all comparison was performed at the jpg-files of the t5-corpus and the results are written in table 5. Most of the scores are 33 or lower, this is corresponding to the c-corpus where almost all scores were 34 or lower. This result confirms that the suggested thresholds are suitable.

Score	Non-similar	Similar
37-100	0	0
34-36	1	0
0-33	Ca 65000	0

Table 5: All-against-all comparison of the doc-files in the t5-corpus using mvHash-L

The edit operation ratio was calculated by using the jpg-files in the c-corpus. In table 6, the result of the calculation of the edit operation ratio is shown, the test was performed for the thresholds 63 and 35.

For sdhash, the threshold for non-text files is 21 [4, sec. 4.2.2] and the edit operations ratio was calculated to 0.92%. As the edit operations ratio is larger for sdhash than mvHash-L, sdhash is able to detect files which are more modified than mvHash-L.

Threshold	63	35
Edit operation ratio	0.27%	0.83%

Table 6: Edit operations ratio for jpg-files using mvHash-L.

Performance

In this paragraph is the performance of mvHash-L evaluated.

First, the hash size will be considered. The average hash size is 1.4% which is short compared to sdhash of 2.6%, but if even shorter hashes are possible, it will be an advantage. To get a shorter hash, the neighbourhood must be extended, which will make longer runs and shorter hashes. To study potential negative consequences of an extended neighbourhood, the smallest hashes were studied. When all the jpg-files in the c-corpus were hashed, it was disclosed that some hashes were really small, the smallest one was 1 byte and 17 hashes were less than 10 bytes. The short hashes have occurred as majority vote has generated some very long runs. A very short hash is a disadvantage as it has a potential of false negative. E.g. consider the file which was hashed to an output of 1 byte, it means that majority vote has changed the entire file into one long run of 0s. By adding a byte of 0x00 in the end of this file, majority vote will still create just one run of 0s, but the value of the hash will increase by 1. Thus, a hash of the modified file compared to

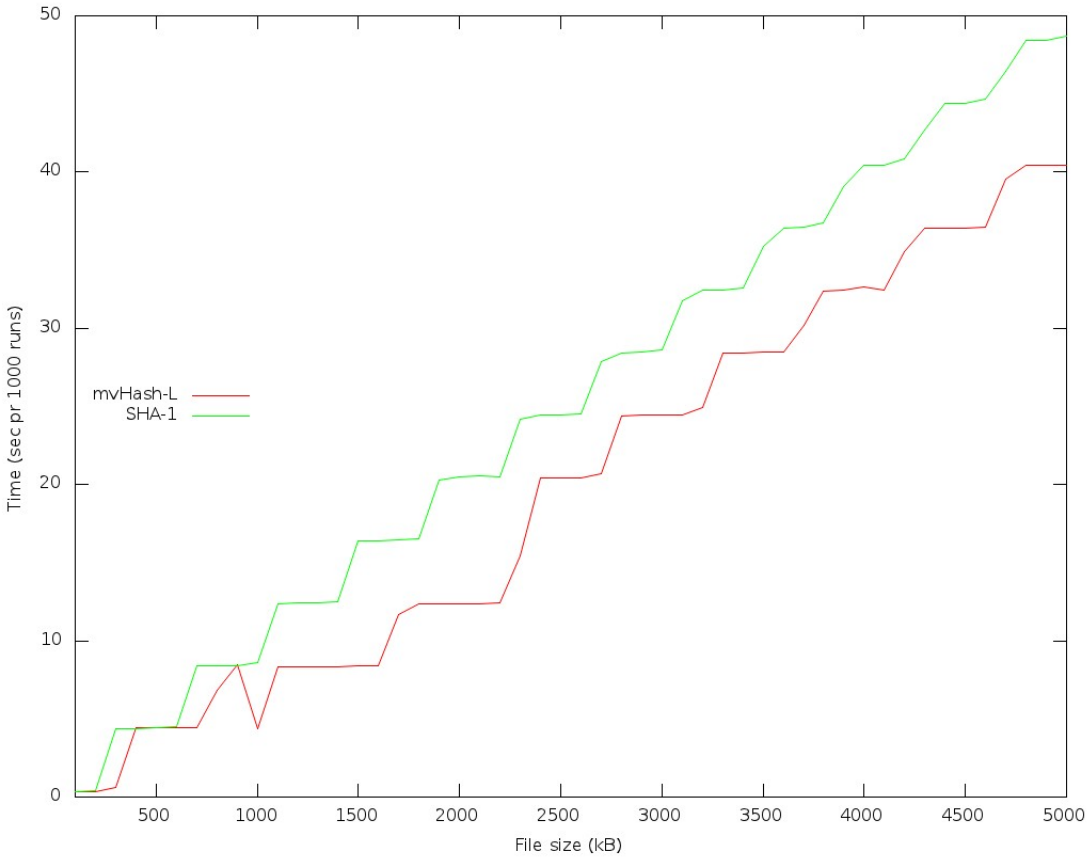


Figure 7: Computation time for mvHash-L and SHA-1.

the hash of the original file will get a score of 0, a false negative. As the current configuration is making some few small hashes, an extended neighbourhood which decrease the average hash size, but also make even more small hashes is not favourable.

It was disclosed that the current configuration is making some really short hashes and it would be an advantage to get rid of the small hashes. To get rid of the small hashes, a shorter neighbourhood must be used, as a smaller neighbourhood increases the granularity. This is not a good solution, as it makes the average hash size longer. Thus, the current configuration seems as the best trade-off.

