Design and Analysis of Approximate Matching Algorithms

By
Monika Singh

Under the Supervision of
Dr. Donghoon Chang
Dr. Somitra Kumar Sanadhya

Indraprastha Institute of Information Technology Delhi
February, 2022

Design and Analysis of Approximate Matching Algorithms

By
Monika Singh

Submitted
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

to the

Indraprastha Institute of Information Technology Delhi
February, 2022

Certificate

This is to certify that the thesis titled - **“Design and Analysis of Approximate Matching Algorithms”** being submitted by **Monika Singh** to Indraprastha Institute of Information Technology, Delhi, for the award of the degree of Doctor of Philosophy, is an original research work carried out by her under our supervision. In our opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree/diploma.

Dr. Donghoon Chang

February, 2022
Department of Computer Science
IIIT Delhi
New Delhi, 110020

Dr. Somitra Kumar Sanadhya

February, 2022
School of Artificial Intelligence
and Data Science (AIDE)
IIT Jodhpur
Rajasthan, 342037

To my parents,
for their unwavering support and belief in me.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisors Dr. Donghoon Chang and Dr. Somitra Kumar Sanadhya, for their constant support and encouragement at every step of the way throughout my Ph.D. journey. They have guided me towards many opportunities and have helped and encouraged me to pursue them. Their invaluable guidance not only has improved my research but also had a huge positive impact on my life on every front. I consider myself incredibly fortunate to have them as my mentors. Without both of them, the journey wouldn't have been the same.

I would also like to convey my sincere gratitude to Barbara Guttman and Douglas White for providing me the opportunity to collaborate with them at NIST. I am eternally grateful for their guidance, encouragement, and insightful suggestions. I would also like to thank my supervisor Douglas Montgomery for giving me the flexibility to work on my Ph.D. while working full time.

I would also like to take this opportunity to thank all my dear friends from IIIT-Delhi and NIST. I am very grateful to my friend Mohona and Robin for the insightful discussion and fruitful collaborations on various projects. I would like to thank my friends Sweta, Suvarna, Shuvo, Megha, Surabhi, Sneihil, Jyoti, and Tarun for making the experience enjoyable with their friendly interactions, various technical & non-technical discussions, and just by being wonderful people.

I would also like to thank Tata Consultancy Services (TCS), India, for awarding me the prestigious TCS Fellowship for my entire Ph.D. period.

Finally, and most importantly, I would like to thank my parents, sisters, and brother for their unconditional love, care, encouragement, and immense support. I cannot thank them enough for giving me the privilege to choose, the confidence to decide, and the comfort of always having my back. This thesis is dedicated to them. This acknowledgment wouldn't be complete without the mention of my US family, Nick, Mary, and Leah. Without their tremendous support, understanding, and motivation, it wouldn't have been possible.

Above all, I am grateful to God, who granted me countless opportunities, blessings, wisdom, strength, and the courage to pursue my dreams and goals.

List of Publications

The authors are listed in alphabetical order by their last names.

1. Donghoon Chang, Mohona Gosh, Somitra Kumar Sanadhya, Monika Singh✉ and Douglas R. White. *FbHash: A New Similarity Hashing Scheme for Digital Forensics.* In Digital Investigation, DFRWS 2019 USA - Proceedings of the Nineteenth Annual DFRWS USA, Portland, OR, July 14-17, 2019.[Journal]

2. Donghoon Chang, Somitra Kumar Sanadhya, Monika Singh✉, *Security Analysis of mvHash-B Similarity Hashing*, In International Conference on Digital Forensics & Cyber Crime(ICDF2C) 2016, New York, U.S.,[Journal] September 28-30 2016.

3. Donghoon Chang, Somitra Kumar Sanadhya, Monika Singh✉ and Robin Verma. *A Collision Attack on sdhash Similarity Hashing.* In Systematic Approaches to Digital Forensic Engineering(SADFE) 2015 - 10th International Conference, Málaga, Spain, September 30 – October 2, 2015.

4. Donghoon Chang, Somitra Kumar Sanadhya and Monika Singh✉. *A Second Preimage and Collision Attack on sdhash Similarity Hashing.* In IFIP WG 11.9 - 12th International Conference , New Delhi, India, January 4-6,2016.[Poster]

List of Submitted Papers

1. Donghoon Chang, Mohona Gosh, Anviksha Khuneta, Somitra Kumar Sanadhya and Monika Singh✉. *FbHash-E:A Time and Memory efficient Version of FbHash Similarity Hashing Algorithm.* In Forensic Science International: Digital Investigation, September, 2021.

2. Donghoon Chang, Somitra Kumar Sanadhya, Monika Singh✉ and Douglas R. White. *Approximate matching evaluation tool.* In NIST Interagency Report (NISTIR), August 2021.

3. Monika Singh✉. *Essential Characteristics of Approximate Matching Algorithms.* In IJCSDF, June, 2021.

List of Published Open-source Tools

1. Donghoon Chang, Mohona Gosh, Somitra Kumar Sanadhya, Monika Singh✉ and Douglas R. White. FbHash: A New Similarity Hashing Scheme for Digital Forensics. https://github.com/MonikaSinghGit/FbHash

2. Monika Singh and Douglas R. White. Approximate matching evaluation tool. https://github.com/MonikaSinghNIST/AMtoolTestingFramework

Other Publications

1. Paul E. Black, and Monika Singh. Opaque Wrappers and Patching: Negative Results. IEEE Computer (Volume: 52, Issue: 12), December, 2019.

2. Monika Singh, and Mudumbai Ranganathan. Formal Verification of Bootstrapping Remote Secure Key Infrastructures (BRSKI) Protocol Using AVISPA. NIST TN, October, 2020.

Abstract

With the rapid growth of the World Wide Web and the Internet of Things, huge amounts of digital data are being produced every day. Digital forensics investigators face an uphill battle when they have to manually screen through and examine such humongous data during an investigation. A major requirement of modern forensic investigation is to perform automatic filtering of correlated data, thereby reducing and focusing the manual effort of the investigator. There are two types of filtering: *blacklisting* and *whitelisting*. Blacklisting is the process of filtering data by matching them with the set of known-to-be-bad files (as determined by the investigator). The resultant files after this process are the ones which an investigator needs to examine closely. On the other hand, whitelisting is the process of filtering by matching the files with a set of already known-to-be-good files. The files passing this process need not be examined by the investigator.

Approximate matching algorithm, also known as *Similarity hashing*, is a generic term used to describe the techniques that are used to perform the filtering process by measuring similarity between two digital objects, typically by assigning a 'similarity score'. Over the years, several approximate matching algorithms have been proposed and are being used in practice. Some of the prominent approximate matching schemes are ssdeep, sdhash, mvHash-B, etc.

This dissertation presents security analyses of existing approximate matching tools and techniques. We show that most of the existing schemes are prone to *active adversary attacks*. An attacker by making feasible changes in the content of the file can intelligently change the final similarity score produced to evade detection. Thus, an alternate hashing scheme is required which can resist this attack.

As a core contributions of this dissertation, we develop a new approximate matching algorithm *FbHash*. We show that our algorithm is secure against active attacks and can detect similarity with 98% accuracy in some common use-cases. We also provide a detailed comparative analysis of our construction with other existing schemes and show that our scheme has a 28% higher accuracy than other schemes for uncompressed file formats (e.g., text files) and a 50% higher accuracy for compressed file formats (e.g., docx, etc.) Our proposed algorithm is able to correlate a file fragment as small as 1% to the source file with an observed 100% detection rate and is able to detect commonality as small as 1% between two documents with an appropriate similarity score. Further, we show that our scheme also produces the least false negatives among all such schemes.

In order to identify the capabilities of similarity matching schemes, it is important to have a systematic method to evaluate the existing and future algorithms. This dissertation also presents a general platform-independent *approximate matching algorithm evaluation tool* to assess existing and future algorithm on four pragmatic test cases on the following metrics: true negative rate, false positive rate, precision, recall, F-score, and Matthews Correlation Coefficient (MCC). In order to understand the true capabilities of an algorithm, it is important to evaluate them on a real-world dataset. Our tool provides a real-world dataset for each of the four test cases previously referred. The tool also provides an automated way to generate a real-world dataset for other cases, which will help support future research in this domain.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The modern world has been turning increasingly digital; conventional books have been replaced by ebooks, letters have been replaced by emails, paper photographs have been replaced by digital images, and compact audio and video cassettes have been replaced by mp3 and mp4 CD/DVDs. Due to the reducing costs of storage devices and their ever-increasing size, the amount of digital data around us has increased exponentially. Consequently, a digital forensic investigator is confronted with several terabytes of digital data on a crime scene, which is too enormous to be analyzed manually. For efficient utilization of time and resources, the foremost requirement of today's forensic investigation process is to have the capability to extract or filter potentially relevant data from all the data collected at a crime scene that an investigator can examine manually in a reasonable amount of time.

The filtering process used in extracting the data typically uses fast hashing-based algorithms. Large files are passed through a hash function to produce a hash output called a digital fingerprint. The fingerprints of the case files are then matched with a known reference dataset, the most popular being the NIST reference data set [7] to extract unknown files. The filtering process can be performed in the following two ways:

- **Blacklisting or Deny-listing** is the process of filtering data by matching them with the set of Known-to-be-bad files (as determined by the investigator). The resultant files after this process are the ones that an investigator needs to examine closely.

- **Whitelisting or Allow-listing** is the process of filtering by matching the files with a set of already Known-to-be-good files. The files passing this process need not be examined by the investigator.

The traditional (cryptographic) hash function could be used to perform the filtering. However, it suffers from a limitation that this kind of filtering only indicates an exact copy of another file. This is due to the fact that even a single bit change in the file content produces a completely unrelated and random-looking hash output [8], whereas the requirement in practical scenarios is often to find similar files. The relevant data or the evidence can be present in the captured data in various forms, such as fragmented, modified, deleted, or partially deleted. Therefore there is a need for a technique that can find similarity between two digital artifacts, not just duplicates.

'Approximate Matching' is a generic term that denotes any technique that can be used to detect similarity between two digital artifacts and produces a 'similarity score' typically on a scale of 0 to 100. The definition and terminology have been defined by Breitinger et al. [9]. According to [9] an approximate matching technique can be characterized into one of the following categories:

- **Bytewise Matching** relies on the byte sequence of the digital object without considering the internal structure of the data object. These techniques are known as fuzzy hashing or similarity hashing.
- **Syntactic Matching** measures the similarity based on the internal structure of the data object. For example structure of TCP packet can be used for network packet analysis.
- **Semantic Matching** relies on the contextual attributes of the digital objects which is more closely related to human perception. It is also called Perceptual Hashing or Robust Hashing.

Our primary focus is Bytewise Approximate matching algorithms (also known as 'Fuzzy Hashing' or 'Similarity Hashing') due to their wide applicability and growing prominence in the digital forensics community. Over the years, several fuzzy hashing algorithms have been proposed. Most of the existing algorithms work in the following two phases:

- **Similarity Digest Generation:** Similarity digest (also called data fingerprint) is a distinct representation of the input data that preserves the similarity features of the data and can be used efficiently and securely to compute similarity score. Generally, the similarity digest is smaller than the original data object, and the similarity digest generation process is a one-way function.
- **Digest comparison:** This phase compares the similarity digest of two data objects and generates the similarity score. In most cases, the similarity score is computed on a

scale of 0 to 100, where 0 indicates no similarity and 100 indicates 100% similarity or an exact copy.

The essential characteristics of an efficient and secure approximate matching algorithms have been defined by NIST [9] [10] and are as follows.

- **Similarity preservation:** The similarity digest should be able to represent the data object in such a way that it can be uniquely identified. Each and every byte of input is expected to influence the final similarity digest, and the similarity score between two digital artifacts.
- **Self-evaluation:** The algorithms should provide the definition and the measure of the accuracy of the resulting similarity score under various scenarios such as default margin of error.
- **Compression:** The generated digest should preferably be smaller than the input size. However, the size of output may or may not be fixed. The digest generation and comparison process should be efficient in terms of required memory space and time.
- **Ease of computation:** Similarity digest calculation (feature extraction) and comparison should be efficient and easy to compute.

Breitinger et al. also stated [9] following fundamental reliability measure that an approximate matching algorithm should have:

- **Sensitivity & robustness** The sensitivity and robustness measures of the algorithm should be provided. The senstivity and robustness is the measure to identitify maximum and minimum size of the data objects and the similarity between two objects that can be detected by the algorithm. There are several tests that can be performed to identify the sensitivity and robustness of an algorithm. For example, fragment detection, common object identification, alignment robustness, and random-noise-resistance [11].

- **Precision & recall** The algorithms should provide the details of the test dataset used to evaluate the performance. Precision, recall, F-score, and other relevant measures should be used to assess the performance.

- **Security** Algorithm should be resistant towards active adversary attacks to ensure the reliability of the results. For example, A malicious user should not be able to manipulate the input in such a way so that dissimilar artifacts appear similar to the algorithm or vice versa. The algorithms should describe whether and how it prevents such attacks.

Similarity queries can be broadly divided into two categories [12][9][11][3]:

1. Resemblance Queries,

2. Containment Queries.

We denote a measure of similarity and commonality between data object D_1 and D_2 by $R(D_1, D_2)$. Similarly, we denote a measure of containment of data object D_1 within data object D_2 by $C(D_1, D_2)$. An approximate Matching algorithm should attain one of the following problems:

1. Object similarity detection : This can be categorized as a resemblance problem. An algorithm should identify similar/related artifacts, such as identifying different versions of a document or malicious code file.

2. Cross correlation : The proposed algorithm should identify artifacts that share a common object, for example, a Microsoft Word document and a HTML document comprising the same the embedded object e.g. an image or a video clip. This can also be characterized as a resemblance query.

3. Embedded object detection : This problem comes under the category of containment. Given a piece of data, such as a JPEG, the algorithm needs to be able to search for (traces of) its existence inside another document, archive, disk image, or network trace.

4. Fragment detection : This is also a containment query. The approximate matching algorithm should identify the fragments of a known artifact. For example, such queries could be used to find the fragments of a deleted and partially overwritten file. Another instance could be when an investigator receives a hard disk that is formatted in quick mode. Thus he is only able to analyze the low-level HDD blocks. These block fragments can be identified in the reference dataset of good and harmful files.

1.1 Motivation

Approximate matching is a relatively new area but conceived as a prominent approach to assist the digital investigation process by automatically filtering correlated data to reduce the amount of data an investigator has to examine manually. Over the last decade, the area has evolved a lot, and several algorithms have been proposed. Some of the most prominent and commonly used approximate matching schemes are ssdeep [1], sdhash [13], mrsh [14] [15], and mvHash [5]. However, there remain several open questions and challenges with regards to these tools as well as the general problem of approximate matching.

The first and foremost challenge is the security aspect of approximate matching algorithms. To understand the reliability and robustness of a new design, it is essential to present a though security analysis. By security, we mean that it should not be possible for a malicious user to mislead the outcome of an algorithm by manipulating the input data. Such kinds of attacks are known as ‘active adversary attacks’. Approximate matching algorithms aim to assist the critical investigation process, although little has been done to analyze their security robustness. Hence there is a strong need for security analysis of existing constructions.

The second major challenge is that very limited or no security properties have been considered in the existing schemes. Therefore, a primary requirement is a design that can provide security and thorough security analysis to prove its resistance against active adversary attacks. Practical implementations of such algorithms often require a tradeoff between security and performance. Security add-ons impact other factors of a system which can incur additional costs. Some of these factors are throughput, scalability, usability, etc. The need of the hour is a construction that provides a balance between security and efficiency while maintaining the accuracy and reliability of the results.

Another challenge is to understand the essential features required from the point of view of practitioners and investigators. The baseline definition and terminology of the approximate matching algorithm is already given by [9]. The NIST Special Publication 800-168 [9] defines the properties at a more general and broader level. However, the definition of similarity as well as the requirements of the approximate matching algorithms vary for different data object types. For example, two files with similar text content may have entirely different underlying structures. They would be perceived as highly dissimilar if an inappropriate algorithm is applied; the actual similarities would remain unnoticed. It is crucial to establish the key characteristics of the approximate matching algorithm based on the requirement of digital forensics practitioners as well as researchers with foundational understanding of different perspectives towards these algorithms.

Another significant challenge is the assessment or evaluation of existing and future approximate matching techniques. For the growth and advancement of this domain, it is essential to have a standard automated evaluation technique. The evaluation technique can be used to assess the absolute and relative performance of existing and future schemes. It is also crucial to perform the test on a real-world dataset and practical test cases. Therefore producing a real-world dataset with known similarity is further required. This thesis aims to address all of the challenges mentioned above and presents solutions, mitigation, and conclusions.

1.2 Contributions

Among the list of publications presented on pages (vi) of (vii) of this dissertation, all the results reported in [16], [17], [18], [19], [6], [8] form the basis of this thesis. In the joint works, the author of the thesis has played a leading role in obtaining the results reported in this thesis. There are three main contributions of this thesis:

1. Security analysis of existing approximate matching algorithms,
2. A new approximate matching design,
3. The evaluation framework.

The following are the details of each of these contributions:

- **Security analysis of existing approximate matching algorithms**: Chapter 3 and 4 present the security analysis of two prominent approximate matching schemes, sdhash and mvHash-B. We briefly describe these results next.

    1. Chapter 3 presents an anti-forensic attack [20] to bypass the sdhash filtering. sdhash is a well-known fuzzy hashing scheme used for finding similarity among files. This digest produces a 'score of similarity' on a scale of 0 to 100. The results include the following:
        - In a prior analysis of sdhash, Breitinger et al. claimed that 20% contents of a file could be modified without influencing the final sdhash digest of that file. They suggested that the file can be modified in certain regions, termed 'gaps', and yet the

  sdhash digest will remain unchanged. In this work, we show that their claim is not entirely correct. In particular, we show that even if 2% of the file contents in the gaps are changed randomly, then the sdhash gets changed with probability close to 1.
- We then provide an algorithm to modify the file contents within the gaps such that the sdhash remains unchanged even when the modifications are about 12% of the gap size.
- On the attack side, the proposed algorithm can deterministically produce collisions by generating many different files corresponding to a given file with a maximal similarity score of 100.

2. mvHash-B is another prominent approximate matching scheme. However, no security analysis of mvHash-B is available in the literature. In chapter 4, we performs the first academic security analysis of this algorithm. Our results include the following:
   - We show that it is possible for an attacker to fool the outcome of the mvHash-B algorithm by causing the similarity score to be close to zero even when the objects are very similar. By similarity of the objects, we mean semantic similarity for text and visual match for images.
   - The designers of mvHash-B had claimed that the scheme is secure against 'active manipulation.' We contest this claim in this chapter. We propose an algorithm that starts with a given document and produces another one of the same size without influencing its semantic and visual meaning (for text

    and image files, respectively) but which has low similarity score as measured by mvHash-B. In our experiments, we show that the similarity score can be reduced from 100 to less than 6 for text and image documents. We performed experiments with 50 text files and 200 images and the average similarity score between the original file and the file produced by our algorithm was found to be 4 for text files and 6 for image files. In fact, if the original file size is small, then the similarity score between the two files was close to 0, almost always.