As ease of computation is important in fuzzy hashing, code optimizing techniques are used. The test named "computation time - graph presentation" was performed and in figure 7, the time mvHash-L uses to generate hashes is compared to SHA-1, in figure 8 mvHash-L is compared to sdhash. For all file sizes, mvHash-L is slightly faster than SHA-1 and significantly faster than sdhash.

To measure the comparison time, the reference test for comparison time was used. MvHash-L used 10264 seconds while sdhash used 35 seconds. This is a very huge difference. Thus, the comparison is not totally fair. Using sdhash all hashes are put into one file and sdhash opens this file and performs the all-against-all-comparison. In mvHash-L each

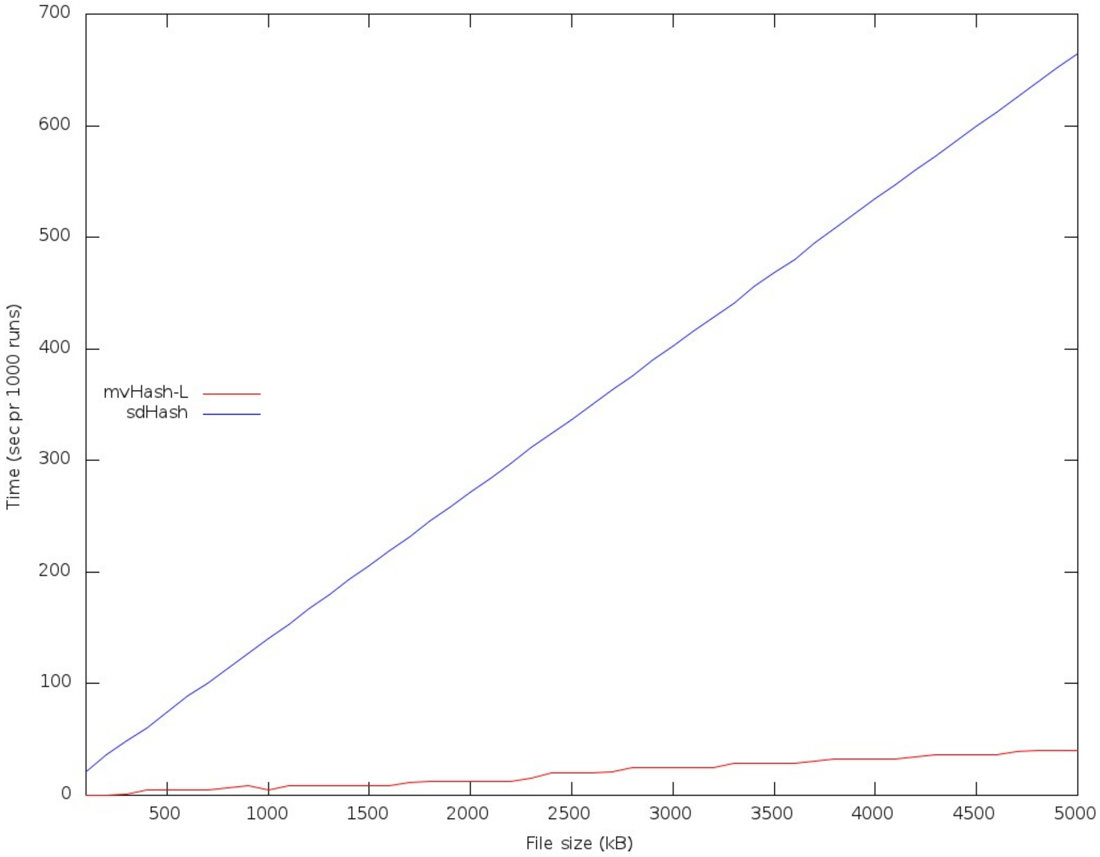


Figure 8: Computation time for mvHash-L and sdhash.

hash has its own file. Each time two hashes are compared, two files must be opened and closed. Thus, mvHash-L must perform much more file operations to do an all-against-all comparison, and file operations are not really a part of the comparison algorithm. On the other hand, the difference between the comparison times is very significant, and file operations are probably not the main reason to this big difference.

To understand under which circumstances mvHash-L is slow, it will be a short test to investigate how the comparison time of mvHash-L is affected by the size of the hash. To do this test, it was generated some random files of specified size. These files were not hashed, but they were considered as a hash themselves. It works to consider them a hash themselves, as a hash in mvHash-L may seem like just a random byte string. The time mvHash-L used when these hashes were compared against each other was measured and the results plotted in table 7.

A short explanation of the table follows, the two first columns are the size of the hashes and the third column is the execution time of mvHash-L when these hashes were compared. In column number four is the execution time divided by the multiple of the hash sizes. As the values in the fourth column are relative similar, it means the comparison time increases proportionally to the multiple of the hash sizes. Thus, the comparison

Hash A (kB)	Hash B (kB)	Comparison time	$\frac{\text{Comparison time}}{\text{Hash A (kB) * Hash B (kB)}}$
10	1	0.04	0.0040
10	5	0.18	0.0036
10	10	0.32	0.0032
10	20	0.74	0.0037
10	30	1.20	0.0040
20	20	1.28	0.0032
30	30	2.92	0.0032

Table 7: Comparison time for some hash sizes using mvHash-L.

will be slow when the hash size is large.

Summary

MvHash-L is able to detect similar jpg-files. The hash size is short compared to sdhash and mvHash-L generates its hashes very fast, way much faster than sdhash. On the other hand, sdhash has a larger edit operation ratio and compares its hashes way much faster than mvHash-L. Very slow comparison time is a significant disadvantage and the main disadvantage of mvHash-L for jpg-files.

6.2.2 Evaluation for doc-files

The evaluation will be performed using four steps: First, an initial test, second, it is tested if mvHash-L for doc-files is able to detect similar files, third, a threshold and edit operations ratio is calculated, and fourth, a summary.

Initial tests

Using a neighbourhood of 20 or 50 seem suitable as the hash size is below 2% and the hash size is not too short [3]. Using a wider neighbourhood, to get even smaller hashes, doesn't seem as an advantage as it will make very small hashes with very little granularity.

It was performed an initial test by using the configuration below. The initial test was performed by using a small part of the c-corpus.

Neighbourhood:	20, 50.
RLE-width:	1B.
Modulus:	256.
Majority vote limit:	8, default.

For this configuration, for both neighbourhoods, there occurred very many non-similar file pairs with high scores (false positive), making it impossible to distinguish between similar and non-similar files and this configuration is not suitable.

A byte has values from 0 to 255, in a specific file type, probably some of these values are more common than others. In a doc-file are particularly the values for the most common letters in lower case frequently used. Which bytes that are most frequently used may affect which majority vote limit which is most suitable. To find a more suitable configuration for doc-files, we experimented by reducing the majority vote limit from 8 to 7. Majority vote limit of 7 means, that if at least 43.75% of the bits in the neighbourhood is 1, then the output byte is set to 0xFF. The configuration is therefore changed to the following:

[3] See table 18 in appendix A Tables

Neighbourhood:	20, 50.
RLE-width:	1B.
Modulus:	256.
Majority vote limit:	7.

The hash size for this configuration was calculated and it is 2.00% and 0.91% for the neighbourhoods 20 and 50, respectively. An initial test using this configuration was promising. In the subsequent tests, this configuration will be used.

Similarity detection

The purpose of this paragraph is to test whether mvHash-L for doc-files is able to detect similar files. The alignment robustness and non-propagation tests were performed and showed that the selected configuration of mvHash-L has the alignment robustness and non-propagation properties.

The general similarity detection test was performed, and the results are presented in table 8. As shown in this table, the fewer edit operations, the higher score, and mvHash-L has ability to detect similar files doc-files.

File: RLResume.doc, 122,9 kB.		
	Neigbhbourhood 50	Neighbourhood 20
Maximum number of edit operations	Lowest score	Lowest score
50	94	98
100	91	97
200	87	92
300	85	91

Table 8: The general similarity detection test for mvHash-L for doc-files.

Threshold and edit operations ratio

In this paragraph, it is identified a threshold and the edit operations ratio is calculated.

It was performed an all-against-all comparison of the doc-files in the c-corpus and the results are summarized in the table 9. In the all-against-all comparison, for every score up 59, there are lots of file pairs for each value. As most files assumable are non-similar, all scores of 59 or lower are considered non-similar. For pairs which got a score of 60 or higher, it is evaluated whether or not the files are similar, in order to identify suitable threshold. Studying the table, it is not possible to find a threshold with no false positive results, but there may be possible to find a threshold with few false positive. A threshold with few false positive may be between 60 and 90. The all-against-all test was also performed at the t5-corpus and the results are shown in table 10. This test confirms the test results from the c-corpus that a suitable threshold is between 60 and 90.

The edit operation ratio for some values between 60 and 90 was calculated and the results summarized in table 11.

Table 11 shows that a neighbourhood of 50 requires a lower amount of edit operations than a neighbourhood of 20, before the score is below a specific threshold. Inspection of this phenomenon discloses the reason. When an edit operation is performed on a file and the file is hashed using a neighbourhood of 50, it has a potential of altering 50 bytes in the output of majority vote. When the neighbourhood is 20, the potential is to alter 20

	Neighbourhood: 50		Neighbourhood: 20	
Score	Non-similar	Similar	Non-similar	Similar
90-100:	0	5	0	5
80-89:	0	0	0	0
70-79:	2	0	1	3
60-69:	2	7	17	3
0-59:	Ca 2 million	0	Ca 2 million	0

Table 9: The result of an all-against-all comparison of the doc-files in the c-corpus.

	Neighbourhood 50		Neighbourhood 20	
Score	Non-similar	Similar	Non-similar	Similar
90-100:	2	2	2	2
80-89:	0	0	2	2
70-79:	3	1	2	5
60-69:	1	3	3	1
0-59:	Ca 140.000	0	Ca 140.000	0

Table 10: The result of an all-against-all comparison of the doc-files in the t5-corpus.

bytes. The more bytes which are altered in the output of majority vote, the more changes in the RLE encoded string and the RLE encoded string is the hash.

Neighbourhood 50				
Threshold	90	80	70	60
Edit operation ratio	0.17%	0.36%	0.60%	0.93%
Neighbourhood 20				
Threshold	90	80	70	60
Edit operation ratio	0.35%	0.77%	1.30%	2.00%

Table 11: Edit operations ratio for different thresholds for doc-files in the c-corpus using mvHash-L.

The time to generate and compare hashes was measured, the results were similar to the test of mvHash-L for jpg-files.

Summary

Using a majority vote limit of 7 and a neighbourhood of 20 or 50, mvHash-L is able to detect similar files. It generates hash fast and the hash size is short. The main disadvantages are that for all potential thresholds, some few false positive results may be expected, and the comparison time is very slow.

6.3 Evaluation of mvHash-B

In this section mvHash-B will be evaluated. This section has three subsections: First, an initial analysis, second and third, mvHash-B is evaluated for jpg- and doc-files.

6.3.1 Initial analysis

In mvHash-B, the RLE encoded string is stored using Bloom filters. To make entries for the Bloom filter, groups of 11 elements are picked from the RLE encoded string, a new group is picked for every 2nd element which results in very many entries. The size of the hash of mvHash-B depends on the number of Bloom filters, the number of Bloom filters depends on the number of entries and the maximum number of entries per Bloom filter. As very many entries are expected, to reduce the size of the hash, it's favourable to use a high number of entries per Bloom filter. In the subsequent evaluation of mvHash-B for jpg-files, there will be a discussion and an experiment to identify a suitable number of entries per Bloom filter.