  - To improve the security of mvHash-B against active adversaries, we propose a modification in the scheme. We show that the amendment prevents the attack we describe in this work.

- **A new secure approximate matching design:** In chapter 5, we propose a new approximate matching scheme called FbHash (Frequency-based Hashing) which we show to produce best correlation results compared to other existing techniques. Our scheme also resists active adversary attack unlike other schemes. The results include the following:

  1. We present design details of FbHash construction, and show the following for our design.
     - We show that our scheme is secure against active attacks and detects similarity with 98% accuracy.
     - We provide a detailed comparative analysis with existing schemes and show that our scheme has a 28%

higher accuracy rate than other schemes for uncompressed file format (e.g., text files) and 50% higher accuracy rate for compressed file format (e.g., docx etc.).

   - Our scheme is able to correlate a fragment as small as 1% to the source file with 100% detection rate and able to detect commonality as small as 1% between two documents with appropriate similarity score.
   - Further, we also show that our scheme produces the least false negatives compared to other schemes.

2. Chapter 6, presents a time and memory-efficient version of `FbHash` and we term it as `FbHash-E`. The chapter includes the following.
   - We show some limitations of the current best approximate matching tool (in terms of security),i.e., `FbHash` and propose a time and memory-efficient version of the same. We term the new design as `FbHash-E`.
   - We present a novel bloom filter based document frequency design implementation in `FbHash-E` that reduces the memory requirements compared to its predecessor `FbHash`.
   - We show that design changes done to improve the performance of `FbHash` do not impact its ability to detect similar files significantly and there is only an average difference of 7.5 in the scores generated by `FbHash` and `FbHash-E`.
   - We show a detailed security analysis of `FbHash-E`, perform more tests to evaluate its robustness. Some

  of these tests were not performed earlier for `FbHash`. We compare the results with other state-of-the-art forensic tools. We show that `FbHash-E` outperforms all the tools in all the security tests conducted.
  - We also show the results of *Consistency Test* and *Code Version Identification Test*, discuss their significance in relation to the evaluation of a forensic tool, and present the comparative results. We further show that `FbHash-E` delivers the best results under these two sets of tests as well.

- **Evaluation framework**: In chapter 8, we present an automated tool to assess the abilities of approximate matching algorithms based on their practical application. Some salient work presented in chapter 7 and 8 are as follows.

  1. We present the findings of a closed survey conducted among a highly experienced group of US federal state and local law enforcement practitioners and researchers, aimed to understand the practitioner and researcher's opinion regarding approximate matching algorithms. The study provides baseline attributes of approximate matching tools that a scheme should possess to meet the real requirement of an investigator.
  2. We present a general platform-independent approximate matching algorithm evaluation tool to assess existing and future algorithms on four pragmatic test cases. The work includes:
     - Each test case performs the evaluation on the following metrics: false-positive rate, false-negative

  rate, precision, recall, F-score, and Matthews Correlation Coefficient (MCC).
  - To understand the true capabilities of an algorithm, it is important to evaluate them on a real-world dataset. We perform each test on real-world data.
  - Our tool provides the first real-world dataset of known similarity for each test case. We also provide an automated platform to generate a real-world dataset for each test case. This dataset can be independently used for further research in this area.

1.3 Organization of the Thesis

In this thesis, we present security analysis and improvements in the security of the existing approximate matching algorithm. We provide a secure approximate matching design and present an approximate matching tool testing framework. Our main results are presented in Chapters 3, 4, 5, 6, 7, and 8. Each chapter provides a literature review, followed by our results and conclusions. The thesis outline is as follows:

- In Chapter 2, we first present an overview of the history and state-of-the-art in the domain of approximate matching algorithms. We review a number of existing approximate matching techniques, which will facilitate understanding of the work discussed in subsequent chapters.

- In Chapter 3, we re-evaluate the security bound of sdhash similarity hashing against active adversary attack. We present an algorithm that can enable an active adversary attack to mislead the sdhash filtering process.

- In Chapter 4, we investigate the mvHash-B algorithm's resistance against active adversary attacks. First, we present an automated attack technique to evade mvHash-B filtering. Then, we propose a few improvements in the design of mvHash-B construction to mitigate such attacks.

- In Chapter 5, we provide a new approximate matching algorithm called "FbHash" or "frequency-based hashing." We show that FbHash is secure against active adversary attacks and produces more reliable results compared to existing state-of-art algorithms. We compare the performance of FbHash with other significant approximate matching algorithms and present a detailed analysis of FbHash.

- In Chapter 6, we propose an improvement over `FbHash`, called `FbHash-E` that has much lower memory footprint, and is computationally faster compared to `FbHash`. We perform various tests to evaluate the security and correctness of FbHash-E.

- In Chapter 7, we discuss the results of a survey conducted in a closed group of highly experienced and trained digital investigators, practitioners, and researchers regarding essential characteristics of approximate matching algorithms.

- In Chapter 8, we present an automated evaluation tool that evaluates the performance of existing and upcoming approximate matching algorithm on a real-world dataset. Our tool also provides a way to generate a real-world dataset for each test case.

- In Chapter 9, we conclude the thesis by discussing the results and giving some future research directions.

Chapter 2

Background and Related Work

This thesis primarily focuses on bytewise approximate matching algorithms. This chapter provides a brief overview of the necessary background required to understand the results included in this thesis. This chapter is divided into three sections. The first section introduces the most popular approximate matching algorithms. The second section focuses on the analysis results of approximate matching algorithms in the literature. The third section discusses the known approximate matching testing tools.

2.1 Introduction to approximate matching algorithm constructions

The first Approximate Matching Technique was proposed in the year 2002 by Nicholas Harbour [21] called dcfldd [1] which is an extension of the well-known disk dump tool dd. **dcfldd** is also called block-based hashing as it divides the input into fixed-size blocks, generates hash each block separately, and concatenates all hash values. By inserting or removing one byte at the begin-

[1]http://dcfldd.sourceforge.net/

ning of the input, the security and accuracy of this scheme can easily be compromised [22]. Since it causes a one-byte shift in each block, hence the resulting hash value is entirely different.

2.1.1 Context Triggered Piecewise Hashing(CTPH)

CTPH can be considered an improvement of dcfldd to overcomes the weakness of dcfldd. Context Triggered Piecewise Hashing (CTPH) was proposed by Kornblum [1] in the year 2006. The CTPH scheme is based on the spamsum algorithm proposed by Andrew et al. [23] for spam email detection. The implementation tool of Kornblum's CTPH algorithm is called ssdeep [2]. CTPH computes a digest of the given file by first dividing the file into several variable size blocks and then concatenating the least significant 6-bits of the hash value of each block. A hash function named FNV [24] is used to compute the hash of each block.

In Context Triggered Piecewise Hashing, the block size is not fixed like dcfldd. Each block is determined by a rolling hash function. A rolling hash function takes a byte string as input and outputs an integer. It proceeds as follows, A window of a fixed size s (in current implementation s = 7 bytes) moves through the input, byte by byte, and generates a rolling hash value of each window.

$$BS_k = B_{k-s+1} + B_{k-s+2} + ... + B_{k-1} + B_k \tag{2.1}$$

Let equation 2.1 denote the byte sequence in the current window of size s at position k within the file and let Rolling_Hash(B$_k$) denotes the corresponding rolling hash value. If Rolling_Hash(B$_k$) matches to a specific value called triggered value, the end of the

[2]https://ssdeep-project.github.io/ssdeep/index.html

current block is identified. Byte B_k is called the trigger point and the current byte sequence BS_k a trigger sequence. The next block starts at byte B_{k+1} and ends at the next trigger point or at the end of the input file. The proposed rolling hash function in [1] is shown in fig 2.1. The proposed algorithm allows to compute the rolling hash value cheaply and efficiently. If B_k is not a trigger point, the next processed byte sequence will be following:

Algorithm 1 Pseudocode of the rolling hash

```
$h_1, h_2, h_3, c, n$ are unsigned 32-bit integers, initialised to zero
$window$ is an array of length $size$ (default $size = 7$)

to update the rolling hash for a byte $c$
  $h_2 = h_2 - h_1$
  $h_2 = h_2 + size \cdot c$                ▷ $h_2$ is the sum of the bytes times the index
  $h_1 = h_1 + c$
  $h_1 = h_1 -$ windows[$n$ mod $size$]     ▷ $h_1$ is the sum of the bytes in the window
  window [$n$ mod $size$] $= c$
  $n = n + 1$
  $h_3 = h_3 << 5$                          ▷ $h_3$ mostly needed to cope with large block sizes values
  $h_3 = h_3 \oplus c$
  return $(h_1 + h_2 + h_3)$
```

Figure 2.1: Pseudo code of rolling hash from [27]

$$BS_{k+1} = B_{k-s+2} + B_{k-s+3} + ... + B_k + B_{k+1} \tag{2.2}$$

The trigger point value is defined as follows. Let b denotes a value called block-size, then the byte B_k is a trigger point if and only if Rolling_Hash(B_k) $\equiv$ -1 mod b. The total number of resultant blocks depends on the value of b. If b is too small, we have too many trigger points and vice-versa. The block-size b is defined by equation 2.3 where b_{min} is the minimum block-size with a default value of 3. S is spamsum length that sets the desired number of blocks(which is by default 64), and n is the input file size in bytes:

$$b = b_{min} \cdot 2^{\lfloor log_2(\frac{n}{s.b_{min}}) \rfloor} \tag{2.3}$$

Once a block is identified by the trigger point, a cryptographic hash (FNV) of this block is computed. Let BS denote a block and h the cryptographic hash function. Then the base64 encoding of the least significant 6 bits of h(BS) is calculated. Let least significant 6 bits of h(BS) is represented as LS6B(h(BS)). The final digest is the concatenation of base64 encoding of LS6B(h(BS)) of each identified block. Since the block-size is used for determining the blocks, only ssdeep hash digest with the same block-size can be compared. ssdeep uses two different block-sizes b and 2b to provide more flexibility.

Two digest are compared by computing the edit-distance [25]. Edit distance is defined by the minimum number of operations required to transform one digest into another. The operations that can be performed are insertion, deletion, and substitution. The final score is computed on a scale of 0-100, where 0 indicates no similarity and 100 indicates a perfect match.

Chen et al. [26], Breitinger et al. [27], Roussev [28] and Seo et al. [29] proposed some modifications to ssdeep to improve its efficiency and security. Baier et al. [20] presented a thorough security analysis of ssdeep and showed that it does not withstand an active adversary for blacklisting and whitelisting. We discuss the results of the work [20] in details in the next section.

2.1.2 sdhash

Roussev [30, 13] proposed a new fuzzy hashing scheme called sdhash. The basic idea of **sdhash** scheme is to identify statistically improbable features that can uniquely represent the input document. A feature is any consecutive 64-byte sequence of the input data. Based on the shannon entropy [31] of each feature, few of them are chosen as statistically improbable features. Only these selected features are used to generate the digest. In order to compute the sdhash digest SHA-1 hash of each selected features is generated and further divided into five parts; we call it sub-hash. Each sub-hash is later inserted into a bloom filter [32]. Thus each selected feature is represented by five bits in the bloom filter. There is an upper limit to the maximum number of elements in one bloom-filter to reduce the false-positive rate. sdhash implementation uses 256 bytes (2^{11} bit) bloom filter with maximum 128 elements per filter. Once the bloom filter has reached its limit, a new bloom filer is created. The final digest is the set of bloom filters.

Two digests are compared by comparing each filter in one digest with all the filters in the other digest. The final score is the average of each comparisons. The resultant similarity score ranges between 0 to 100, where 0 indecates no similarity and 100 indecates the highest similarity. We provide an in-depth, step-by-step description of the sdhash algorithm in chapter 3.

Breitinger et al. [33] showed some weaknesses of sdhash and presented improvements to the scheme. Detailed security and implementation analysis of sdhash is also done by Breitinger et al. in [34]. Chapter 3 and work [17] uncover several implementation bugs and show that it is possible to deceive the similarity score by tampering the input file without changing the perceptual behavior of this file (e.g., image files look almost the same despite the tampering).

2.1.3 bbHash

bbHash approximate matching scheme was proposed by Breitinger et al. in the year 2012 [2]. bbHash aims to two properties of the approximate matching algorithm: 1) **Coverage:** Every byte of input is expected to be involved in final hash digest generation. Thus every byte should influence the hash digest. 2) **Variable sized length:**, Unlike the traditional hash functions, the bbHash design ensures that the hash digest length is proportional to the size of the input. The key idea of bbHash is based on data deduplication (e.g., [35]) and eigenfaces (e.g., [36]). bbHash works in the following three phases:

1. **Building Block Generation** A building block is a random byte sequence of l byte. A set of N building blocks is generated using the algorithm shown in 2.2 where the default value of N is 16. Let bb$_i$ represent ith building block in the set. The set of N building block can be represented as follows, bb$_0$,bb$_1$,bb$_2$,. . . ,bb$_{N-1}$. Since the cardinality of the set N is 16, it can be indexed using a hex digit between 0 to f. The size of one building block (denoted as l) impacts the performance of bbHash. If l is too big then it will shorten the length of hash digest, and if l is too small then it will increase the hash generation time. In order to maintain run time efficiency, bbHash uses a reasonably small value of l. The default value of l is 128 bytes (1024 Bits).

2. **bbHash Digest Calculation** Consider a window of l bytes. Slide the window through the input file byte-by-byte and compute the Hamming distances of all the building blocks against the input byte sequence of the current window. The building block, which results the smallest ham-

```
/** amount: amount of the building blocks; default=16
 *  size: bitsize of each Building block; default=1024 */
uint32_t setBuildingBlocks(int amount, int size) {
    int i; uint32_t *tmp;

    tmp = (uint32_t *)malloc(sizeof(uint32_t) * amount * size/32);
    for (i = 0; i < amount * size / 32; i++) {
        tmp[i] = rand() << 10 ^ rand();
   }
    return *tmp;
}
```

Figure 2.2: Generation of Building Blocks from [2]

ming distance and is also smaller than a certain threshold, is called the triggered building block. All triggered building blocks are used to generate the final bbHash digest. The bbHash digest is an ordered sequence of triggered building blocks indexes. figure 2.3 shows the pseudocode of the bbHash algorithm.

bbHash ensure that every byte of the input is represented by atleast one trigger building block in the digest. Thus for every l-bytes, there should be a trigger. The authors have shown that for l=128 Bytes and N=16, threshold value 461 provides full coverage.

3. **Digest Comparison** Similarity between two files can be calculated by comparing the corresponding digest. Digest comparison can be done in the following two ways: 1) By storing the digests of both the file in bloom filters and calculating the hamming distance between both the bloom filters. 2) By calculating the weighted edit distance of both the digest.

A significant limitation of bbhash is that it does not perform efficiently and is extremely slow for practical use. For instance,

```
Algorithm 1 Pseudocode of the bbHash Algorithm
$l$                 ▷ Processed chunk size; default length is 128 bytes
$N$                 ▷ Amount of building blocks; default is 16
$BS_i$              ▷ A byte sequence of length $l$ starting at the $i$-th byte of the input
$t$                 ▷ Threshold; default is 461
$L_f$               ▷ Length of the input file in bytes
$mHD, tHD, tk$      ▷ Unsigned int; temporary variables
$signature$         ▷ String variable for final hash value

  $bb$ = set_building_blocks($N$, $l$)          ▷ Processing is given in Fig. 1
  for $i = 0 \rightarrow L_f - 1 - l$ do       ▷ run through input, byte-by-byte
  $tHD = mHD = 0xffffffff$                     ▷ reset
      for $k = 0 \rightarrow N - 1$ do         ▷ run through all building blocks
          $tHD$ = getHammingDistance($bb_k$, $BS_i$)
          if $tHD < mHD$ then                  ▷ two same HDs will use the smaller $k$
              $mHD = tHD$
              $tk = k$                         ▷ $tk$ is a hex digit
          end if
      end for
      if $mHD < t$ then signature = "signature" + $tk$     ▷ Conversion to string
      end if
  end for
```

Figure 2.3: bbHash Algorithm from [2]

it requires 10 minutes to process a 10 MB file.

2.1.4 mvHash

Majority vote hashing (mvHash) [37] was first proposed by Åstebøl et al. in the year 2012. mvHash algorithm is based on the idea of majority voting in conjunction with run-length encoding to represent the input data. The paper [37] present two variants of the algorithm: 1) mvHash-B and mvHash-L. Later in the year, 2013 Breitinger et al. published a detailed description of mvHash-B design and analysis in [5]

mvHash-B works in the following three phases:

1. **Majority Votes:** This phase converts the input data into a sequence of 0x00s and 0xFFs. The decision has been made

by taking the neighboring bytes into account. If the majority of bits are 1's in a particular byte's n-neighborhood [38], it is set to 0xFF otherwise to 0x00.

2. **Encoding the majority vote bit sequence with RLE:** Run-length encoding [39] is a form of lossless data compression. Run-length encoding counts identical subsequent bytes. mvHash starts the RLE count with the number of identical 0x00 bytes.

3. **Fingerprint generation using Bloom filters:** mvHash stores the resultant RLE sequence in bloom filter [32] for fast and efficient comparison.

Two mvHash-B digests are compared by computing the Hamming distance between the bloom filters. We discuss the mvHash-B design in details in chapter 4. The key difference between mvHash-B and mvHash-L is that mvHash-L does not use bloom filters. Instead, it compares the RLE sequence using the Levenshtein distance [25]. Levenshtein distance is a comparison operation which computes the number of edit operation required to transform one string into another. The calculation of Levenshtein distance is very time-consuming, whereas hamming distance computation is very fast. Therefore mvHash-B is much faster than mvHash-L. Breitinger et al. have shown in their work [5] that mvHash-B is faster than all other previously proposed similarity preserving hashing algorithms. Although there is a limitation of the mvHash-B algorithm in that it requires a specific configuration for each file type.

2.1.5 mrsh and mrsh-v2 Hash

Roussev et al. introduced a variant of ssdeep called 'Multi-resolution similarity hashing (mrsh)' [14] in the year 2007. In this design, rolling hash (used for chunk identification in ssdeep) is replaced by a Polynomial hash function djb2 and another cryptographic hash function MD5 is used in the place of FNV hash function. The final digest is represented in bloom filters. Breitinger et al. in [15] presented an extension of mrsh called mrsh-v2. mrsh-v2 identifies trigger points using rolling hash in any given byte sequence (e.g., a file or a device) to identify the chunks in the input data. Next, each chunk is hashed using FNV. Till this point, mrsh-v2 works exactly like ssdeep. mrsh-v2 represents the hash digest in bloom filters. Bloom filter is an array of size m elements. In mrsh-v2 implementation m=256 bytes (2^{11} bits). FNV hash outputs the 64-bit hash. The least significant 55 bits are taken and divided into five sub-hashes of 11 bits. Each sub-hash sets one bit in the bloom filter. mrsh-v2 allows a maximum of 160 chunks per Bloom filter which means at most 160*5=800 bits of a bloom filter can be set. If this limit is reached, a new Bloom filter is created. Hence, the final fingerprint for an input can be a sequence of multiple Bloom filters. Two digests are compared by calculating the Hamming distance between the bloom filters. As we can see, mrsh-v2 is a combination of the ssdeep and sdhash algorithm.