Moreover, note that 11 elements of the RLE encoded string are used to make one entry in the Bloom filter. When testing mvHash-B, it should be used a neighbourhood which for all files in the test corpus makes an RLE encoded string of at least 11 elements.

It may seem contradictory to first discuss the case of many entries per Bloom filter and then discuss how to get at least 11 elements in the RLE encoded string. The explanation is that even the selected neighbourhood makes the average RLE encoded string long, some few files may get a really short string, as majority vote create very long runs, due to the distribution of 0s and 1s within the file.

Considering the configuration of mvHash-B. In mvHash-L, the RLE-width and modulus are configurable parameters, while these are not configurable parameters in mvHash-B. As in mvHash-B, the elements of the RLE encoded string is hashed using modulus 2 to create entries for the Bloom filter.

6.3.2 Evaluation for jpg-files

The evaluation is performed using four steps: First, a discussion considering neighbourhood size, second, a discussion considering the number of entries per Bloom filter. Third, mvHash-B for jpg-files is compared to other algorithms and fourth, a summary.

Neighbourhood size

The purpose of this paragraph is to identify a neighbourhood for mvHash-B for jpg-files. When mvHash-L was tested for jpg-files, it was disclosed that using a neighbourhood of 200 bytes, the hash of mvHash-L (the RLE encoded string) for some files is really short. MvHash-B requires at least 11 elements in the RLE encoded string to make an entry in the Bloom filter. By using neighbourhood of 200, 100 and 50 bytes at the c-corpus, 50 was the largest neighbourhood where the RLE encoded string was long enough for all jpg-files in the c-corpus. From mvHash-L it's known that a neighbourhood of 200 has high enough granularity to distinguish between similar and non-similar jpg-files. As shorter neighbourhood increases the granularity, while the neighbourhood is shortened to 50, granularity is not expected to be a case and it's not necessary to shorten the neighbourhood even more.

Entries per Bloom filter

The purpose of this paragraph is to find the most suitable number of entries per Bloom filter and then the most suitable configuration of mvHash-B.

To avoid a long hash, it is desired to use a high number of entries per Bloom filter. Two or more entries may have the same value and then set the same bit in the Bloom filter (a collision). When this happen, the second time an entry sets a bit, the Bloom filter is not altered and the second entry doesn't affect the Bloom filter and not the hash. The probability that an entry doesn't affect the Bloom filter increases by the number of entries in the Bloom filter.

On the other hand, as a new entry starts for every second element and each entry covers 11 elements, each element in the RLE encoded string affects 5 or 6 entries. Even there are lots of collisions, each element have a great probability of affecting the Bloom filter. Another general problem of many entries per Bloom filters is that the filters become really similar, making the difference between the Bloom filters really small. MvHash-B will be tested using an experimental approach. The following configuration will be used:

Neighbourhood:	50
Majority vote limit:	8.
Entries per Bloom filter:	512, 1024, 2048, 4096

This configuration means that four different configurations of mvHash-B will be tested, the difference between the configurations is the number of entries per Bloom filter. The specified numbers of entries per Bloom filter is chosen to test different high values. Our Bloom filter has 2048 bits, studying the configuration above, there are up to twice as many entries as there are bits to set in the Bloom filter. Using the following equation [26], it may be determined how many percent of the bits which are set in the Bloom filter after n entries, if the values of the entries are randomly distributed:

$$1 - \left(1 - \frac{1}{2048}\right)^{n} \tag{6.1}$$

A computation shows that the proportion of bits set are 22%, 39%, 63% and 86% for 512, 1024, 2048, and 4096 entries, respectively. For some of these configurations, the Bloom filters are really full and there are lots of collisions. The hypothesis is that even really many bits are set in the Bloom filter, each file will make their unique pattern in the Bloom filters.

First, we performed some tests which showed that all four configurations have the alignment robustness and non-propagation properties. Then, the general similarity detection test was performed and all four configurations are able to detect similar files [4].

To identify the most suitable configuration, for each of the different configurations (different number of entries), several tests were performed and the results are written in table 12. Below is a description of each line of the table:

Threshold: The threshold was calculated by doing an all-against-all comparison using the jpg-files in the c-corpus. For all 4 configurations, it was possible to find a well-defined threshold. Well-defined means that for each value up to the threshold, there were lots of non-similar files for each score, but none non-similar files got a score equal to or above the threshold.

Edit operations ratio: By using the threshold, the edit operations ratio was calculated. It was also calculated a value named edit operations absolute. The average size of the jpg-files in the c-corpus is 251B. The edit operation absolute was calculated by multiplying the size of the average file size in bytes with the edit operations ratio. Thus, the absolute-value gives a picture of how much an average file may be edited in the number of bytes, and still be detected as similar.

Hash size: A calculation of the average hash size.

[4] The results of the general similarity detection test are written in table 19 in appendix A Tables

Computation time: The computation time was measured by using the reference test for computation time.

Comparison time: The comparison time was measured by using the reference test for comparison time.

Number of entries pr Bloom filter	**512**	**1024**	**2048**	**4096**
Threshold	51	55	69	85
Edit operations ratio	0.53%	0.67%	0.50%	0.22%
Edit operations absolute	1365	1725	1288	567
Hash size	1.26%	0.78%	0.59%	0.52%
Computation time (s)	1.48	1.42	1.36	1.34
Comparison time (s)	495.5	490.5	486.7	470.7

Table 12: Statistic of the mvHash-B approach for jpg-files.

Analysing table 12, a result which seems surprising is the edit operations ratio. It is the only line where the results not are linearly increasing or decreasing. If there are more entries per Bloom filter, a higher amount of edit operations is required to go below a specific score. Comparing the edit operations ratio for 512 and 1024, it's required more edit operations to go below threshold 55 using 1024 entries, than below threshold 51 using 512 entries. Another interesting aspect to review in this table is the hash size. When the entries per Bloom filter increase from 2048 to 4096, the hash size only slightly decrease. Many files are hashed to just one Bloom filter using 2048 entries per Bloom filter and it's therefore not possible to make them shorter by increasing the number of entries.

In this paragraph, 4 different configurations are tested. The most suitable configuration may depend on the situation. In the following discussion a configuration which seems to suit generally will be identified.

With respect to computation and comparison time, it is very little difference between the configurations. The higher number of entries per Bloom filter, the shorter time, but the shortest time is only 10% and 5% faster than the slowest one for computation and comparison time, respectively. On the other hand, the edit operations ratio and the hash size differ significantly between the configurations.

The configuration of 2048 entries has a very short hash, and the edit operation ratio is 0.50% which is acceptable, this configuration is therefore chosen as the most suitable. For this configuration, it was performed an all-against-all comparison by using the jpg-files in the t5-corpus and this test confirmed the threshold from the calculation which used the c-corpus.

Comparison to other algorithms

In this paragraph, the chosen configuration of mvHash-B is compared to other algorithms. The computation time of mvHash-B was compared to SHA-1 using the test "computation time - graph presentation" and the results may be studied in figure 9, mvHash-B generates its hashes almost as fast as SHA-1. Note, mvHash-L was in a previous section measured to be slightly faster than SHA-1, while mvHash-B is slightly slower, thus, mvHash-B is slightly slower than mvHash-L. This is due to the fact that mvHash-B uses one more phase than mvHash-L to generate a hash.

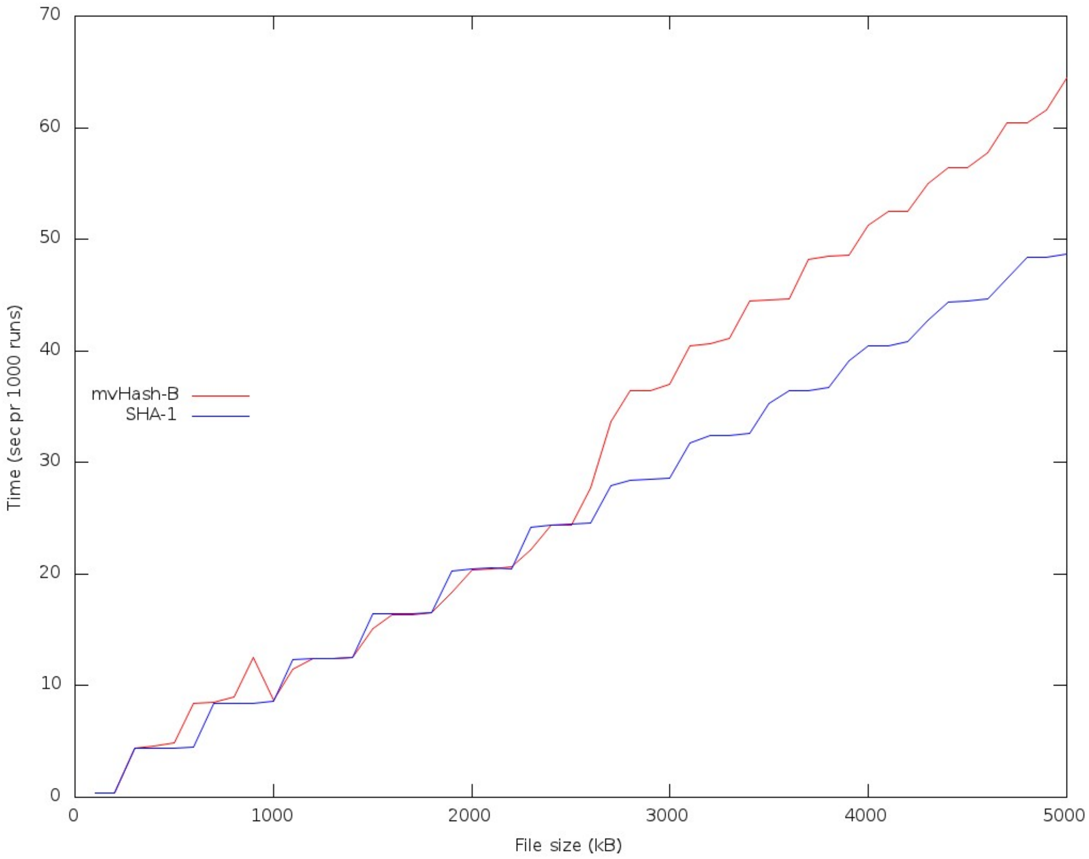


Figure 9: Computation time for mvHash-B and SHA-1.

It is also important to compare mvHash-B against sdhash, it is used the same tests as when different configurations of mvHash-B was tested against each other in previous paragraph. The results of the comparison are shown in table 13.

Studying the table, MvHash-B has significantly lower computation time and shorter hash size than sdhash, while sdhash has larger edit operation ratio. Considering the comparison time, mvHash-B has comparison time of 486.7 seconds and the corresponding value for sdhash is 34.92 seconds. As described in the evaluation of mvHash-L for jpg-files, this comparison to sdhash is not totally fair. It is reasonable to expect that mvHash-B has a potential of doing the comparison as fast as sdhash, due to the fact that both are storing their hashes using Bloom filter and uses Hamming distance to perform comparison.