2.1.6 Lempel-Ziv Jaccard Distance(LZJD)

LZJD is a comparatively recent similarity hashing scheme proposed by Raff et al. [40] in the year 2018. LZJD uses Lempel–Ziv 77 [41] compression algorithm to generate the digest and Jaccard similarity [42, 43] to compare two digests. Lempel–Ziv 77 is a universal algorithm for sequential data compression. Lempel–Ziv 77 splits the input data into unique variable-size sub-strings of shortest redundancy. The algorithm starts with an empty set and a substring of size 1. Each previously unseen smallest sub-string is added to the set. If the current substring has appeared in the set, then the substring size is increased by adding the next subsequent byte in the input until a new substring is encountered. After each insertion, the substring size is reset to one. The same is repeated until the end of the input. Once the substring set of both input is generated. The Jaccard similarity between the two sets is calculated. A faster LZJDh [44] approach is used to compute approximate similarity. The authors have shown that LZJD performs fastest compared to all currently existing approximate matching algorithms. They have shown that it is 60 times faster than the sdhash scheme.

There are few more similarity hashing algorithms that have also been proposed such as simHash [45], minHash [12], and TLSH [46]. However, they do not completely align with the concept of approximate matching. They more coincide with the idea of Locality-Sensitive Hashing(LSH). Harichandran et al. [47] clarify the distinction between the two. LSH algorithm hashes similar input items into the same buckets with high probability, whereas approximate matching outputs a comparable similarity digest, which can be used for whitelist and blacklist filtering.

Apart from the above-mentioned schemes, several forensic tool-kits use different similarity hashing techniques to determine related or similar documents. FTK or Forensic Toolkit is one of the popular computer forensics tools build by AccessData. FTK [48] performs cluster analysis to identify related documents and email threads. Encase [49] is another well know digital investigations tool by Guidance Software. Encase performs an entropy analysis technique to discover near matches. X-ways Forensics [50] is an integrated computer forensics software. X-ways incorporates a new similarity hashing technology called FuzZyDoc to identify known documents. Autopsy [51], an open-source digital forensics tool, uses sdhash in its AHBM [3] (Approximate Hash Based Matching) module to match files against other files or search for similar files.

2.2 Analysis of Approximate Matching Algorithms

The analysis of approximate matching algorithms can be categorized into two broad categories: 1) Performance Analysis 2) Security Analysis. In this section, we review existing literature in the both categories.

2.2.1 Performance Analysis

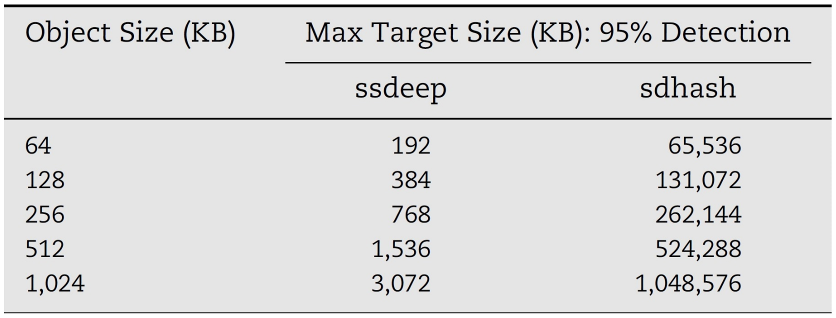
There is substantial literature available regarding performance analysis. The performance analysis measures include scalability, robustness, sensitivity, efficiency, etc. The first detailed study was performed by Roussev [3] in the year 2011. The paper provides a baseline evaluation of the approximate matching tool capabilities in a controlled environment and real-world

[3]https://github.com/pcbje/autopsy-ahbm

data. Controlled environment evaluation is performed to find an upper bound of the abilities of the tools under ideal conditions. Controlled environment evaluation considered following three scenarios:

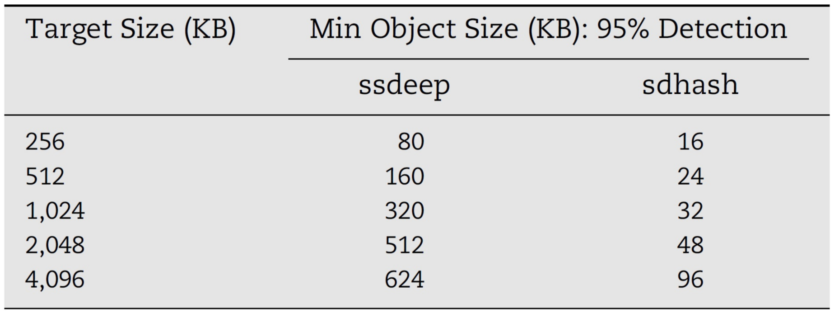
- Embedded object detection: This test aims to measure the ability of an approximate matching tool to correlate a known embedded object to a document. For example, Given an object, such as a JPEG image, the algorithm needs to search for its existence inside another document, archive, disk image, or network trace.
- Single-common-block file correlation: This test simulates a situation where two files have a single common object or data. For example, documents sharing an image/header/-footer or pieces of software sharing library code.
- Multiple-common-blocks file correlation: This test identifies the ability of a tool or algorithm to find similarities between two digital artifacts where commonality might be significant but fragmented.

This paper compared the performance of two most prominent schemes, ssdeep and sdhash. Figure 2.4 shows table from [3] showing the result of embedded object detection test. The gtable in the figure shows that maximum file size for which ssdeep and sdhash detected similarity with atleast 95% true positive rate. The results conclude that ssdeep can correlate an embedded object to its source document if the object size is no less than 1/3 of the source file. However, sdhash results reliably regardless of the document or object size. Fig. 2.5 shows snippet from [3] showing the result of single common block correlation test. The figure represents the minimum size of shared single-common-block-data or object between two documents that can be identified with atleast a 95% true positive rate. The results show that sdhash can detect similarity for smaller objects or data-block in comparison to ssdeep. The result of the multiple common block file correlation test is summarized in fig 2.6 from [3] where the total amount of common data is split into four and eight (non-overlapping) pieces, respectively. Each piece is then independently and randomly placed into the targets. The figure presents the probability of correlation detection of ssdeep and sdhash. The results indicate that sdhash performs better than ssdeep.



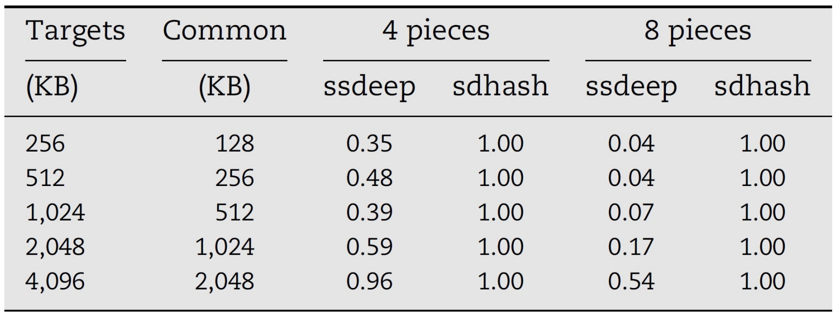
Object Size (KB)	Max Target Size (KB): 95% Detection	
	ssdeep	sdhash
64	192	65,536
128	384	131,072
256	768	262,144
512	1,536	524,288
1,024	3,072	1,048,576

Figure 2.4: Embedded Object Detection(Higher is better) from [3]



Target Size (KB)	Min Object Size (KB): 95% Detection	
	ssdeep	sdhash
256	80	16
512	160	24
1,024	320	32
2,048	512	48
4,096	624	96

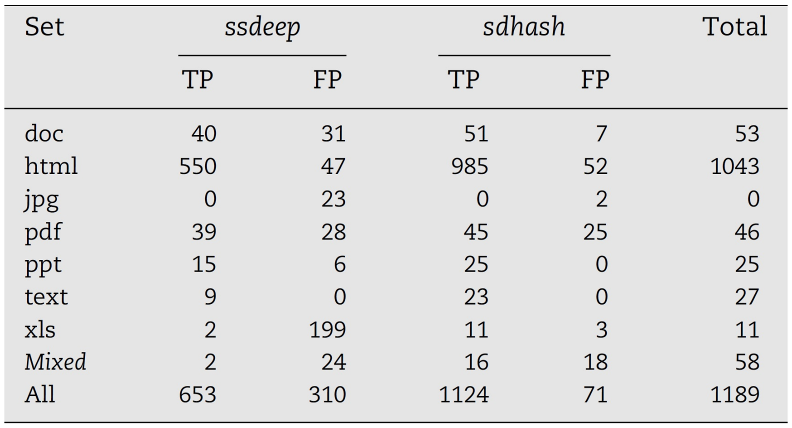
Figure 2.5: Single Common Block file correlation(Lower is better) from [3]



Targets	Common	4 pieces		8 pieces	
(KB)	(KB)	ssdeep	sdhash	ssdeep	sdhash
256	128	0.35	1.00	0.04	1.00
512	256	0.48	1.00	0.04	1.00
1,024	512	0.39	1.00	0.07	1.00
2,048	1,024	0.59	1.00	0.17	1.00
4,096	2,048	0.96	1.00	0.54	1.00

Figure 2.6: Multiple common block correlation probability(Higher is better) from [3]

The result of the real-world data is shown in Fig. 2.7 [3]. This figure provides the number of true and false file detection for both the tools for various file types. The 'total' column provides the total number of known true positive detection identified by both the tools.



Set	*ssdeep*		*sdhash*		Total
	TP	FP	TP	FP	
doc	40	31	51	7	53
html	550	47	985	52	1043
jpg	0	23	0	2	0
pdf	39	28	45	25	46
ppt	15	6	25	0	25
text	9	0	23	0	27
xls	2	199	11	3	11
Mixed	2	24	16	18	58
All	653	310	1124	71	1189

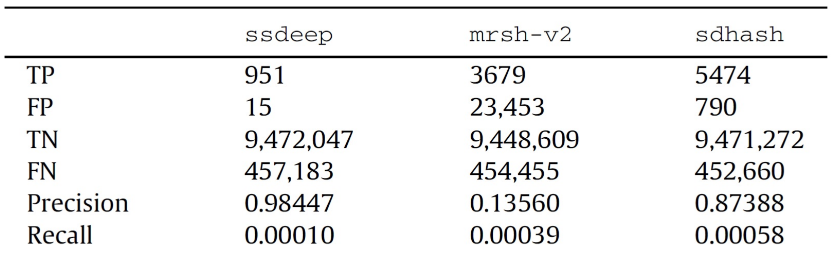
Figure 2.7: True positive, False Positive and Total known positive from [3]

All of the above results show that sdhash outperforms ssdeep in terms of all the above-mentioned test cases. However, further analysis must validate the results in different scenarios, such as real network and RAM captures.

Breitinger et al. in [11] proposed a framework to test the approximate matching algorithms called 'FRASH' which includes two test cases called efficiency and sensitivity, & robustness. The efficiency test comprises the runtime efficiency, fingerprint comparison, and compression tests. The efficiency test results show that ssdeep has better compression and fingerprint comparison, whereas sdhash supports parallelism and outperforms ssdeep in runtime efficiency. Sensitivity & robustness test comprises of four tests: single-common-block correlation, fragment detection, alignment robustness, and random-noise-resistance. Test results show that sdhash dominates in all of the tests. We discuss FRASH in detail in section 2.3. Later, Breitinger et al. in their work [52] extended the FRASH framework and incorporated precision and recall test. The results of this work also align with the previous outcome.

[4] presents automated precision and recall tests on real-world data. Using real-world data yields more realistic results and allows a better characterization of the behavior of approximate matching algorithms. Precision and recall tests quantify the detection error trade-off between false positive and false negative rates for the algorithms. Fig. 2.8 shows that recall rates for all the tools are relatively low; therefore, negative results can not be trusted completely. Precision rates for ssdeep and sdhash are high, which means that a positive result is a clear indication of similarity at the byte level. Overall, sdhash shows the best performance.

Most performance analysis papers in existing literature have extensively analyzed ssdeep and sdhash tools, and a few have included mrsh-v2. This is due to the popularity of these algorithms as well as the fact that ssdeep and sdhash have shown



	ssdeep	mrsh-v2	sdhash
TP	951	3679	5474
FP	15	23,453	790
TN	9,472,047	9,448,609	9,471,272
FN	457,183	454,455	452,660
Precision	0.98447	0.13560	0.87388
Recall	0.00010	0.00039	0.00058

Figure 2.8: Results of Precision and recall tests over the ssdeep, sdhash and mrsh-v2 approximate matching algorithms from [4]

the most promising results so far. This points to the belief of the forensic community that ssdeep and sdhash are the most prominent algorithms.

2.2.2 Security analysis

Security analysis is necessary to ensure the reliability of a tool. If a malicious program or image in a suspect's hard disk can evade detection by minor and non-significant modifications to the file, then this will defeat the purpose of the tool. Such modifications can easily be automated and can be used to make all malicious data detection proof. Such counter-forensic tools will impact investigation results, hence it is essential to ensure the security features of the approximate matching algorithm. However, very little security studies have been attempted in the current literature. In this section, we provide a brief review of published security analysis.

2.2.2.1 Security analysis of ssdeep

Baier et al. [20] have shown an active anti-blacklisting attack on the ssdeep algorithm and practically proven that ssdeep does not withstand an active adversary against a blacklist. An Anti-blacklisting attack generates false-negative results. Let F_1 be a malicious file and F_2 be another file generated by manipulating F_1 in a way that it is semantically similar to F_1. However, ssdeep finds F_2 as non-similar to F_1, which is a false negative. [20] presents two anti-blacklisting approaches as follows:

- Editing between trigger points: As we discussed in § 2.1 ssdeep first divides the input byte sequence into several blocks by identifying trigger points. Interestingly, a trigger point depends only on seven bytes. The critical point is that apart from the seven bytes of a block that contribute to the value of the trigger point; any other byte of each block can be modified without affecting the trigger point. The modification is done in such a way so that it preserves the semantic similarity of both the files, for example, changing a character from upper case to lower case or replacing spaces with underscores or vice-versa. Only after one-byte modification in each block, the corresponding traditional hash value will change with a high probability, and it will change the resultant ssdeep digest completely. This anti-blacklisting approach is suitable for text files and bitmap images.

- Adding trigger points: This attack takes advantage of an implementation weakness of the ssdeep scheme, which considers only 64 trigger points. Suppose a file has more than 64 trigger points. In such case, ssdeep considers only the first 63 trigger points, calculates their traditional hash, and adds LS6B (least significant six bits) of its base64 encoding to the digest. The rest of the trigger points are ignored, and the last block will include all remaining bytes in the files.

This attack is easily applicable to jpg files or pdf files. For example, we have a file F_1. We insert 63 trigger sequences in the header of F_1. Let's term the resultant modified file with the updated header as F_2. This will not affect the semantic of the contents of the file. However, it will alter the digest significantly. The authors have provided a way to pre-compute a set of global trigger sequences. To deceive the similarity between F_1 and F_2, these global trigger sequences (producing different base64 hash strings than for F_1) can be used.

2.2.2.2 Security analysis of sdhash

A detailed analysis of sdhash has been done by Breitinger et al. [34]. Their work shows that it is possible to perform undiscovered modifications to the input of sdhash. An undiscovered modification implies that the input can be modified without influencing the final sdhash digest. Hence two or more documents can have exactly the same sdhash digest. Breitinger et al. [33] have shown that 20% of the input bytes are not part of any selected feature. Therefore any alteration to these bytes will not influence the sdhash digest. The author has termed such part of the input as 'gap'. This work also uncovers a few implementation errors in sdhash. We discuss this work in detail in chapter 3.

Security analysis of the remaining schemes has not been done in the current literature. The most significant constructions are ssdeep, sdhash, bbhash, mvHash, and mrsh-v2. Though they are being used to assist critical investigations, little has been done to analyze their security robustness. For example, FNV (Fowler–Noll–Vo) hash function is used in ssdeep construction to compute the hash. FNV hash is a non-cryptographic hash function based on multiplication and xor operations. Similarly, mvHash uses Run-length encoding (which claims no security characteristics) in its construction. So these schemes may not be secure and are likely to be prone to attacks. Any sensible design strategy should provide analysis of its security but it is lacking in these schemes. Therefore, independent security analysis of the schemes is needed to verify their reliability, dependability, and robustness. Emergence of new techniques of analysis can also be take place by such studies. We have pointed to the previous security analysis of ssdeep and sdhash in this section. Another scheme bbHash has low practical relevance due to its run time inefficiency (e.g. a 10 MB file needs nearly 2 minutes). We present a thorough security analysis of sdhash and mvHash in chapter 3 and chapter 4, respectively.

2.3 Approximate matching algorithm testing tools

The first approximate matching tool testing framework was proposed by Breitinger et al. [11] in the year 2013. It was termed "FRamework to test Algorithms of Similarity Hashing" (FRASH). This tool provides an automated assessment of efficiency, sensitivity, and robustness. Later, precision and recall tests were added by Breitinger et al. in the work [52]. All of these tests were conducted on synthetic data. While it is convenient to use synthetic data as it provides perfect ground truth, it does not yield realistic results. Further, Breitinger et al. in the work [4] presented automated precision and recall tests on real-world data by relating approximate matching results to the longest common substring (LCS).

FRASH generates the synthetic data by creating manual mutations of the target files. The sensitivity and robustness test is performed on the following four test cases.

- Single-common block correlation: analyzes the smallest amount of commonality between two data objects which can be correlated by the similarity hashing tool. The dataset is generated by generating two random files and overwriting a common block on both the files at different and randomly generated offsets. The same has been done for various sizes of randomly generated blocks.

- Fragment detection: identifies the smallest fragment of a file that can be matched by the similarity tool. The fragments are generated by cutting the randomly generated files.

- Alignment robustness: inserts blocks of various sizes at the beginning of a file; this attribute simulates scenarios like expanding log files or emails.

- Random-noise-resistance: analyses random modification in an input. The test is performed by randomly altering the input at a random position. Various amount of arbitrary modification has been done.

Precision and recall tests have also been done on synthetic and real-world data. These tests are performed to indicate the similarity algorithm's reliability. Real-world data ground truth has been defined by using 'approximate longest common substring (aLCS)', which computes the longest common substrings between two files. The work [4] declares two inputs as similar if their aLCS is sufficient (e.g., 1% of the total input length, or at least 2 KiB).

While this framework is the first step in the right direction, there are several concerns with FRASH to be used practically. As Harichandran et al. rightly mention in [47], it is challenging to implement FRASH, which is why [47] couldn't compare many algorithms. Another major issue is the definition of ground truth for real-world data. Longest common substring does not reflect the real similarity in many cases. The ground truth varies for each test case. Hence there should be a dedicated dataset for each test case.

2.4 Applications of Approximate Matching Algorithms

Over the years, researchers have shown the application of approximate matching algorithms in various areas apart from digital forensics [47]. In this section, we review a number of currently known results.

2.4.1 Network traffic analysis

Gupta [53] and Breitinger et al. [54] have shown a novel technique using approximate matching to detect files in network traffic. The papers propose the optimization and application of the presented technique on single network packets. The authors conclude that approximate matching with minor modifications can reliably perform file identification on network traffic. These works have shown the results on random network data as well as on real-world data. The paper demonstrates that in the case of random data the detection rate is 100% and in case of real-world data, the detection is done with the false positive rate between 10^{-4} and 10^{-5}. Jimoh [55] presented the performance testing of GPU and CPU implementation of sdhash to determine its suitability in high-network traffic environments. The test results demonstrate that CPU implementation of sdhash is more appropriate for small to medium-network environment, whereas GPU implementation is more suitable for high-network environment and sensitive data.

2.4.2 Malware Detection

Several recent studies have shown the application of approximate matching to malware detection. A study by Bjelland et al. [56] shows that approximate matching tools may be able to identify malware by comparing unknown documents to known malware. Another interesting work by Payer et al. [57] presents a technique to create diversified malware instances with low similarity but equivalent functionality. It demonstrates the effectiveness of similarity metrics on diversified malware instances and concludes that similarity-based metrics are not effective due to the variations in the binary layout. In the year 2015, Faruki el al. has presented a mechanism called AndroSimilar [58] to detect variants of Android malware. The paper uses the concept of sdhash [13] to find the similarity. The authors have shown that AndroSimilar effectively discovers the existing and variants of known malware families with an accuracy of more than 76%.

2.4.3 Data exfiltration and leakage prevention

Symantec [59] has demonstrated a new technique, ‘Describe, Fingerprint, Learn’ to prevent data loss. Here, fingerprint part may use partial data matches, which is similar to the concept of approximate matching algorithm. Jimoh [55] examined the suitability of the approximate matching algorithm to prevent data exfiltration and leakage. The paper tests sdhash implementation and conclude that while sdhash may work well for detecting exfiltration of sensitive files, it can not assure complete exfiltration and leakage prevention. Furthermore, it is possible for a malicious insider to trick the algorithm if they are aware of its presence in the network.

Chapter 3

A Collision attack on sdhash similarity hashing

Sdhash, proposed by Roussev (Roussev, 2010a) in 2010, is one of the most widely used fuzzy hashing schemes. It is used as a third-party module in the popular forensic toolkit 'Autopsy/Sleuthkit' [1] and in another toolkit 'BitCurator' [2].

Breitinger et al. analyzed sdhash in [34, 33] and stated that "approximately 20% of the input bytes do not influence the similarity digest. Thus it is possible to do undiscovered modifications within gaps". In this chapter, we show that this claim is not entirely correct. We show that if data between the 'gaps' is randomly modified, the digest changes even when the modifications are only about 2% of the 'gap size.' After that, we propose an algorithm that can generate multiple files with a sdhash similarity score of 100 corresponding to a given file by modifying up to 12% of the 'gap size'. The proposed algorithm can also be used to carry out an anti-forensic mechanism that defeats the purpose of digital forensic investigation by filtering

[1] http://sleuthkit.org/

[2] https://bitcurator.net/

out similar files from a given storage media. An attacker could generate multiple dissimilar files corresponding to a particular file with 100% matching sdhash digest using our technique.

The rest of the chapter is organized as follows: We discuss notations and definitions used in this chapter in section 3.1. In section 3.2, we have provided an elaborate description of the sdhash scheme. Existing analysis of the scheme is presented in section 3.3. section 3.4 contains our analysis and attack on sdhash and proposed algorithm in the appendix A. Finally, we conclude the chapter in section 3.5 by proposing solutions to mitigate our attack on sdhash.

3.1 Notations

Following notations are used throughout this chapter:

- D denotes the input data object of N bytes, D = $B_0B_1B_2.....B_N$, where B_i is the i^{th} byte of D .
- f_k is a L byte subsequence of consecutive bytes of data object D. It is termed the *k*th 'feature' of the data object. In the sdhash implementation, L = 64, $f_k = B_{k+0}B_{k+1}B_{k+2}.....B_{k+63}$ where $0 \leq k < n$ and n is the total number of features of the data object D. Thus n = N - L + 1.
- H(X) represents the entropy of random variable X.
- $H_{max}(X)$ represents the maximum entropy of random variable X.
- $H_{min}(X)$ represents the minimum entropy of random variable X.

- $H_{norm}(X)$ denotes the normalized entropy of random variable X.
- nbf_k denotes the next byte of feature f_k of data object D.
- $R_{prec,D}(f_k)$ denotes the precedence rank of feature f_k of data object D.
- $R_{pop,D}(f_k)$ denotes the popularity score of feature f_k of data object D.
- bf denotes the bloom filter of 256 bytes.
- $\overline{bf}$ represents the number of features within bloom filter bf.
- $|bf|$ denotes number of bits set to one within the bloom filter bf.
- t denotes some threshold (sdhash uses t = 16).
- $SF_{score}(bf_1,bf_2)$ represents the similarity score of bloom filter bf_1 and bf_2.

3.2 Description of sdhash

We now describe the working of sdhash using the notation defined in § 3.1. Given a data object D of length N bytes ($B_0B_1B_2.....B_{N-1}$), a feature f_k is a subset of L (= 64) consecutive bytes of D, that is f_k:$B_{k+0}B_{k+1}B_{k+2}...B_{k+63}$ where $0 \leq k < n$ and n = N-L+1. In order to generate sdhash fingerprint, the first step is to calculate the normalized entropy of each feature. Entropy of a random variable X with probability P_X and alphabet α is defined as

$$H(X) = \sum_{x \in \alpha} (P[X = x] log_2 P[X = x])$$

The entropy of X attains its maximum value if $P[X = x] = \frac{1}{|\alpha|}, \forall x \in \alpha$; that is, if all possibilities for X are equiprobable. This maximum value of entropy $H_{max}(X)$ is $\log_2|\alpha|$. Similarly, the entropy is minimum if $\exists x \in \alpha : P[X = x] = 1$; hence $H_{min}(X) = 0$. Entropy of a random variable ranges between 0 to $\log_2|\alpha|$. Normalized entropy of a random variable X is defined as $H_{norm}(X)= \frac{H(X)}{H_{max}(X)}$. The normalized entropy ranges between 0 to 1. Random variable in the context of a feature (f_k) is the next byte of the feature (f_k), represented as nbf_k. In sdhash implementation, α is the set of all possible 256 values of x. The probability distribution of nbf_k is defined as:

$$\forall x \in \alpha \; P[nbf_k = x] = \frac{|\{j|B_{k+j} = x, 0 \leq j < 64\}|}{64}$$

where $f_k = B_{k+0}B_{k+1}B_{k+2}.....B_{k+63}$. The Entropy of nbf_k is:

$$H(nbf_k)= \sum_{x\in\alpha}(P[nbf_k = x]log_2P[nbf_k = x])$$

$H_{max}(nbf_k)= \log_2|\alpha|= 8$ and $H_{min}(nbf_k)=0$. Normalized entropy of nbf_k is $\frac{H(nbf_k)}{H_{max}(nbf_k)} = \frac{H(nbf_k)}{8}$. Range of normalized entropy of nbf_k is 0 to 1. It is being scaled up to the range 0 to 1000 and represented by $H_{norm}(nbf_k)$:

$$H_{norm}(nbf_k)= \left\lfloor 1000 * \frac{H(nbf_k)}{8} \right\rfloor$$

After calculating the normalized entropy of each feature, a precedence rank is assigned to the respective feature of the data object D based on the empirical observation of probability density function for normalized entropy of experimental data set.

Let Q is the experimental data set of q data objects $D^1D^2D^3.....D^q$ of same type and same size. Here the random variable is normalized entropy of next data object's nbf_k of set

Q, represented as nenfd_Q. Let A is a set of integers from 0 to 1000 i.e. 0,1,2,....,1000.

For a∈A $P[nenfd\_Q = a] = \frac{|\{(i,k)|H_{norm}(nbf_k^i)=a, 0\leq k<n, 0\leq i<q\}|}{qn}$

where q is number of data object in set Q, nbf_k^i is next byte of feature f_k of D^i data object, 0≤i<q, 0≤k<n. $H_{norm}(\mathrm{nbf}_k^i)$ is normalized entropy of nbf_k^i. Each D^i consists n features. A characteristic probability distribution of each type of data object (i.e. doc, html, gz etc.) can be found.

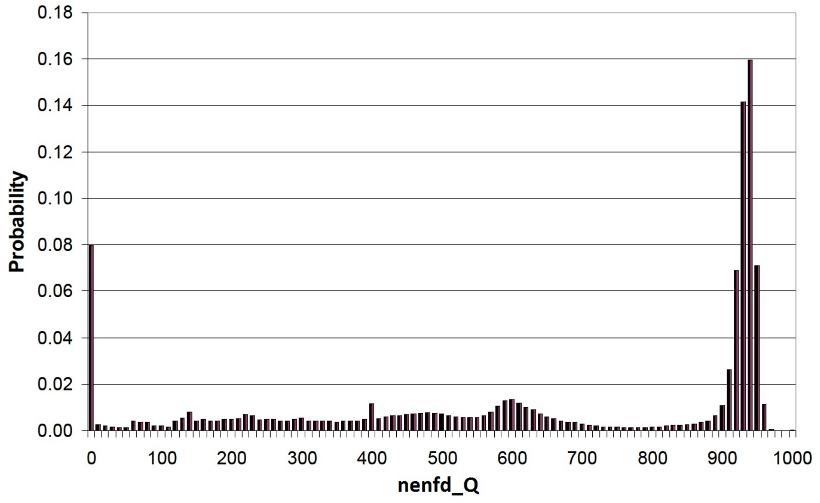


Figure 3.1: Empirical probability density function for experimental data set of doc files taken from (Roussev, 2009, 2010a)

Based on the probability distribution, each element of set A (all possible outcomes) is now assigned a rank. Let t_a = Pr[nenfd_Q=a] ∀ a∈A, where A is the set of integers {0,1,2,3,. . . .,1000}.

t_0= Pr[nenfd_Q=0],
t_1= Pr[nenfd_Q=1],

$$\vdots$$

$$t_{1000} = \Pr[\text{nenfd\_Q}=1000].$$

We assign a rank r_i to each t_i as follows: r_i=1000 if t_i is the largest, and r_i=0 if t_i is the smallest.
Now each feature f_k of D is assigned a precedence rank $R_{prec,D}(f_k)$ as follows:

$$\forall f_k \text{ of D}, R_{prec,D}(f_k) = r_i, \text{ where } \Pr[\text{nenfd\_Q}=H_{norm}(nbf_k)]=t_i$$

where D is the given data object, n is number of features of data object D and $0 \leq k < n$. Data type of data object D and data objects $D^i (0 \leq i < q)$ of set Q is the same. $H_{norm}(nbf_k)$ is normalized entropy of next byte of feature f_k of data object D. Essentially, the least common f_k gets the lowest rank whereas the most common one is assigned the highest rank.
Now based on the precedence rank, each feature(f_k) is assigned a popularity score denoted by $R_{pop,D}(f_k)$. The non-zero popularity score of a feature f_k of a data object D shows that there are ($R_{pop,D}(f_k)$+W-1) W-neighboring features of a feature f_k such that the precedence rank of its left neighboring feature f_i is greater than the precedence rank of f_k; and precedence rank of its right neighboring features f_j is greater than or equal to precedence rank of f_k where $i<k$, $j>k$; and number of f_i + number of f_j = ($R_{pop,D}(f_k)$+W-1). W-neighboring features of feature f_k: A feature f_{nb} is called a W-neighboring feature of feature f_k if $|k-W| < nb < k$ or $k < nb < |k+W|$. If $|k-W| < nb < k$: feature f_{nb} is called left neighboring feature of feature f_k. If $k < nb < |k+W|$: feature f_{nb} is called right neighboring feature of feature f_k. $R_{pop,D}(f_k)$ for $0 \leq k < n$ is calculated as follows:

- Initialize $R_{pop,D}(f_k) = 0$ for each $0 \leq k < n$.
- Consider a window of size W (64).
- For every sliding window leftmost feature f_k with lowest $R_{prec,D}(f_k)$ is taken and value of $R_{prec,D}(f_k)$ is incremented by 1.
- Slide window by one and same steps are repeated (n-W) times,.

Fig.2 shows an example of the $R_{pop,D}(f_k)$ calculation of a data object D where n=18.

R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				1														
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				2														
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				3														
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				4														
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				4	1													
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				4	1							1						

Figure 3.2: Popularity Rank Calculation from (Roussev, 2009, 2010a)

Now features with $R_{pop,D}(f_k) \geq$ t(threshold) are selected (in sdhash implementation t=16). Selected features are the least likely features to occur in any data object. These features are called "Statistically Improbable Features". These Statistically Improbable Features will be used to generate fingerprint of the data object D. Let $\{f_{s_0}, f_{s_1}, ..., f_{s_x}\}$ are the selected features, where $0 < x < n$, and n is the total number of features of data object D. SHA-1 [60] hash of each selected feature is calculated.

Then the resultant 160 bit hash is split into 5 chunks of 32 bits and least significant 11 bits of each chunk are used to address a bit in the bloom filter array. sdhash implementation uses 256 byte (2^{11} bit) bloom filter with maximum 128 element per filter (i.e. 5 bits per feature hence 640 bits per bloom filter).
Similarity between two different sdhash digests is defined as the number of overlapping bits of the corresponding bloom filters. Let bf_1 and bf_2 are two bloom filters. Then the similarity between two bloom filter is defined as follows and represented as $SF_{score}(bf_1,bf_2)$:

$$SF_{score}(bf_1, bf_2) = \begin{cases} 0, & \text{if } e \leq C \\ [100\frac{e-C}{E_{max}-C}], & \text{otherwise} \end{cases}$$

where $e = |bf_1 \cap bf_2|$ (number of overlapping bits), C is the cut-toff of minimum and maximum number of possible overlapping bits by chance, defined as $C=\alpha(E_{max} - E_{min}) + E_{min}$ and E_{min}, E_{min} are minimum and maximum number of overlapping bits by chance respectively.

3.3 Existing Results

Implementation and security analysis of sdhash has been done by Breitinger et al. [34]. Two of the implementation bugs, 'Window size bug' and 'left most bug' mentioned in [34] still exist in the latest version 3.4 of sdhash implementation.

Listing 3.1 shows the implementation of above stated bugs. At line number 13, there is an error in first condition that causes incorrect identification of minimum precedence rank ($R_{prec,D}(f_k)$), referred to as the 'Window size bug'. This error can be removed by replacing the first condition of while loop with 'chunk_ranks[i+pop_win-1] $\geq$ min_rank'. There is another error in the if condition at line number 14-15 and 26-27, that has been referred to as the 'Left most bug'.

If two features (f_i, f_j) have equal precedence rank ($R_{prec,D}(f_i) = R_{prec,D}(f_j)$) and are lowest within a popularity window, then this condition will cause the selection of right most feature that contradicts the proposed sdhash scheme. According to the proposed sdhash scheme [13], the left most feature with lowest precedence rank should get selected. In order to mitigate this bug, line numbers 14-15 & 26-27 should be removed from the code. We corrected the above mentioned bugs in the provided sdhash version 3.4[3], and the same corrected code is used to carry out experiments stated in § 6.

In the same work [34], the authors have indicated that undiscovered modifications to the input of sdhash are possible. However, details of how to achieve this are not provided. An undiscovered modification means that input can be modified without influencing the final sdhash digest. Therefore two or more files can generate same sdhash digest, which is called collision in terms of cryptographic hash functions and such files are called **'colliding files'** in rest of the chapter.

Collision detection violates one of the basic properties of similarity preserving hash functions called 'Coverage' [33]. Every byte of input is expected to influence the final sdhash digest. Breitinger et al. [33] have statistically shown that 20% of the input bytes are not part of any selected feature. So, these bytes are not expected to influence the final sdhash digest, and are referred to as **'gap'**. Table 3.1 shows statistics of both

[3]http://roussev.net/sdhash/sdhash.html

Table 3.1: Different statistic on sdhash from (Breitinger & Baiber, 2012)

	Average	improved	original
1	filesize*	428,912	428,912
2	gaps count	2888	2889
3	min_gap*	1.09	1.076
4	max_gap*	1834	1834
5	avg_gap*	33.46	34.27
6	ratio to file size	20.65%	21.21%

original sdhash code and improved code (after correcting the bugs discussed above).

```
void
sdbf::gen_chunk_scores( const uint16_t *chunk_ranks, const uint64_t chunk_size
    , uint16_t *chunk_scores, int32_t *score_histo) {
uint64_t i, j;
uint32_t pop_win = config->pop_win_size;
uint64_t min_pos = 0;
uint16_t min_rank = chunk_ranks[min_pos];

memset( chunk_scores, 0, chunk_size*sizeof( uint16_t));
if (chunk_size > pop_win) {
for( i=0; i<chunk_size-pop_win; i++) {
// try sliding on the cheap
if( i>0 && min_rank>0) {
while( chunk_ranks[i+pop_win] >= min_rank && i<min_pos && i<chunk_size-pop_win
    +1) {
if( chunk_ranks[i+pop_win] == min_rank)
min_pos = i+pop_win;
chunk_scores[min_pos]++;
i++;
}
}
min_pos = i;
min_rank = chunk_ranks[min_pos];
for( j=i+1; j<i+pop_win; j++) {
if( chunk_ranks[j] < min_rank && chunk_ranks[j]) {
min_rank = chunk_ranks[j];
min_pos = j;
} else if( min_pos == j-1 && chunk_ranks[j] == min_rank) {
min_pos = j;
}
}
if( chunk_ranks[min_pos] > 0) {
chunk_scores[min_pos]++;
}
}
// Generate score histogram (for b-sdbf signatures)
if( score_histo) {
for( i=0; i<chunk_size-pop_win; i++)
score_histo[chunk_scores[i]]++;
}
}
}
```

Listing 3.1: sdfb_core.cc from sdhash-3.4

3.4 Our Contribution

The purpose of fuzzy hashing or similarity hashing schemes is to filter similar or correlated files corresponding to a given file that an investigator needs to examine. These schemes reduce the search space and corresponding manual effort of analysis for the investigator. The process of filtering the files by matching them with a set of already known to be bad files is called **Blacklisting**.