Summary

MvHash-B for jpg-files generates its hashes faster and has shorter hash size than sdhash, while sdhash is able to detect files which are more changed, as the edit operation ratio is higher. Considering the comparison time, mvHash-B has a potential of performing the comparison as fast as sdhash.

	sdhash	mvHash-B
Computation time (s)	22.13	1.36
Comparison time (s)	34.92	486.7
Hash size	2.60%	0.59%
Threshold	21	69
Edit operations ratio	0.92%	0.50%

Table 13: Evaluation of mvHash-B vs sdhash for jpg-files.

6.3.3 Evaluation for doc-files

When mvHash-L was tested for doc-files, it was tested for neighbourhood of 20 and 50 using a majority vote limit of 7. Initial tests showed that a neighbourhood of 50 generated RLE encoded string shorter than 11 elements for some files in the c-corpus and is not suitable, but a neighbourhood of 20 was suitable.

While testing mvHash-B at jpg-files, 2048 entries per Bloom filter seemed to work and it will be used for doc-files as well. The configuration which will be used is summarized in the following:

Neighbourhood:	20.
Majority vote limit:	7.
Entries per Bloom filter:	2048.

For this configuration, the average hash size was 0.47%. First the alignment robustness and non-propagation tests were performed for this configuration and this configuration has the alignment robustness and non-propagation properties. Then, the general similarity detection test was performed and the results are presented in table 14. Studying this table, mvHash-B has ability to detect similar files.

To identify a threshold, it was performed an all-against-all comparison of the doc-files in the c-corpus and the results are shown in table 15. Studying the results, it's not possible to find a threshold with no false positive, but a suitable threshold with few false positive results is 70. To test whether another corpus gives the same results, the all-against-all comparison was performed by using the doc-files from the t5-corpus and the results are shown in table 16, this table verifies that 70 may be a suitable threshold.

The edit operation ratio for threshold 70 is 0.51%. The computation and comparison time is expected to be similar for both doc- and jpg-files and the performance for doc-files is not measured.

Summarized, mvHash-B for doc-files is fast, makes very short hashes, and have an acceptable edit operation ratio. The only disadvantage is a small number of false positive results.

File: RLResume.doc.	
File size: 122,9kB.	
Hash size: 1,1kB.	
Maximum number of edit operations	All scores above
10	96
50	92
100	87
150	78

Table 14: The general similarity detection test for mvHash-B for doc-files.

Score	Non-similar	Similar
90-100	0	5
80-89	0	5
70-79	8	3
0-69:	Ca 2 million	0

Table 15: Test results of the all-against-all comparison of the doc-files in the c-corpus.

Score	Non-similar	Similar
90-100	2	2
80-89	3	1
70-79	6	5
0-69	Ca 140.000	0

Table 16: Test results of the all-against-all comparison of the doc-files in the t5-corpus.

7 Discussion

MvHash-L and mvHash-B are tested and work for jpg- and doc-files. A test of how well it works for other file types was not performed, but this aspect is important to discuss. When configuring mvHash-L or mvHash-B for a new file type, it is the majority vote limit and neighbourhood size which must be configured. Which configuration that fits best depends on the distribution of bits of 0s and 1s within the file. Due to the configuration options, it is possible to adjust the configuration for different distributions of 0s and 1s and it is reasonable to believe that for each file type, it is possible to identify a configuration which fits.

A relevant question is why the RLE and the number of entries per Bloom filter not should be considered when configuring the algorithm for a specific file type. RLE and Bloom filter are only used to compress and store the output of majority vote. How the output of majority vote is compressed and stored is a question which not depends on the file type.

An important aspect to discuss is the evaluation methodology and particularly the way used to measure how well the fuzzy hashing algorithm is able to detect similar files. It was measured by first identifying a threshold and then calculating the edit operation ratio, the larger ratio, the better. As stated in the chapter Basics of testing and evaluation, a graph with a calculation of false match rate vs false non-match rate for different thresholds would have been a better way to measure the algorithm, but a test set where all file pairs are classified as either similar or non-similar does not exist. Even the method in this thesis is not perfect, it is no doubt that the method gives a good evaluation of how well the algorithm is able to detect similar files.

The concept of majority vote is to transform a long string into a sequence of two states, it was chosen to use the bytes 0x00 and 0xFF to represent these two states. Another way to represent the states would simply have been to represent them by using one bit, such that the states were 0 and 1, then the output of majority vote had been reduced by a factor of 8. Even this may seem as a huge improvement, it wouldn't change anything in the implementation, because in the implementation the output string from majority vote is never stored. After each calculation of an output byte of majority vote, the RLE encoded string is immediately updated.

When doing a comparison, mvHash maps the result to a scale from 0-100 (from -1 for mvHash-B). For all configurations tested in this thesis, it is identified a threshold, e.g. for the chosen configuration of mvHash-B for jpg-files, the threshold is 69. Thus, if two files get a score below 69, they are non-similar. Using a scale from 0-100, it may be confusing that 68 or lower means non-similar, therefore a re-alignment would have been a great advantage. The re-alignment could work such that all non-similar scores, 68 or below, are mapped to 0 and the scores which indicate that the files are similar, 69-100, are mapped from 1-100. It would make the scale less confusing.

In some files, it is possible to do small changes in the input which not changes the output of majority vote for any neighbourhood and then, the hash will not be altered. If such a change is performed, using mvHash-L or mvHash-B, the altered file will get a score of 100 when compared to the original file. It may be considered a weakness that a score of 100 only means very similar, not identical.