We propose a scheme that can generate multiple similar files corresponding to a given file with a similarity score of 100 for the sdhash similarity hashing. The scheme shows a weakness of the sdhash algorithm that an attacker could exploit to confuse and delay the investigative process. An example attack scenario is explained in the following paragraph.

Let us suppose a scenario where a suspected person 'X' has accessed and downloaded some proprietary images from a commercial website 'A' while she is logged in as a registered user. X, as an anonymous owner, runs a parallel website 'B' that hosts content from the original website available for free from a hosting location in some different part of the world. She intends to popularize her website 'B' to get a large viewership that might attract web advertisers to put their ads on the website. A consistent viewership over a period would result in high chances of advertisement hits and consequently monetary returns for X. She would recover the membership cost gradually while the rest of the revenue is profit.

The original website 'A' eventually comes to know about the existence of website 'B' which is hosting their proprietary content. Since the owner of the domain name is registered as anonymous on records, the only way to track her is her IP address. Fortunately, the country where the website is hosted follows anti-piracy and Intellectual property protection laws. The physical location of systems on which the data of website 'B' is stored can be determined. X uploads content downloaded from original website after putting a watermark of his own website on each image. The use of cryptographic hash functions is ruled out in that case and investigators would need a similarity digest algorithm, possibly sdhash to find the files.

Here, in this condition if X has any time to prepare herself for such an investigation, she could use our tool to generate multiple similar files, with same metadata, corresponding to each file. The approach is definitely heavy on storage but can help X in increasing the effort of the investigation by forcing the investigators to analyze the files manually. Secondly, the investigation process could also be confused as by X's claim that she is innocent and it is a work of someone else who has access to her system or even a malware. In both the cases, investigation effort is increased many folds. Moreover, the primary purpose of a similarity digest to help investigators quickly filter out files of interest is defeated.

Breitinger et al. in [34] mentioned that 20% of the input data can be modify without influencing the final sdhash digest. We used two approaches to verify the number of undiscovered modification within gaps. These are (1) Random modification and (2) Deliberate Modification.

In the random modification approach, gap bytes are filled with randomly chosen ASCII characters. Our experiments on text files show that random modification of only 2% of the gap bytes influences the sdhash digest with probability close to 1. In the second approach of 'Deliberate modification' we propose an algorithm for careful modifications in order to increase the available bytes for modification within gaps. Experimental analysis of the proposed algorithm shows that by using this algorithm, around 12% of the gap bytes can be modified with maximal similarity score of 100.

3.4.1 Random Modification

We randomly choose several byte positions within the gap and modify each with a randomly chosen ASCII character to find the maximum number of random modifications within the gap that do not influence the sdhash digest of the entire document. We performed experiments on a data set of 50 text files of variable size from the T5-corpus dataset. We found that even one byte of random modification within the gap would influence the sdhash digest with an average probability of 0.22, and the modification of all bytes in the 20% gap will impact the final hash digest with probability 1. So, we focused on finding the minimum number of modifications that would influence the final sdhash digest with probability 1.

We started with single byte modifications and generated more

Table 3.2: Minimum number of random modification, that modifies final sdhash digest with probability 1.

S.No.	File size	Gap	Random Modification		
	(In KBs)	(In Bytes)	Bytes	Gap%	File%
1†	1.5	354	45	12.70%	3%
2	22.9	3948	50	1.26%	0.21%
3	50	8917	70	0.78%	0.14%
4	81	14084	60	0.42%	0.07%
5	307	46296	80	0.17%	0.03%
6	841	215894	20	0.01%	0.00%
7	1095	139038	50	0.04%	0.00%
8	1554	378636	35	0.01%	0.00%
On an avg.			51.25	1.92%	0.42%

† This file is not from T5-corpus database

than 5000 files with only one byte tampering and evaluated its influence the hash digest.

We gradually increased number of modifications until the hashes for all 5000 files got influenced. It was found that with a random modification of only around 2% bytes of the gap there is an influence on the sdhash digest of each of the randomly generated file which is on an average 0.42% of the respective file size. Experimental results for a small sample of 8 files is given in table 3.2.

As described in § 3, only the selected features (statistically improbable features) participate in the generation of final similarity digest. Therefore gaps (the data bytes which are not part of any selected feature) are expected not to influence the final hash digest. However, as we showed in the experiments, these bytes do influence the sdhash digest. This happens since each feature in the sdhash construction is highly correlated to its neighbors. Each feature differs from its left and right neighbor by only one byte. For example, let D be a data object under investigation which has the following byte sequence and features.

$$\begin{array}{ll}
 & B_0B_1B_2B_3B_4B_5.\ .\ .\ .\ .\ B_{63}B_{64}B_{65}B_{66}B_{67}.\ .\ .B_N \\
f_0 & \boxed{B_0B_1B_2B_3B_4B_5.\ .\ .\ .\ .\ B_{63}}B_{64}B_{65}B_{66}B_{67}.\ .\ .B_N \\
f_1 & \quad\boxed{B_1B_2B_3B_4B_5B_6\ .\ .\ .\ .\ .\ B_{64}}B_{65}B_{66}B_{67}.\ .\ .B_N \\
f_2 & \qquad\boxed{B_2B_3B_4B_5B_6B_7\ .\ .\ .\ .\ .\ .\ B_{65}}B_{66}B_{67}.\ .\ .B_N \\
 & \qquad\qquad\qquad . \\
 & \qquad\qquad\qquad . \\
 & \qquad\qquad\qquad . \\
f_n & \qquad\qquad\boxed{B_{N-63}B_{N-62}.\ .\ .\ .\ .\ .\ .B_{N-2}B_{N-1}B_N}
\end{array}$$

where N is the number of bytes in the data object D, and n is the number of features in D (n=N-L+1). Each byte is part of atleast one and at-most L (i.e. 64) features. Each byte (B_k), except the first L-1 and the last L-1 bytes (L≤k≤N-L+1), is part of exactly L features. Change in any byte, B_k will reflect in a change in features f_k to f_{k-L+1}, which may lead to a change in the precedence ranks $R_{prec,D}(f_{k-L+1})$ to $R_{prec,D}(f_k)$. A change in the rank of any feature($R_{prec,D}(f_k)$) will reflect in a change in the popularity score of features of D, which may affect the list of selected features. Any modification in the list of selected features will lead to changes in the final hash digest.

3.4.2 Deliberate Modification

The experiment results from § 3.4.1 show that the entire 20% gap of any file cannot be modified by random modification. We now propose an algorithm that performs careful modifications in order to increase the number of changes within the gaps while still ensuring no change in the similarity digest.

3.4.2.1 Algorithm Description

As discussed in § 3.4.1, modification in any byte B_k will influence the rank of all features containing B_k. This might cause changes in the list of selected statically improbable features. In the sdhash construction, a feature with leftmost lowest rank gets selected in a popularity window. If the rank of a feature is leftmost lowest in t or more than t (threshold) popularity windows then it gets selected as a statistically improbable feature. These selected statistically improbable features participate in the computation of the final sdhash digest. Let D be a data object with f_{S_1} and f_{S_2} as two consecutive statistically improbable features.

$$\begin{array}{l}
f_0\ f_1\ f_2.\ .\ .\enclose{circle}{f_{S_1}} f_{S_1+1}\ .\ .\ .f_{S_1+63}\ f_{S_1+64}.\ .\ .f_{S_2-1}\enclose{circle}{f_{S_2}}.\ .\ .f_n \\
B_0B_1B_2.\ .\ B_{S_1}B_{S_1+1}.\ .\ B_{S_1+63}\boxed{B_{S_1+64}.\ .B_{S_2-1}}B_{S_2}.\ .B_n.\ .\ .B_N
\end{array}$$

$$\begin{array}{rl}
\text{where } f_{s_1} : & B_{s_1}B_{s_1+1}B_{s_1+2}.\ .\ .\ .\ .B_{S_1+L+1} \\
f_{s_2} : & B_{s_2}B_{s_2+1}B_{s_2+2}.\ .\ .\ .\ .B_{S_2+L+1}
\end{array}$$

Data bytes B_{S_1+64} to B_{S_2-1} are not a part of any selected features. The aim is to modify these bytes in such a way that modified features never get selected over f_{S_1} and f_{S_2}. For every data byte B_k, where $S_1+L \le k \le S_2-1$, a specific value among all possible ASCII characters satisfying the following two conditions is chosen:

1. $R_{prec,D}(f_j^{'}) > R_{prec,D}(f_{S_2})$ AND $R_{prec,D}(f_j^{'}) \ge R_{prec,D}(f_{S_1})$
2. $R_{prec,D}(f_j^{'}) \ge R_{prec,D}(f_j)$

where $(k\text{-}L+1) \le j \le k$ and $(S_1+L) \le k \le S_2\text{-}1$ and $f_j^{'}$ is modified feature f_j obtained as the result modification of byte B_k. The above two conditions ensure that all the modified features

f'_j have rank $R_{prec,D}(f'_j)$ greater than the rank of the right selected statistically improbable feature ($j<S_2$) i.e. $R_{prec,D}(f_{S_2})$. At the same time, $R_{prec,D}(f'_j)$ is greater than or equal to the rank of the left selected ($j>S_1$) statistically improbable feature, i.e. $R_{prec,D}(f_{S_1})$. It can be equal to this value because even if two features have equal rank, the left most feature always gets selected. Ultimately, no other feature gets selected over both the statistically improbable features.

The above mentioned conditions are not enough if $(S_2\text{-}1)\text{-}(S_1+L) \geq t$, where L is the feature length and t is the threshold. Even if each modification satisfies both the conditions, still new features may get selected. The reason this happens is that if the distance between two selected features is more than L+t, then after modification, the rank of some modified features may become local minimum among their t or more neighbors. Since t is the threshold for a feature to get selected, it may get selected as a statistically improbable feature and hence may influence the final sdhash digest. In the case mentioned above, it needs to be verified that no modification causes any change in the list of selected statistically improbable features. To mitigate this problem, after modification of the gaps bytes being considered, the popularity score($R_{pop,D}$) of all the features of D is calculated. If any new feature, f'_j contains the popularity score $R_{pop,D}(f_j) > t$ then all the previous modifications are discarded. Similarly the gaps between each adjacent pair of selected improbable features are modified.

Algorithm 1 and 2 (presented in appendix A) cab be used to generate multiple colliding files corresponding to a given data object with maximal similarity score. Each execution of algorithm1 produces a different file with dissimilar modification and different number of modifications. Therefore, we can generate G^{256} different files with maximal similarity corresponding to a given file, where G denotes the total number of gap bytes in the data object. The attacker can easily confuse the investigator by generating a huge number of files corresponding to a malicious or desired file. Since our current implementation is focused on text files, so we have chosen the characters only from the set of 95 printable ASCII characters, starting from char 32 till char 126. The maximum number of files that can be generated are G^{95}, which is sufficiently large even for $G = 2$.

We ran the proposed algorithms for the same data set of 50 text files which were used for our earlier random experiment. We found that around 12% of the gap bytes can be modified with maximal similarity score of 100 using the proposed algorithm. This is a huge improvement over the random modification case when even 2% of the gap bytes cannot be modified without changing the final sdhash digest. Experimental results for a small sample of 8 files are presented in table 3.3.

3.5 Countermeasures

In order to reduce the amount of undiscovered modifications, we propose the following two mitigations.

3.5.1 Minimization of popularity score threshold

Decrease in the threshold of popularity score in selection of statistically improbable features will increase the number of selected features. This, in turn, will result in the reduction of gap

Table 3.3: Number of modification with maximal similarity score through proposed algorithm

S.No.	File size (In KBs)	Gap (In Bytes)	Deliberate Modification		
			Bytes	Gap%	File%
1†	1.5	354	89	25.14%	5.99%
2	22.9	3948	552	13.98%	2.41%
3	50	8917	1065	11.94%	2.13%
4	81	14084	1273	9.03%	1.50%
5	307	46296	3357	7%	1.09%
6	841	215894	31371	14%	3.73%
7	1095	139038	6211	4%	0.56%
8	1554	378636	13787	3.60%	0.88%
On an avg.			7185.25	11.08%	2.28%

† This file is not from T5-corpus database

bytes that could be modified without affecting the final sdhash digest.

3.5.2 Bit level feature formation

In the sdhash scheme, each feature differs from its neighboring features by one byte. Therefore, the attacker has 2^8 possible choices to modify the feature without influencing its neighboring feature. If each neighboring feature differs by only 1 bit (in place of the original one byte), it will reduce the number of possible choices with the attacker from 256 to 2. Hence the probability of modifying each bit without affecting the final hash will also get reduced substantially. However, it will increase the number of features and hence the selected features, thereby causing some loss in efficiency. Increase in the number of selected improbable features will not only increase the computation time, it will also cause an increase in the size of the final sdhash digest.

3.6 Summary

Currently sdhash is one of the most widely used byte-wise similarity hashing scheme. It is possible to do undiscovered modification to a file and yet obtain exactly the same sdhash digest. We have proposed a novel approach to do maximum number of byte modification with maximal similarity score of 100. We also provided a method to do an anti-forensic attack in order to confuse or delay the investigation process.

Chapter 4

Security Analysis of mvHash-B Similarity Hashing

mvHash-B, proposed by Breitinger et al. [5] in 2013, is one of the most well known fuzzy hashing schemes. The runtime complexity of the mvHash-B scheme is almost equivalent to cryptographic hash function SHA-1, which makes it fastest among the existing approximate matching schemes. Moreover, the length of the mvHash-B similarity digest is just 0.5% of the input length. Both of the above desirable features make it one of the most prominent approximate matching scheme.

The work [5] claims that mvHash-B is sufficiently robust against active manipulations. In this work, we propose an anti-forensic attack on mvHash-B similarity hashing. We develop an algorithm that can be used to circumvent the blacklisting based filtering of the mvHash-B scheme, i.e. it is possible to hide a malicious file from the blacklisting process of mvHash-B similarity hashing. This work shows that less than 0.03 % deliberate modifications can take down the similarity score of a file from 100 to less than 6 without influencing the file semantically and visually.

Our attack can also be used to carry out an anti-forensic mechanism that defeats the very purpose of the approximate matching scheme by hiding a malicious file from the filtering process. An attacker could modify the desired file without changing its semantics and visual meaning. When an investigator tries to filter the desired malicious file using mvHash-B similarity preserving hashing from the hard disk of a suspect, it will not appear in the filtered output.

Finally, we also propose an improvement to the mvHash-B construction in order to prevent our attack. The minor tweak we propose to the scheme ensures the security of the modified mvHash against an active adversary.

The rest of the chapter is organized as follows: Notations and definitions used in the paper are provided in § 4.1. The mvHash-B scheme is explained in § 4.2 . § 4.3 contains our analysis and attack on mvHash-B, followed by our proposed attack against the scheme. Experimental results on text and image files validating our attack are presented in § 4.4. Finally, we conclude the paper in § 4.5 and § 4.6 by proposing solutions to mitigate our attack on mvHash-B.

4.1 Notations

- BS denotes input byte sequence i.e. BS= $B_0B_1B_2.....B_{L-1}$ where B_i represents the ith byte of input and L denotes the length of the input file in bytes

- $N_{k,n}$ denotes the n-neighborhood of input byte B_k $N_{k,n} = B_{k-\frac{n}{2}}B_{k-\frac{n}{2}+1}....B_{k-1}B_kB_{k+1}.....B_{k+\frac{n}{2}-1}B_{k+\frac{n}{2}}$ where n denotes size of neighborhood and n is always even.

- bitcount($N_{k,n}$) denotes the function that outputs number of bits set to 1 in binary representation of $N_{k,n}$
- t denotes the threshold
- ib denotes average number of influencing bits for one byte.

4.2 Description of mvHash-B

mvHash-B works in following three phases:

1. **Majority Votes:** The idea of this phase is to convert each input byte into 0x00 or 0xFF in order to compress the input in subsequent phases. mvHash-B counts number of bits set to one for n-neighboring of each input byte i.e. bitcount($N_{k,n}$) for $0 \leqslant k \leqslant (L-1)$. If bitcount($N_{k,n}$)$\geqslant$ t(threshold), then the value majority vote of the byte B_k is set to 0xFF else 0x00. The value threshold t is calculated as follows:

   $$t = \frac{(n+1) \cdot ib}{2}$$

   where ib denotes average number of influencing bits for one byte($0 \leq ib \leq 8$), default value of ib=8.

2. **Encoding the majority vote bit sequence with RLE:** Run length encoding(RLE) [39] is a data compression algorithm. Number of the identical subsequent byte is called 'Sun-count'. The output of run length encoding is the sequence of run-counts as shown in Fig 4.1 and denoted as RLE. mvHash-B assumes that each RLE starts with number of identical 0x00 bytes, therefore, if the majority vote

of input starts with 0xFF then run-length-encoding keeps 0 in the beginning.

3. **Fingerprint generation using Bloom filters:** mvHash-B stores the resultant RLE sequence in bloom filter. The reason behind using bloom filter is its efficient comparison capability. Bloom filter is an array of m elements with all elements initialized to 0. In mvHash-B implementation m=2018(2^{11}). Select first 11 bytes of RLE sequence to built a group and perform mod 2 of each RLE element of this group. Mod 2 operation transforms the RLE sequence into 11 bit sequence of 1 or 0 i.e. $b_0b_1b_2. . . b_{10}$ where $b_i \epsilon$ 0,1 and $0 \leq i \leq 10$. This is further divided into two parts:

   - $v_1 = b_{10}b_9b_8b_7b_6b_5b_4b_3$ is used to identify the byte within the Bloom filter and
   - $v_2 = b_2b_1b_0$ is used to identify the bit within the byte.

   Identified bit position of bloom filter is set to 1. Next group can not be consecutive in order to ensure alignment robustness. Next group starts will the alternate element of RLE sequence. This process is explained in detailed in Fig 4.1. Same process is applied to till the last element of RLE sequence.

   In order to find similarity between two bloom filters, a distance score is calculated which is represented as di_{score}. di_{score} computation uses the hamming distance. Let bf_1 and bf_2 are two bloom filters, the value of di_{score} is calculated as follows:

$$di_{score} = \frac{hd(bf_1,bf_2)}{|bf_1|+|bf_1|} \cdot 100$$

where hd(bf_1, bf_2) denotes the hamming distance between bf_1 & bf_2 and $|\mathrm{bf}_i|$ denotes the number of bits set to 1 in bloom filter bf_i. Value of di_{score} ranges from 0 to 100. Where 0 indicates the exact similarity(100% similarity) and 100 indicates zero similarity.

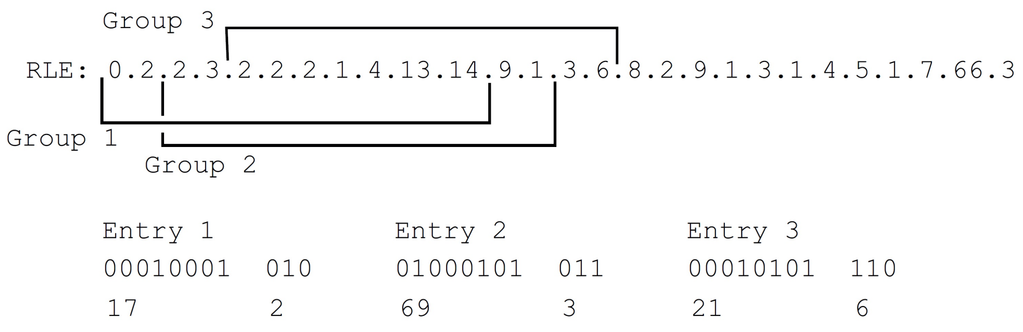


Figure 4.1: Processing of RLE encoding by mvHash-B from [5] .

4.3 Anti-BlackListing attack on mvHash-B

We present an anti-forensic attack based on mvHash-B blacklisting i.e. we have shown that it is possible to circumvent mvHash-B blacklist filtering. We have provided an algorithm/tool that can be used to avoid the automated detection of the blacklisted or malicious files by the blacklisting process of mvHash-B similarity hashing. Blacklisting is the process of filtering out the files by matching it with the set of already Known-to-be-bad files. The resultant file of the blacklisting process is similar to known-to-be-bad or malicious files and need to be examined manually. The proposed algorithm modifies the target file without changing the semantics of the file in a way so that it does not appear in the list of blacklisted files. These kind of attacks are termed as an 'Anti-Blacklisting attack' in literature by baier et al. [20].

The proposed Anti-Blacklisting attack shows that mvHash-B scheme does not withstand an active adversary against a blacklist. This paper follows the definition of anti-blacklisting provided by Baier et al. in their paper [20]. Proposed attack generates false negatives for mvHash-B similarity results. Let F_1 is a malicious file. The malicious user or attacker can use the proposed algorithm and generate file F_2, which is semantically and perceptually same as F_1. However, mvHash-B similarity hashing results F_2 as non-similar to F_1, thus a false negative.

As discussed in § 4 mvHash-B works in three phases 1)Majority Votes 2)Encoding the majority vote bit sequence with RLE 3) Fingerprint generation using Bloom filters, where the first phase majority votes converts each input byte into 0x00 or 0xFF then the next phase Run length encoding transforms the input into run-count of the identical subsequent bytes (0x00 or 0xFF) and finally the last phase performs a modulo 2 operation on the resultant RLE sequence and stores the resultant RLE sequence in bloom filter as shown in Fig. 4.1. Each consecutive 11 elements of RLE-sequence forms a group. Each group sets one bit in bloom filter. Fig.4.2 shows an example.

The key idea of the attack is by modifying one element in each group, we can change the position of all the set bits in bloom filter. Since the bloom filter represents the final mvHash-B digest hence the entire digest is modified. Each element of a group is modulo 2 representation of corresponding RLE elements, therefore, it is either 1 or 0. We need to flip any one bit among all 11 bits of a group. Which requires to increase or decrease any of the corresponding 11 RLE-elements just by 1. Any modification in the RLE sequence requires replacement of the corresponding

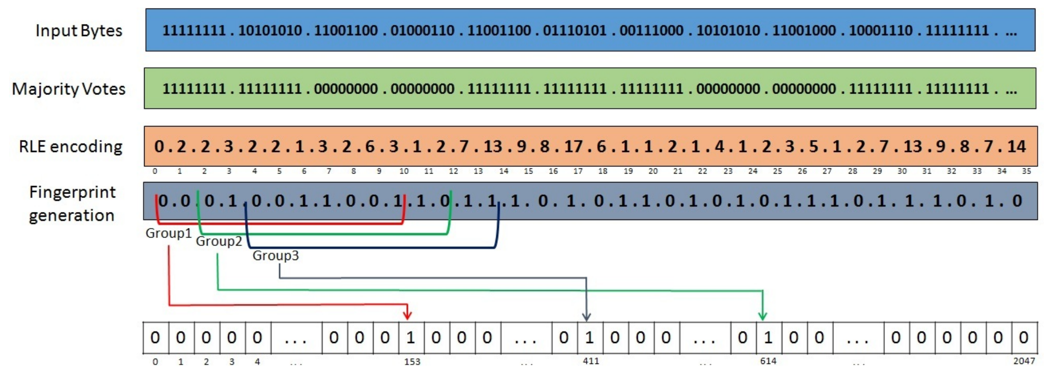


Figure 4.2: mvHash-B similarity digest generation considering n=2

input byte with the byte containing more or less number of 1's (in its binary representation). Now the important question is, how to modify RLE-encoding without influencing the semantics of the file. Algorithm 3 (shown in appendix B) presents a way of making such modifications. Algorithm3 works in following steps:

1. Go through each byte of the input file. Check if the number of bits set to 1 in the n-neighborhood(bitcount($N_{k,n}$)) of the current byte(B_k) is equal to t or (t-1), where k is the current byte index and t is the threshold. If yes, proceed with following steps:

   (a) Modify the B_k with semantically similar character (defined later in this section). Let $B_k^{'}$ denotes modification on B_k and similarly $N_{k,n}^{'}$ denotes modification on $N_{k,n}$. Check for following condition:

      i. bitcount($N_{k,n}^{'}$)<bitcount($N_{k,n}$) and bitcount($N_{k,n}$)=t

      ii. bitcount($N_{k,n}^{'}$)>bitcount($N_{k,n}$) and bitcount($N_{k,n}$)=(t-1)

    If any of the above condition satisfies then accept the modification else revert the changes.

    (b) If the modification happens at step (a) it will change the majority vote of the corresponding byte from 0x00 to 0xFF or vice versa.

    (c) Any change in majority vote will reflect modification in corresponding RLE element since it is the count of consecutive 0x00 or 0xFF.

    (d) Any alteration in RLE sequence will modify the index of bloom filter element addressed by the group containing the modified RLE element. Fig. 4.3 shows an example of one byte modification. One RLE modification may effect several groups.

2. Perform step 1 after the last byte of the last modified group.
3. Repeat all the above steps till the last byte of the input file.

The semantically and perceptually similar alteration can be performed as follows:

- For text documents:
    1. Lower case to upper case conversion or vice-versa
    2. Space to tab or tab to space, etc
- For the image documents:
  The modification can be performed by doing a minor change in the RGB value of a pixel. For example, in bmp (RGB32) format each pixel in an indexed color image is described by 3 bytes, representing its RGB (Red-Green-Blue)

value. This RGB value is the index of single color described by the color table. Altering the least significant bit of any of these three bytes does not produce a visual change in the image.

- For a program file:
  1. changing the names of variables
  2. writing looping constructs in a different way
  3. adding comments, etc.

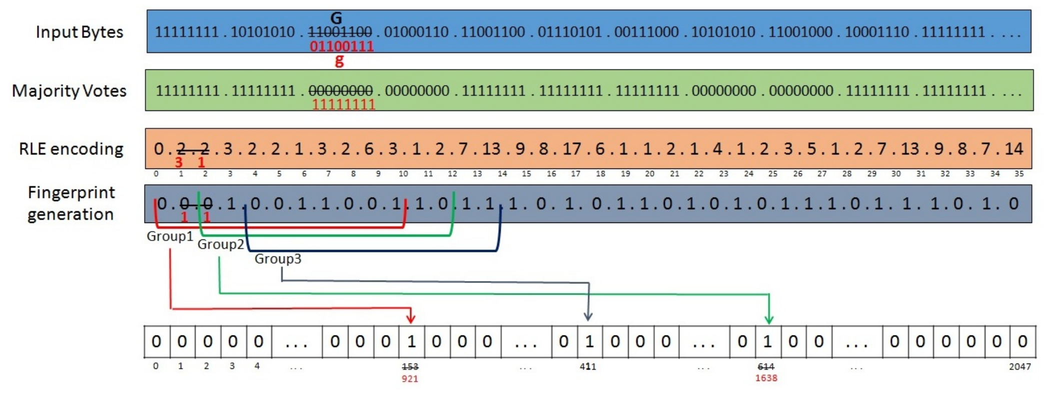


Figure 4.3: Example of one byte modification

One byte modification in a group is enough. Since a modulo 2 operation is performed on the RLE elements so addition or subtraction of one byte will change the position of the addressed bit in bloom filter. Each group differs from its neighboring group only by two elements. Therefore one modification influences several groups. Fig. 4.4 contains an example which shows that for a 174 byte long document, just 2 input byte modifications are enough to change the entire mvHash-B digest of te document. The value of n considered in te example explained

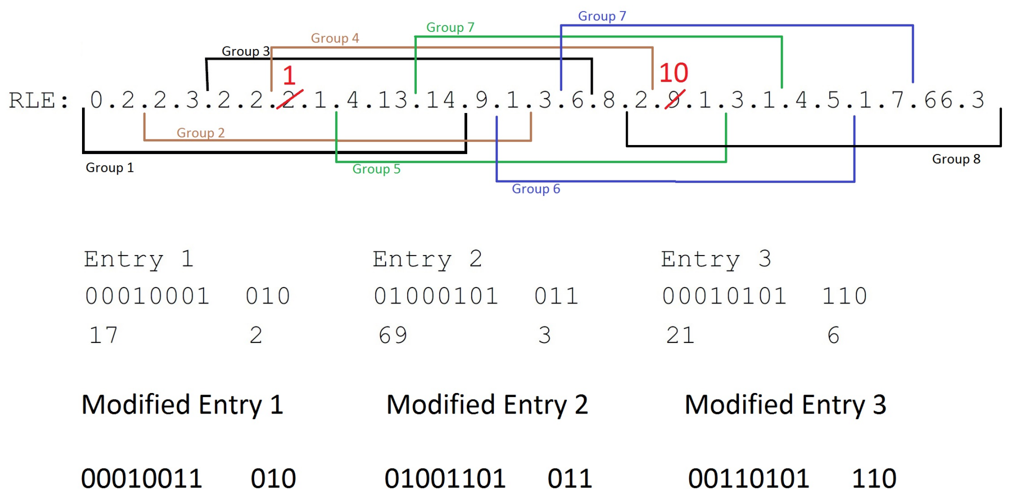


Figure 4.4: Example illustrating number of Deliberate Modifications required in order perform anti-blacklisting attack

in Fig. 4.4 is 2, where as recommended value of n by design of mvHash-B scheme is 50 or 20 depending on the file type. If the value of neighborhood(n) is higher then number of modifications required are smaller. For example if n is 50 then one byte modification will impact majority vote calculation of 50 neighboring bytes.

4.4 Results

We performed two experiments: One on the textual data(text documents) and other on visual data(images).

4.4.1 Experiment 1

Our first experiment was performed on a dataset of 50 text files of variable sizes from the T5-corpus dataset[1]. As recommended

[1]`http://roussev.net/t5/t5.html`

in [5], we took parameters n and ib to be 50 and 7, respectively. The results obtained from the experiment show that merely 3% deliberate modification in the file takes down the similarity score from 100 to 4 (on an average), whereas the modified file is semantically similar to the original file. Table 4.1 shows the experimental results of our proposed anti-blacklisting attack for a small sample of 10 text files.

4.4.2 Experiment 2

We performed a second set of experiments on a dataset of 200 bitmap images of variable sizes. We took images from various publicly available datasets such as Microsoft Windows Bitmap Sample Files [2], CVonline: Image Databases [3] and Yokogawa Y-Link [4]). The value of input parameter n was chosen to be 50 and ib was taken to be 8. This is as per the suggestion in [5].

Table 4.2 illustrates the experimental results of our proposed attack on a sample set of 10 bitmap images. Observed results demonstrate that by varying as little as 0.3% of the original image bytes, the similarity score of mvHash-B digest for the image reduces from 100 to 6 on an average. Moreover, the resultant modified images are visually same as the original image. Fig. 4.5 shows an image and a similar image obtained by our attack algorithm. The left image is the unmodified source RAY.BMP, taken from Microsoft Windows Bitmap Sample Files [5] while the image on the right is the modified image generated by our algo-

[2]http://www.fileformat.info/format/bmp/sample/

[3]http://homepages.inf.ed.ac.uk/rbf/CVonline/Imagedbase.htm

[4]https://y-link.yokogawa.com/YL008/?V_ope_type=Show&LANG=EN&Language_id=EN&Login_type=2&Download_id=DL00002164

[5]http://www.fileformat.info/format/bmp/sample/1d71eff930af4773a836a32229fde106/view.htm

rithm. Visual similarity between these two files can be seen to be very high. However, the mvHash-B similarity score for these images is 0.

An attacker can apply the proposed algorithm on a malicious image and can generate a modified image, which is visually same as the original malicious image but can not be detectable from mvHash-B similarity hashing. Therefore an attacker can easily hide the malicious image/text file from the mvHash-B filtering process defeating the very purpose of approximate matching algorithms.

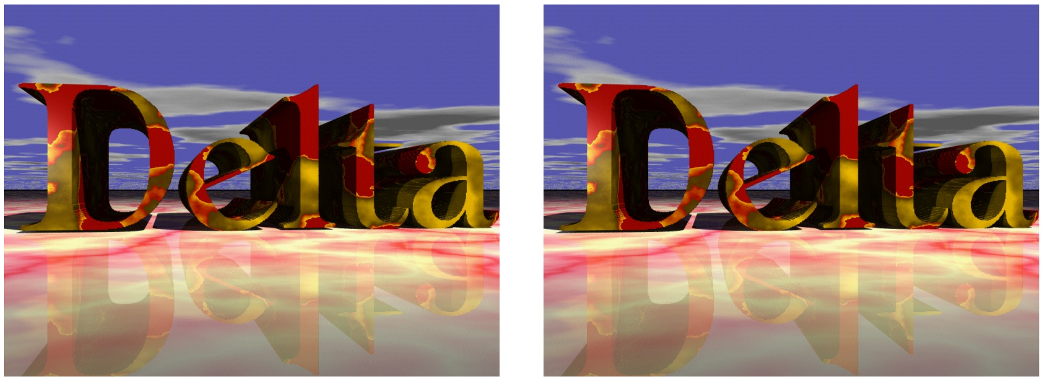


Figure 4.5: Example: First image from the left is original image taken from Microsoft Windows Bitmap Sample Files[7] and the other image is the generated modified image from the proposed algorithm; mvHash-B similarity score of above images is 0

4.5 Countermeasures

In order to prevent the proposed attack, we suggest some improvement in the design of mvHash-B construction. The root cause of the attack is that attacker has the ability to identify the

Table 4.1: Experimental results obtained from the proposed attack technique on Text file

S.No.	File Name	File size (In Kilo Bytes)	Number of Modification (in Kilo Bytes)	Similarity Score
1†	Test_02.text	0.47	0.01	0
2	Test_4955.text	14	0.40	0
3	Test_4950.text	23	0.61	0
4	Test_4954.text	27	0.74	0
5	Test_4956.text	30	0.91	0
6	Test_3518.text	34	0.41	0
7	Test_4960.text	47	1.20	7
8	Test_4953.text	76	2.00	14
9	Test_4953.text	189	6.00	9
10	Test_4953.text	190	4.00	10
On an avg.			2.59%	4

† This file is not from T5-corpus database

Table 4.2: Experimental results obtained from the proposed attack technique on bitmap images

S.No.	File Name	File size (In Kilo Bytes)	Number of Modification (in Kilo Bytes)	Similarity Score
1	Air Conditioner S.bmp	6	0.01	0
2	Pressure Transmitter 03 M.bmp	20	0.45	0
3	Control Valve L.bmp	55	0.028	0
4	DadWood.bmp	265	0.480	0
5	Edison.bmp	500	1.838	8
6	carsgraz_287.bmp	901	2.383	10
7	Ray.bmp	1407	2.14	0
8	Alex bit.bmp	3984	16.682	13
9	MARBLES.bmp	4165	13.424	16
10	12x12 Women sample 400 dpi.bmp	22502	42.244	16
On an avg.			0.228%	6.3

† This file is not from T5-corpus database

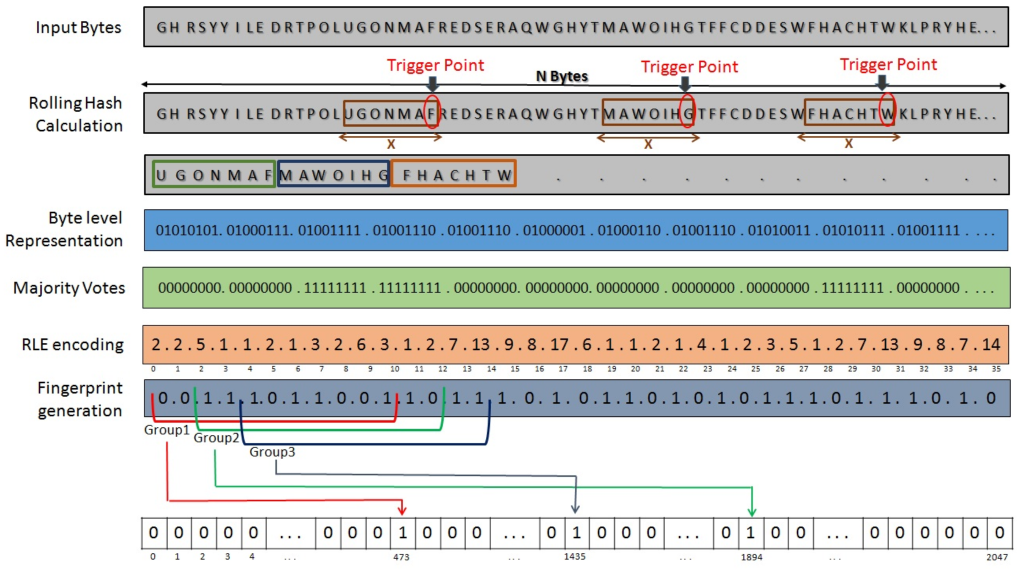


Figure 4.6: Counter measure for the proposed attack

position of input byte that he can modified with the maximum influence over the mvHash digest. We want to restrict this liberty by adding two secret input parameter called 'trigger' and 'x' to the scheme. The investigator can choose any value of these two parameter during mvHash digest computation.

Let T be the trigger value chosen by the investigator. Fig. 4.6 explains our suggested improvement to mvHash-B scheme. Before Majority vote calculation rolling hash over the input byte is calculated. The rolling hash calculation is be done in same way as explained by Kornblum in Context Triggered Piecewise Hashing [1]. The value of rolling hash depends only on last s bytes of the input file. Let R_i denotes the rolling hash of ith input byte.