8 Conclusion

In this master thesis, a new fuzzy hashing algorithm, named mvHash, has been analysed, developed and tested. Due to the lack of a renowned evaluation methodology for fuzzy hashing algorithms, there has in this thesis been established such a methodology, and this methodology has been used to test mvHash.

Two versions of mvHash were developed based on the same basic structure, the difference between the versions is how they represent their hash and perform comparison. One version is using Levenshtein distance to compare the hashes (mvHash-L) and one version is using Bloom filters and Hamming distance (mvHash-B). To compare hashes using Levenshtein distance is very time consuming, comparing hashes using Hamming distance is very fast as only some few low-level operations are required. Due to the very large difference in comparison time, mvHash-B was the most successful one.

An advantage of mvHash-B is the computation time, the time used to generate a hash. MvHash-B is almost as fast as the cryptographic hash function SHA-1 which is known to be very fast. This is way much faster than the leading fuzzy hashing algorithm sdhash. Another advantage of mvHash-B is the hash size, the hash size is very short compared to sdhash.

MvHash-B has some configuration options and each file type require its own configuration, no standard configuration works for all file types. In this thesis, mvHash-B is tested for jpg- and doc-files. For jpg-files mvHash-B seems to work very well, it is able to distinguish between similar and non-similar files. Considering doc-files, mvHash-B works, but not as excellent as for jpg-files and some few false positive results may be expected.

9 Future work

MvHash-B is tested using an experimental approach without a comprehensive theoretical analysis. Bloom filters work best if the input is random. In future work it may be calculated the extent to which the entries in the Bloom filters are random and based on this calculation, mvHash-B may be improved.

If a large part of a file is removed, then the remaining part is called a fragment. Fragment detection is to detect that the fragment is similar to a part of the original file. In future work, there may be performed a study of mvHash' ability to perform fragment detection.

In mvHash-B, the most suitable configuration used 2048 entries per Bloom filter. Consider a large file, its hash consists of several Bloom filters. If the start of the file is cut, such that the 1024 first entries are removed. Then, each Bloom filter will consist of 50% of each of the old entries and 50% of the entries which originally was in the subsequent Bloom filter. Thus, it is expected that the original and the modified file will get a low score when compared to each other. This is a special case of fragment detection and may be studied in future work.

As it is no standard configuration for mvHash-L and mvHash-B, a suitable configuration must be identified for each file type. Due to the huge amount of file types, it is not feasible to manually identify a configuration for each file type. In future work it may be developed a method to automatically detect the most suitable configuration for each file type.

To make a graph of false match rate vs false non-match rate (detection error tradeoff curve), a test corpus where all files pairs are classified as similar or non-similar is required. In further work, such a test corpus may be made and mvHash and other fuzzy hashing algorithms may be evaluated using this test corpus.

A Tables

This chapter includes tables which are referred from different chapters in the thesis. These tables are not important to understand the thesis, but are included for readers who want to study all the details.

Configuration:				
Majority vote threshold:	8, default.			
Neighbourhood: 200B				
File type	**JPG**	**DOC**	**PDF**	**TXT**
Median	4	4	4	7
Upper quartile	19	27	27	59
Maximum	65535	65535	65535	65535
Average	73.40	172.39	172.39	1886.59
Neighbourhood: 100B				
File type	**JPG**	**DOC**	**PDF**	**TXT**
Median	4	4	4	6
Upper quartile	19	24	23	54
Maximum	65535	65535	65535	65535
Average	41.91	96.69	47.15	915.62
Neighbourhood: 50B				
File type	**JPG**	**DOC**	**PDF**	**TXT**
Median	4	4	4	7
Upper quartile	18	23	21	60
Maximum	65535	65535	65535	65535
Average	24.97	55.88	29.49	275.39
Neighbourhood: 20B				
File type	**JPG**	**DOC**	**PDF**	**TXT**
Median	4	4		4
Upper quartile	17	19		30
Maximum	65535	65535		65535
Average	13.75	26.86		37.41

Table 17: Distribution of run-lengths for different neighbourhoods and file types.

Configuration:				
RLE-width:	1B			
Modulus:	256			
Majority vote threshold:	8, default			
Neighbourhood: 200B				
	JPG	**DOC**	**PDF**	**TXT**
	1.45%	0.22%	1.43%	0.05%
Neighbourhood: 100B				
	JPG	**DOC**	**PDF**	**TXT**
	2.47%	0.40%	2.27%	0.10%
Neighbourhood: 50B				
	JPG	**DOC**	**PDF**	**TXT**
	4.05%	0.75%	3.60%	0.36%
Neighbourhood: 20B				
	JPG	**DOC**	**PDF**	**TXT**
	7.27%	1.61%	6.51%	2.83%

Table 18: Average hash size for different configurations of mvHash-L.

Configuration	
Neighbourhood: 50	
Majority vote limit: 8	
Modulus: 256	
File name: book-and-pen.jpg	
File size: 110kB	
Entries per Bloom filter: 512	
Maximum number of edit operations	**Lowest score**
50	92
100	87
150	82
200	75
Entries per Bloom filter: 1024	
Maximum number of edit operations	**Lowest score**
50	94
100	89
150	85
200	79
Entries per Bloom filter: 2048	
Maximum number of edit operations	**Lowest score**
50	95
100	89
150	86
200	80
Entries per Bloom filter: 4096	
Maximum number of edit operations	**Lowest score**
50	96
100	94
150	92
200	89

Table 19: The results of the general similarity detection test for jpg-files using mvHash-B.

B Prototype of mvHash

The purpose of this section is to describe how to use the prototype of mvHash, such that interested readers may verify the results which are claimed in this thesis.