R_i=Rolling-Hash(B_i,B_{i-1},B_{i-2},. . .,B_{i-s})

where B_i represents the i^{th} input byte. If R_i== -1 mod T for s$\leq$i$<$ n where n denotes input size. i^{th} byte position is called trigger point. Select last x bytes from the trigger point for further digest calculation as shown in Fig. 4.6. The number of trigger points is inversely proportional to T [20]. Thus, for the higher value of T number of trigger points will be smaller or vice-versa. An investigator can choose the value of T and x based on input file size and these values are unknown to the attacker. The attacker does not know selected input byte. Therefore, cannot perform the deliberate intelligent modification. Random modification also does not impact the mvHash-B digest much because every input byte is not taking part in final digest calculation. Since the attacker is unaware of trigger positions, thus cannot perform random insertion/deletion as well.

4.6 Summary

This chapter explored the weakness of mvHash-B similarity digest scheme, which can be exploited by an active adversary to defeat the purpose of the approximate matching scheme. We showed an Anti-Blacklisting attack on mvHash-B similarity digest and practically proved that mvHash-B does not withstand an active adversary against blacklist. We provided an anti-forensic tool that can be used by an adversary to bypass the blacklist filtering process of mvHash-B similarity digest. Additionally, we suggested an improvement to mvHash design to conquer the proposed attack. Furthermore the proposed improvement ensures the security of scheme against active adversary.

Chapter 5

FbHash: A New Similarity Hashing Scheme for Digital Forensics

We present a new approximate matching scheme which is secure against active attacks. We term our scheme as `FbHash` -Frequency Based Hashing. The idea of `FbHash` is based on the TF-IDF (Term Frequency - Inverse Document Frequency) concept of information retrieval [61]. TF-IDF is a statistical measure used to evaluate the importance of a word to a document in a collection or corpus. `FbHash` uses this notion to identify important fragments (features) of a document. A file fragment's contribution to the final similarity score is based on its importance or relevance as per this measure.

We also provide a comprehensive comparative analysis of `FbHash` with other prominent approximate matching approaches i.e., `ssdeep` and `sdhash`. We show that `FbHash` detects similarity with 28 % higher accuracy for uncompressed file formats (i.e., text files) and around 50 % higher accuracy for compressed file formats (i.e., docx). We also show that our proposed scheme is able to correlate a fragment as small as 1 % to its source file with 100 % detection rate and able to detect commonality as small as 1 % between two documents with correct appropriate (low) similarity score and 100 % detection rate. Further, our scheme also produces the least false negatives in comparison to other schemes.

We also observe that measuring similarity only at the byte-level does not allow a good match for compressed file format documents. Hence, we present two versions of our tool:

- `FbHash-B.` This version of our tool measures similarity at the byte-level. We show in section 5.5 that it can detect similarity with 98 % accuracy for uncompressed file formats.

- `FbHash-S.` This version performs *Syntactic matching* and uses information about the internal structure of a document in order to measure similarity. This is recommended for compressed file formats.

Finally, we also provide security analysis of our scheme and show that `FbHash` is resistant against active adversary attacks.

The rest of the chapter is organized as follows: In Section 5.1, we present our scheme `FbHash` and its variant `FbHash-B` that works for uncompressed file formats. In Section 5.2, we show how our scheme generates the final hash to calculate a similarity score. This is followed by Section 5.3, in which we present our `FbHash-S` scheme for finding similarity in compressed file formats. Section 5.4 presents the security analysis of `FbHash` followed by comparative analysis with other existing schemes in Section 5.5. Finally, we summarize this chapter in Section 5.6.

5.1 Construction of `FbHash` (`FbHash-B`) Similarity Hashing Scheme

To facilitate better understanding of our scheme, we first define some important terms and notations that are used throughout the paper.

5.1.1 Notation and Terminology

- Chunk : Sequence of k consecutive bytes of a document.
- ch_i^D : represents the i^{th} chunk of a document D.
- Chunk Frequency : Number of times a chunk ch_i appears in a document D. Represented as $\mathrm{chf}_{ch_i}^D$.
- Document Frequency : The number of documents that contain chunk ch. Represented as df_{ch}.
- N : denotes the total number of documents in the document corpus.
- $\mathrm{RollingHash}(\mathrm{ch}_i)$: Rolling hash value of ch_i
- ch-wght_{ch_i} : Chunk weight of ch_i
- doc-wght_{ch_i} : Document weight of ch_i.
- $\mathrm{W}_{ch_i}^D$: denotes the chunk-Score of ch_i in document D.

5.1.2 Design of `FbHash-B`

Our scheme adopts the TF-IDF weighing method [61] to find similar documents. The working of our scheme `FbHash-B` is divided into the following three steps:

5.1.2.1 Chunk Frequency calculation:

In this step, we first divide our document into certain blocks of bytes. We term each block as a *chunk*. The aim is to then calculate the number of times each chunk appears in the given document, i.e., calculate *chunk frequency*

1. Let D $=\mathrm{B}_0^D, \mathrm{B}_1^D, \mathrm{B}_2^D, \ldots, \mathrm{B}_{l-1}^D$ be a l byte long document, where B_i^D indicates the i^{th} byte of the document D. A chunk is a sequence of k consecutive bytes of D, where

$$\begin{gathered}
\mathrm{ch}_0^D=\mathrm{B}_0^D, \mathrm{B}_1^D, \mathrm{B}_2^D, \ldots\ldots, \mathrm{B}_{k-2}^D, \mathrm{B}_{k-1}^D \\
\mathrm{ch}_1^D=\mathrm{B}_1^D, \mathrm{B}_2^D, \mathrm{B}_3^D, \ldots\ldots, \mathrm{B}_{k-1}^D, \mathrm{B}_{k}^D \\
\mathrm{ch}_2^D=\mathrm{B}_2^D, \mathrm{B}_3^D, \mathrm{B}_4^D, \ldots\ldots, \mathrm{B}_{k}^D, \mathrm{B}_{k+1}^D \\
. \\
. \\
\mathrm{ch}_i^D=\mathrm{B}_i^D, \mathrm{B}_{i+1}^D, \mathrm{B}_{i+2}^D, \ldots\ldots, \mathrm{B}_{i+k-2}^D, \mathrm{B}_{i+k-1}^D \\
. \\
. \\
\mathrm{ch}_{l-k}^D=\mathrm{B}_{l-k}^D, \mathrm{B}_{l-k+1}^D, \mathrm{B}_{l-k+2}^D, \ldots\ldots, \mathrm{B}_{l-2}^D, \mathrm{B}_{l-1}^D
\end{gathered}$$

2. To compute the frequency of each of the identified chunks in the document, rolling hash technique is used. A rolling hash is a non-cryptographic hash function which allows the rapid computation of hash of each of the consecutive chunks. The fast computation of the rolling hash is due to the fact that the hash computation of a chunk utilizes the hash of the previous chunk, with which the current chunk shares most of the data bytes.

   In our construction, we use the Rabin Karp rolling hash function [62], which calculates the hash value with a very simple function using multiplications and additions as shown below:

RollingHash(ch$_i$) = $\mathrm{B}_i^D a^{k-1}$ + $\mathrm{B}_{i+1}^D a^{k-2}$+$\mathrm{B}_{i+2}^D a^{k-3}$ +. . . + $\mathrm{B}_{i+k-1}^D a^0$ modulus n

RollingHash(ch$_{i+1}$) = a*RollingHash(ch$_i$) - $\mathrm{B}_i^D \mathrm{a}^k$ + B_{i+k}^D modulus n

where a is a constant, k is the chunk size, and n is a large prime number.

In our implementation, the value of RollingHash(ch$_i$) is an unsigned 64-bit number, i.e., the rolling hash value lies between 0 to $(2^{64} - 1)$. the byte value B$_i$ and the constant a range between 0 to 255. This in turn puts a limitation on k as the value of k must satisfy the following relation:

$$B_i^D * a^{k-1} \leq 2^{64} - 1$$

As the maximum values of B$_i$ and a can be 255, thus,

$$255 * 255^{k-1} \leq 2^{64} - 1.$$

The maximum value of k which satisfies the above equation is 7 as shown below

$$2^{64} - 1 > 255 * (255)^6 \approx 2^{56}.$$

Hence, we choose k=7.

3. Once the rolling hash value of a chunk is calculated, the frequency of each chunk will be computed by the number

of times a rolling hash value appears. We make this observation by storing the rolling hash values in a hash table as follows :

- Index of the hash table is the rolling hash value of a chunk
- Value of the hash table is the number of times that rolling hash value (i.e., the chunk) appears in a document.

4. To guarantee that each unique chunk gets a unique rolling hash value, i.e., no collision happens, the value of n is taken as a prime number greater than 2^{56} (since, $256 * 256^{k-1} = 2^{56}$)
5. Based on chunk frequency, a chunk weight will be assigned to each chunk, using the following formula:

$$\text{ch-wght}_{ch} = 1 + \log_{10} (\text{chf}^{d}_{ch})$$

Thus, the higher chunk frequency, the higher weight and vice-versa. [1]

5.1.2.2 Document frequency calculation:

Document frequency of a chunk is the number of documents containing that chunk. The aim of this step is to identify the important chunks of the given document that can help identify it. Usually, the chunks that occur too frequently in a document have little relevance with respect to identifying the document. On the other hand, the less frequent chunks of a document are

[1]The chunk frequencies are normalized.

more important and relevant. Thus, there is a need to weigh up the effects of less frequently occurring chunks.

1. In order to calculate *Document Frequency*, a dataset of N document files has been taken (in our implementation N=1000). The document Frequency of a chunk ch is referred as df_{ch}. Document frequency is being calculated as follows:

   - Identify chunks of each document in the dataset.
   - Calculate rolling hash of each chunk.
   - Create a hash-table where the index of the hash table indicates the rolling hash value of chunks and the value of the hash table indicates the document frequency of the corresponding chunk.
   - Every unique chunk of each document will increase the value of the hash-table indexed by it′s rolling hash by 1.

2. Based on the document frequency, a *document weight* will be assigned to each chunk indicating the informativeness or uniqueness of the chunk. We denote it as doc-wght_{ch} and doc-wght_{ch} is calculated as follows.

   if $\text{df}_{ch} > 0$: $\text{doc-wght}_{ch} = \log_{10} (1000/\text{df}_{ch})$. [2]
   otherwise : $\text{doc-wght}_{ch} = 1$

5.1.2.3 Digest Generation

1. Once we have the chunk-weight and document-weight of each chunk in document D, a Chunk-Score (denoted is as

[2]1000 denotes the total documents considered in the dataset

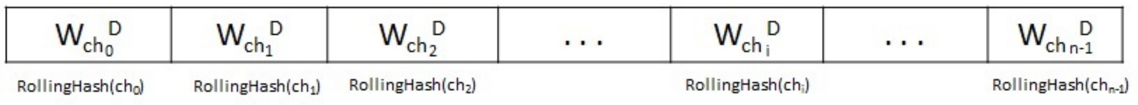


Figure 5.1: FbHash Digest

$W_{ch_i}^{D}$) is then calculated as follows

$$W_{ch_i}^{D} = \text{ch-wght}_{ch_i}^{D} * \text{doc-wght}_{ch_i}$$

This chunk score will be utilized to calculate similarity between two documents as shown later.

2. Now, the final `FbHash` digest of document D can be represented as a n element long vector where the index of vector represents the RollingHash(ch_i) and the value of the corresponding element is $W_{ch_i}^{D}$ as shown in fig 5.1. The index of the vector is represented by RollingHash(ch_i) because this value uniquely identifies ch_i. This is because, since we form 7-byte chunks, the total number of unique chunks can not be more than n.

   For ease of explanation in this paper, we represent the `FbHash` digest as shown below.

$$\text{digest}(D)=W_{ch_0}^{D},\ W_{ch_1}^{D},\ W_{ch_2}^{D},\ .\ .\ .\ ,\ W_{ch_{n-1}}^{D}$$

   The time complexity of `FbHash-B` digest generation is calculated as follows: The total chunks in a given document is $l - k$ where, l is the length of the document in bytes as mentioned at the start. Thus, computations of chunk frequencies in Section 5.1.2.1 will be done in $l-k$ steps. Steps involving document frequency calculation and assigning of document weights in Section 5.1.2.2 will be done offline and incurs no complexity in the online stage. Again, in Section 5.1.2.3, calculating the chunk score will be done in $l - k$ steps. Since the complexities of all the steps will be added, the overall complexity of `FbHash-B` digest generation will be $O(l)$, where, l is the length of the document in bytes.

5.2 Digest Comparison and Similarity Score Calculation

This section explains digest comparison and similarity score calculation of two documents. Let D_1 and D_2 be two documents and `FbHash` vector digest of D_1 and D_2 is as follows.

$$\text{digest}(D_1)=\mathrm{W}_{ch_0}^{D_1},\ \mathrm{W}_{ch_1}^{D_1},\ \mathrm{W}_{ch_2}^{D_1},\ .\ .\ .\ ,\ \mathrm{W}_{ch_{n-1}}^{D_1}$$
$$\text{digest}(D_2)=\mathrm{W}_{ch_0}^{D_2},\ \mathrm{W}_{ch_1}^{D_2},\ \mathrm{W}_{ch_2}^{D_2},\ .\ .\ .\ ,\ \mathrm{W}_{ch_{n-1}}^{D_2}$$

The similarity score between D_1 and D_2 is calculated using cosine similarity [63] as follows:

$$Similarity(D_1, D_2) = \frac{\sum_{i=0}^{n-1} W_{ch_i}^{D_1} * W_{ch_i}^{D_2}}{\sqrt{\sum_{i=0}^{n-1} {W_{ch_i}^{D_1}}^2} * \sqrt{\sum_{i=0}^{n-1} {W_{ch_i}^{D_2}}^2}} * 100$$

Final similarity score ranges between 0 to 100. where, 100 indicates the files are exactly the same whereas score of 0 indicates no similarity.

5.3 Design of `FbHash-S`

The purpose of `Fbhash-S` is to find similarity in compressed file format documents, e.g., docx, pptx, pdf etc. During our experiments, we observed that similarity detection at the byte-level does not work for compressed documents. For example, if there are two docx files that have 90 % similar content after compression, at the byte level there won't be any similarity with high probabilty. Thus, applying `FbHash-B` does not deliver good results.

The idea of `FbHash-S` is to use internal structure information of a document and perform syntactic matching to find similarity. Using the internal structure information of the document, `FbHash-S` first extracts the uncompressed content of the document. For example, in case of docx files, it will extract the text content (available in xml files stored in word folder) and images (stored in media folder under word folder). In our implementation we have used Apache POI package to extract text and images from the docx files. Then all the four steps, i.e., 1) *Chunk frequency calculation* 2) *Document Frequency Calculation* 3) *Digest Generation* 4) *Digest Comparison and Similarity Score Calculation* are peformed similarly as `FbHash-B` but individually on the text content and the images. Then the final score is the average of the score generated by text content and images. The run-time complexity of `FbHash-S` is higher than `FbHash-B` due to the additional step of content extraction.

5.4 Security Comparison of `FbHash` with Other Schemes Against Active Adversary Attacks

`ssdeep`, `sdhash`, `mrsh`, `mvHash` are some of the most popular and prominent approximate matching schemes. Several papers have shown that these algorithms are not secure against active adversary attacks. In the subsequent part, we discuss attacks on each of the above mentioned schemes and explain why `FbHash` is not prone to these attacks:

ssdeep[1]: The paper 'Security Aspects of Piecewise Hashing in Computer Forensics [20]' by Baier and Breitinger shows an anti-blacklisting attack by performing intentional modification. `ssdeep` divides the input document into variable sized non overlapping blocks and then computes a cryptographic hash (e.g., md5) of each block, which then contribute to the final `ssdeep` hash digest. The blocks are generated based on some trigger points. The way `ssdeep` works, irrespective of the file size, the file will always be split into 64 blocks of variable size. Thus the final hash signature will also consist of 64 bytes only. Also, for `ssdeep` to detect a similar file to a known blacklisted file, the two hash signatures should have at least a common 7-byte substring in both.

- To evade detection, an attacker thus makes sure that such a common 7-byte substring is never found by making minor modifications in the malcious file's content. For example, in one of the attack scenarios, the attacker changes one byte in only the 7^{th} block, 14^{th} block, 21^{st} block (multiples of 7 blocks) and so on while preserving the trigger point locations to change the hash signature. In the other attack scenario, the attacker finds few global trigger sequences that will always create a trigger irrespective of the file size. Insertion of such global trigger sequences will lead to different blocks creation, which will change the hash signature completely and thus will help evading detection. The advantage of such attacks is that by making very small changes in the content, the hash signature can be changed significantly.

However, in our case such attacks won't work. This is so because making small changes in the content will lead to creation of only few new chunks, having very low chunk frequency and thus low chunk score, preserving most of the high scoring chunks. In our similarity calculation, the low scoring chunks (i.e., the less relevant chunks) do not contribute much in the actual similarity comparison and the file will still be detected as similar to a known file with very high probability. In order to change the hash signature, the attacker will have to modify the majority of the high scoring chunks. In `FbHash`, as each chunk differs from its neighboring chunk by only one byte (the rest of the bytes are overlapping), in order to highly influence the final score, each chunk needs to be modified. Since the chunk size is 7-bytes only, in order to impact similarity score every 7th byte has to be modified. This will alter the content of the original document data significantly and the attacker's aim to make feasible changes will be defeated and thus of no use.

sdhash: Breitinger et al. in their work titled - "Security and Implementation Analysis of the Similarity Digest sdhash" [34] state that given a file, it is easily possible to tamper a given file to bring down the similarity score to approximately 28. Another paper titled - "A collision attack on `sdhash` similarity hashing" [16] by Chang et al. shows an anti-forensics mechanism that allows someone to generate multiple dissimilar files corresponding to a particular file with 100% `sdhash` similarity, which can confuse the filtering process. Both of the attacks are possible because the entire content of a file doesn't contribute to the final hash generation. Only some of the selected chunks participates in the final hash generation.

In our scheme, each and every byte of the document contributes to the final score (by formation of a new chunk) and their influence on the final score depends on their importance to the document. Hence, any modification will impact the final score. Further, to bring the similarity score really low or close to zero, almost every chunk has to be modified, which as discussed earlier will alter the content of the document significantly and make it altogether a different file.

mvHash-B The paper titled - “Security Analysis of MVhash-B Similarity Hashing” [17] shows that it is possible for an attacker to fool the algorithm by causing the similarity score to be close to zero even when the objects are very similar. The proposed attack is possible because `mvHash` compresses the input document using Run-length encoding (RLE). This gives the attacker freedom to bring the similarity score down with very few modifications.

No such compression is performed in `FbHash`. Every byte contributes to the final score calculation and hence our scheme is resistant to the attack.

5.5 Comparative Evaluation of `FbHash`

In this section, we present a comparative analysis of `Fbhash` with the two most prominent approximate matching algorithms, (i.e., `ssdeep` and `sdhash`) on two test cases: **Fragment Detection** and **Single-common-block correlation**. We chose these two algorithms for comparison as their reference standard implementation codes are available online and they are the most popular algorithms used by the forensics community. Section 5.5.1 describes the results of the Fragment Detection test, and the results of the Single-common-block correlation test are shown in Section 5.5.2.