A modular code is normally a great advantage, but during the implementation of mvHash it turned out that modularity and fast execution is contradictory. To make the execution fast, the code is written by using code optimization techniques and the code is not that modular.

The prototype generates and compares hashes of mvHash-L and mvHash-B. Note, when generating a hash of mvHash-L, it uses always RLE-width 1B and modulus 256, as it in the evaluation chapter was decided to always use this configuration. Thus, the RLE-width and modulus are not configurable parameters.

This chapter has two sections: First section describes how to generate a hash using the prototype, second section describes how to compare two hashes.

B.1 Generate a hash

Following flags must be specified to generate a hash:

-i: Input file.

-o: Output file.

-w: Neighbourhood size in bytes.

-t: Majority vote limit.

-e: Maximum number of entries per Bloom filter. By specifying a positive value, it is generated hash of mvHash-B with the specified numbers of entries per Bloom filter, by specifying 0, it is generated hash of mvHash-L.

Examples

Using the command below, mvHash-L will hash input_file to output_file using a neighbourhood of 50 and majority vote limit 8.

- mvHash -w 50 -t 8 -e 0 -i input_file -o output_file.

Using the command below, mvHash-B will hash input_file to output_file using a neighbourhood of 50, majority vote limit 8 and 2048 elements per Bloom filter.

- mvHash -w 50 -t 8 -e 2048 -i input_file -o output_file.

B.2 Compare hashes

To compare two hashes, the flags below must be specified. The result of the comparison will be printed in the terminal.

-i: File with hash 1.

-m: File with hash 2.

-e: Entries per Bloom filter, by specifying 0 the hashes are compared as hashes of mvHash-L. By specifying the number of Entries per Bloom filter (which was used when the hashes was created), the files are compared as hashes of mvHash-B.

Examples

Using the command below, two hashes of mvHash-L is compared.

- mvHash -e 0 -i hash_1 -m hash_2

Using the command below, two hashes of mvHash-B is compared.

- mvHash -e 2048 -i hash_1 -m hash_2

Bibliography

[1] Kornblum, J. 2006. Fuzzy hashing. http://jessekornblum.com/presentations/htcia06.pdf. Presentation. Last visited: June 2012.

[2] Hurlbut, D. 2009. Fuzzy hashing for digital forensic investigators. http://accessdata.com/downloads/media/Fuzzy_Hashing_for_Investigators.pdf. Access data. Last visited: June 2012.

[3] Chawathe, S. june 2009. Effective whitelisting for filesystem forensics. In *Intelligence and Security Informatics, 2009. ISI '09. IEEE International Conference on*, 131 –136.

[4] Roussev, V. 2011. An evaluation of forensic similarity hashes. *Digital Investigation*, 8(Supplement 1), 34 – 41. The Proceedings of the Eleventh Annual DFRWS Conference.

[5] Kornblum, J. 2006. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3(Supplement 1), 91 – 97. The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06).

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

[7] Laganà, A. 2004. *Lecture notes in computer science*. Springer.

[8] Weng, L. & Preneel, B. 2011. Image distortion estimation by hash comparison. In *Proceedings of the 17th international conference on Advances in multimedia modeling - Volume Part I*, MMM'11, 62–72, Berlin, Heidelberg. Springer-Verlag.

[9] Uzuner, O. & Davis, R. 2003. Content and expression-based copy recognition for intellectual property protection. In *Proceedings of the 3rd ACM workshop on Digital rights management*, DRM '03, 103 – 110, New York, NY, USA. ACM.

[10] Garfinkel, S. 2007. Anti-forensics: Techniques, detection and countermeasures. *Proceedings of the 2nd International Conference on Information Warfare & Security*, 77–84.

[11] Kessler, G. C. 2007. Anti-forensics and the digital investigator. *Proceedings of The 5th Australian Digital Forensics Conference*, 1–7.

[12] Roussev, V. 2010. Data fingerprinting with similarity digests. In *Advances in Digital Forensics VI*, Chow, K.-P. & Shenoi, S., eds, volume 337 of *IFIP Advances in Information and Communication Technology*, 207–226. Springer Boston.

[13] DigitalNinja. 2007. Fuzzy clarity: Using fuzzy hashing techniques to identify malicious code. rev 1. http://digitalninjitsu.com/sites/default/files/Fuzzy_Clarity_rev1.pdf. Last visitied: June 2012.

[14] Breitinger, F. 2011. Fuzzy hashing: Similarity preserving hash function. Presentation.

[15] Breitinger, F. Sicherheitsaspekte von fuzzy-hashing. Master's thesis, Hochschule Darmstadt, 2011.

[16] Tridgell, A. Spamsum readme. http://www.samba.org/ftp/unpacked/junkcode/spamsum/README. Last visited: June 2012.

[17] Harbour, N. Dfcldd. http://dcfldd.sourceforge.net. Last visited: June 2012.

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

[19] Menezes, A., Oorschot, P., & Vanstone, S. 1997. *Handbook of applied cryptography*. CRC Press.

[20] Delfs, H. & Knebl, H. 2007. *Introduction to cryptography: principles and applications*. Springer.

[21] Masek, W. J. & Paterson, M. S. 1980. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1), 18 – 31.

[22] Ziroff, G. 2012. Approaches for similarity-preserving hashing. Bachelor thesis.

[23] Lewenstein, W. I. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10, 707.

[24] Bloom, B. H. July 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), 422–426.

[25] Roussev, V. 2012. http://roussev.net/sdhash/sdhash.html. sdhash Home. Last visited: June 2012.

[26] Cao, P. 1998. Bloom filters - the math. http://pages.cs.wisc.edu/ cao/papers/summary-cache/node8.html. Last visited: June 2012.