5.5.1 Fragment Detection

This test aims to identify the tool's ability to correlate a fragment (small part of a file) to its source file. We present a comparison between `ssdeep`, `sdhash` and `FbHash` performance. Fragments are generated in two ways - Sequential Fragments and Random Fragments, in a similar way as shown in [11].

- **Sequential Fragments**: Create the fragment from the beginning of the file. For example, for a 1000 byte long file, a 1 % fragment of a file is the first 10 bytes of the source file.
- **Random Fragments:** Generate the fragment from a randomly chosen position in the file . For example, for a 1000 byte long file, if the randomly chosen position is 'r', then the 1 % long fragment is the next 10 bytes from r.

We perform the test on 'Text dataset'and 'Docx dataset' described in sections 5.5.1.1 and 5.5.1.2 respectively.

5.5.1.1 Text Dataset Result

Experimental Setup: The test is performed on a dataset of 960 fragment files (480 sequential fragments and 480 randomly generated fragments), generated from 20 variable size text files (5 KB to 1 MB taken from T5 corpus[64]). Each text file generates 24 sequential fragment files and 24 random fragment files of the following sizes: 95 %, 90 %, 85 %, 80 %, 75 %, 70 %, 65 %, 60 %, 55 %, 50 %, 45 %, 40 %, 35 %, 30 %, 25 %, 20 %, 15 %, 10 %, 5 %, 4 %, 3 %, 2 %, 1 %, <1 %. The total number of comparisons performed by each scheme for text files is thus 19 200.

Results:

- The graph in Fig. 6.1 represents the results of `ssdeep`, `sdhash` and `FbHash` on Text datasets.
    1. *X-axis* represents the different fragment sizes.
    2. The *first Y-axis(left)* represents the *Match Percentage*. *Match percentage* or correlation detection rate is defined as the percentage of those test samples where, the tools are able to detect similarity by giving a valid match score (in other words, the number of times on a scale of 100, the tool is able to correlate fragments to their original source files for a given particular fragment size). This is illustrated in the form of lines in the graph. For example, in Fig. 6.1, for the fragment size 50 %, `ssdeep` (represented by blue line) detects similarity between a fragment and its source file for 90 % of the total samples tested but fails for the remaining 10 %. On the other hand, for the fragment size 50 %, `sdhash` (red line) and `FbHash-B` (green line) are able to correlate the fragments to their original source files for all the samples tested, i.e., 100 % correlation detection rate (due to overlap between the red and green line, only the red line is visible). Since the test is performed

for the fragments as small as 1 % of the file, hence any similarity score greater than 1 is being considered as a valid match in these experiments.

3. The *second (right) Y-axis* represents the average similarity score calculated by the `ssdeep`, `sdhash` and `FbHash-B` between a fragment considered as one document and the original source file as the other document for a given fragment size. Bars in the graph illustrate the average score. For example, in Fig. 6.1 for 95 % long fragments, we calculated the similarity score between each file and its 95 % long fragment. The blue bar represents `ssdeep`, the red bar represents `sdhash` and `FbHash` is represented by the green bar. The averge similarity scores generated by `ssdeep`, `sdhash` and `Fbhash` for 95 % fragments are 95, 89 and 97 (out of total of 100) respectively.

From Fig. 6.1, it can be seen that all the three tools show a 100 % correlation detection rate for fragment sizes ≥ 55 % (due to overlap only the horizontal blue line is visible). `ssdeep` can correlate a fragment to its source file if it is 50 % or more of the source file with high correlation detection percentage, i.e., ≥ 90 % of the times of the total samples tested. However, it cannot identify similarity for 20 % or smaller fragment size. `sdhash` can detect similarity for fragment size of 15 % or more with high percentage, i.e., ≥ 85 % of the times for the total samples tested. However, its correlation detection rate drops to 60 % or less as the fragment size decreases beyond 10 % or less. On the other hand, the correlation detection rate for `FbHash` is 100 % for all the fragment sizes, i.e., all the fragments that were

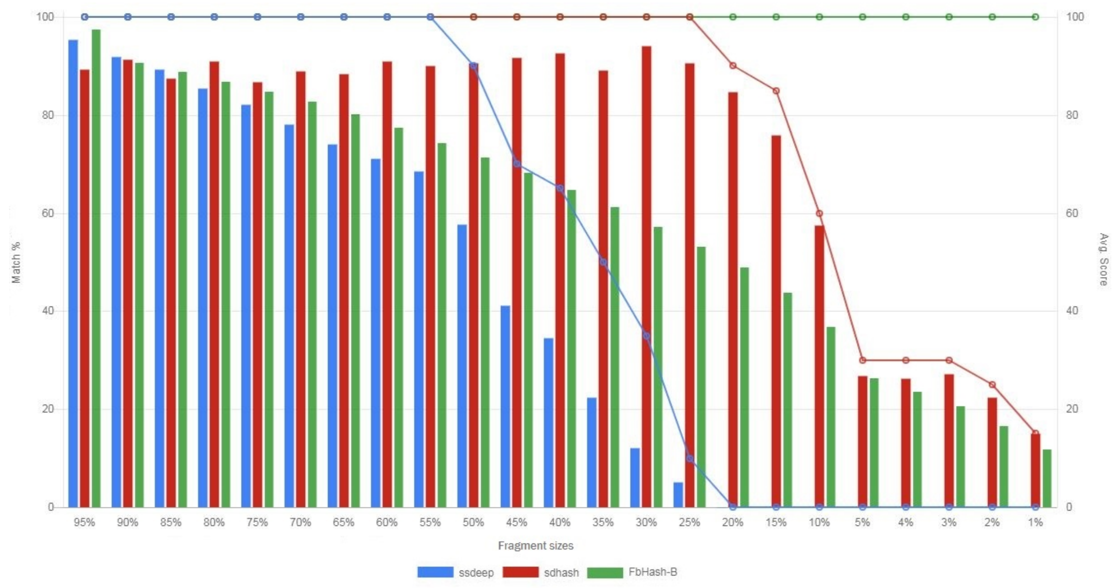


Figure 5.2: Fragment Detection Test Results on Text dataset

tested were successfully correlated to their original source files even when the fragment size was as low as 1 % as represented by the horizontal green line.

If we look at the right y-axis of Fig. 6.1, it can be seen that in case of `sdhash`, the relationship between the similarity scores predicted by the tool and actual similarity of the fragment to its source file is not consistent. For example, for fragment size 30 %, the similarity score given by `sdhash` is comparatively higher than that given for fragment size 95 % whereas it should be the reverse. This shows that `sdhash` similaritiy scores do not reflect the actual similarity. On the other hand, this relationship is correctly reflected by `FbHash`. It can be seen that as the fragment sizes decrease from 95 % to 1 %, the average score given by `FbHash` also decreases. This holds true for `ssdeep` as well up to fragment size ≥ 25 %. However, beyond that `ssdeep`, cannot identify the similarity which is not the case for `FbHash`. `FbHash` shows the correct relationship even for fragments as small as 1 % to 5 % of the file.

- **F-score:** We calculate the F-score in order to calculate the accuracy of the `ssdeep`, `sdhash` and `FbHash-B`. The F-score is a generic measure to test the accuracy of a tool that considers both the precision and recall values of the tool while computing the final score. The precision parameter signifies how many similar files were predicted similar by the tool and recall indicates how many similar files predicted by the tool were actually similar. Precision, Recall and F-score are calculated as follows:

$$F\text{-}score = 2 * \frac{precision*recall}{precision+recall}$$

$$precision = \frac{TP}{TP+FP}$$
$$recall = \frac{TP}{TP+FN}$$

where, TP refers to true positive, TN refers to true negative, FP refers to false positive and FN refers to False negative results generated by the tool.
Let f1 and f2 be two given files and the similarity score generated by an approximate matching tool be represented as AM(f1,f2) which ranges between 0 to 100. Let t be a threshold value, defined later in this section. Since we have generated the dataset with known similarity, thus, we know the actual similarity in the files which we call as ground similarity represented by GS(f1,f2) (ranges between 0 to

100 where 0 indicates no similarity and 100 indicated f1 and f2 are identical). Any value of GS(f1,f2) $>$ 0 indicates that f1 and f2 shares some similarity. The result of a tool is considered TP, TN, FP and FN according to the following conditions:

TP: if GS$\geq$1 and AM(f1,f2)$\geq$ t
TN: if GS$<$1 and AM(f1,f2)$<$ t
FP: if GS$<$1 and AM(f1,f2)$\geq$ t
FN: if GS$\geq$1 and AM(f1,f2)$<$ t

`t` represents the threshold value of the similarity score generated by a tool. It is considered that a tool has found a match if the similarity score generated by the tool is greater than or equal to t (i.e. AM(f1,f2)$\geq$ t is a match). The paper [13] claims that the threshold score of up to 22 yields near-perfect detection for sdhash. Since no such value is suggested for ssdeep, the value of t is taken to be 22 for all three schemes in order to compare the results. We observed that for t=16 we get the best detection rate for `FbHash`. Thus we have shown F-score results of `FbHash` for both t=22 and t=16 shown in Fig. 6.2 and 5.4 respectively. Table 5.5.1.1 shows the TP, TN, FP, FN, precision, recall and F-score value generated by the experiment. A total of 9200 comparisons are performed for each sequential fragment and random fragment test case.

As the results show, all the three schemes have 0 False Positive Rate (FPR). However, `ssdeep` has the highest false Negative Rate (FNR) and `FbHash` has the minimum FNR. Figs. 6.2 and 5.4 show that `FbHash-B` detects similarity with the highest accuracy of 98 % with suggested threshold

	ssdeep (t=22)		**sdhash** (t=22)		**FbHash-B** (t=22)		**FbHash-B** (t=16)	
	Sequential	Random	Sequential	Random	Sequential	Random	Sequential	Random
True Positive (TP)	244	246	373	373	408	419	438	442
True Negative (TN)	8740	8740	8740	8740	8740	8740	8740	8740
False Positive (FP)	0	0	0	0	0	0	0	0
False Negative (FN)	216	214	87	87	52	41	22	18
False positive rate (FPR)	0	0	0	0	0	0	0	0
False negative rate (FNR)	0.0234	0.023 26	0.009 404	0.0094	0.0056	0.0044	0.0023	0.0019
Precision	1	1	1	1	1	1	1	1
Recall	0.5304	0.5347	0.8108	0.8108	0.8869	0.9108	0.9521	0.9608
F-score	0.6931	0.6968	0.8955	0.8955	0.9400	0.9533	0.9755	0.9789

Table 5.1: Fragment Identification test case F-Score calculation for Text-Data set. Total number of comparisons performed for each sequential and Random fragments is 9200.

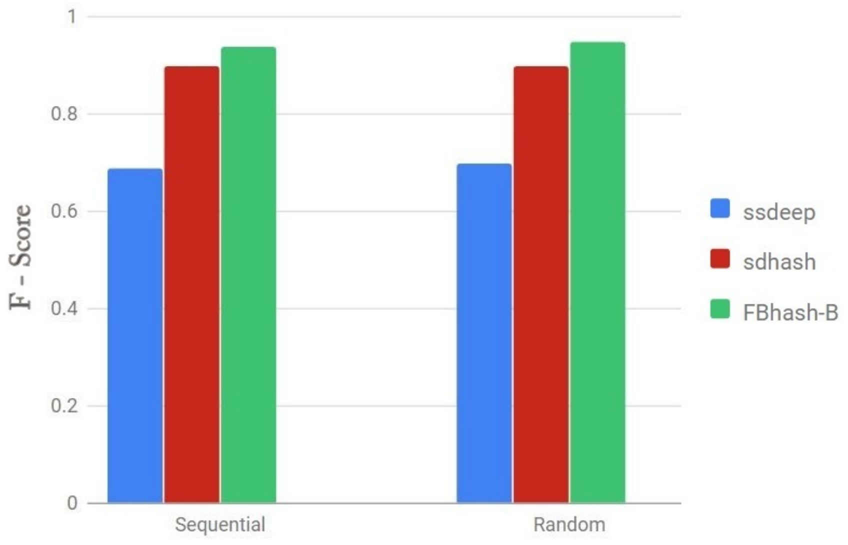


Figure 5.3: Figure shows the F-score comparison for Fragment Identification test on Text dataset. The value of t is taken to be 22 for all three schemes.

(16) and 95 % with threshold 22, whereas the accuracy of `ssdeep` is 69 % and `sdhash` is 89 %.

5.5.1.2 Docx Dataset Results

We also test `ssdeep`, `sdhash` and `FbHash` for docx dataset. Following are the details of the experiment.

Experimental Setup: The test is performed on the dataset of 960 fragment files (480 sequential fragments and 480 randomly generated fragments), generated by 20 variable size docx files. Each docx file generates 24 sequential fragment file and 24 random fragment files of the following sizes: 95 %, 90 %, 85 %, 80 %, 75 %, 70 %, 65 %, 60 %, 55 %, 50 %, 45 %, 40 %, 35 %, 30 %, 25 %, 20 %, 15 %, 10 %, 5 %, 4 %, 3 %, 2 %, 1 %, <1 %. The fragments are generated only by segmenting (cutting) *content* of docx files into pieces. Total number of comparisons

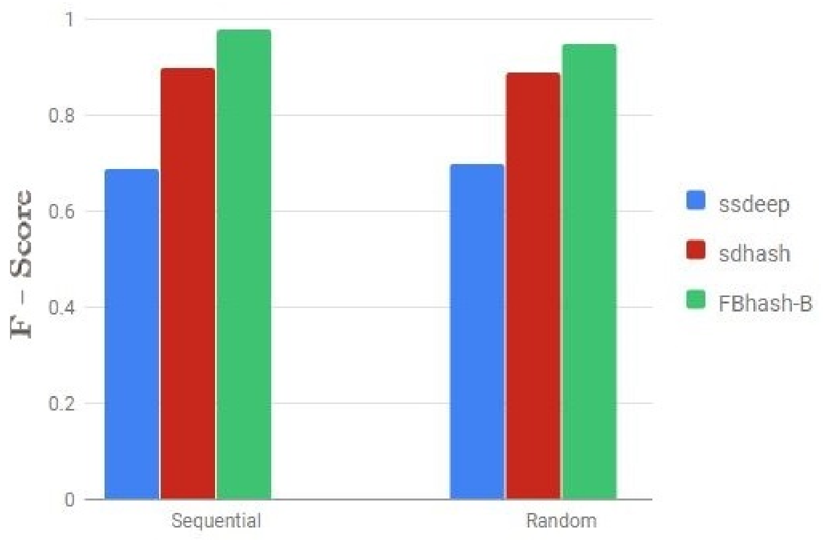


Figure 5.4: Figure shows the F-score comparison for Fragment Identification test on Text dataset. The value of t is taken to be 22 for ssdeep and sdhash and 16 for `FbHash`.

performed by each scheme for docx files is 19 200.

Results:

- Fig. 5.5 shows the average similarity score and match percentage of `ssdeep`, `sdhash` and `FbHash-B` on docx dataset. The results obtained by all three algorithms are imprecise (inaccurate). As shown in Fig. 5.5, `sdhash` can detect similarity for all fragment sizes with higher match percentage compared to `ssdeep`. `FbHash-B` on the other hand is able to correlate even the smallest fragment with 100 % detection rate (green horizontal line). However, the average matching scores of all the three algorithms do not reflect the actual similarity as fragment sizes decrease from 95 % to 1 %. Thus, none of these algorithms is useful.
  The reason behind this is that docx is a compressed file structure due to which any modification in the content of a file changes the final compressed file completely with high percentage. Hence, at the byte level, the two different fragments or versions of a docx file are completely different. Since both `ssdeep` and `sdhash` work at byte level, the resultant similarity score is completely inaccurate. Hence, we state that byte-level matching is not sufficient to find similarity of compressed file structures.

  Fig. 5.6 presents a comparison between the results obtained by `ssdeep`, `sdhash`, `FbHash-B` and `FbHash-S`. As Fig. 5.6 shows, the results obtained by `FbHash-S` are accurate in terms of similarity score and its relationship to the actual similarity of a fragment to its source file, i.e., as the fragment sizes decrease, the scores also decrease.

- **F-score:** F-score is calculated similarly as explained in Sec-

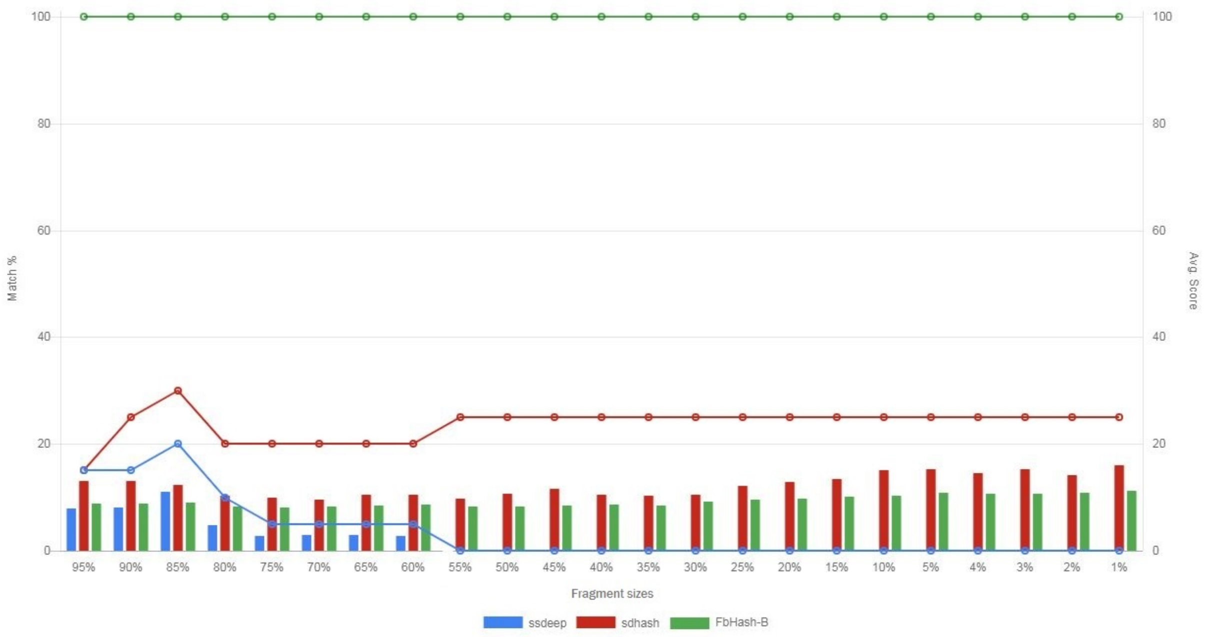


Figure 5.5: Fragment Detection Test Results on Docx dataset

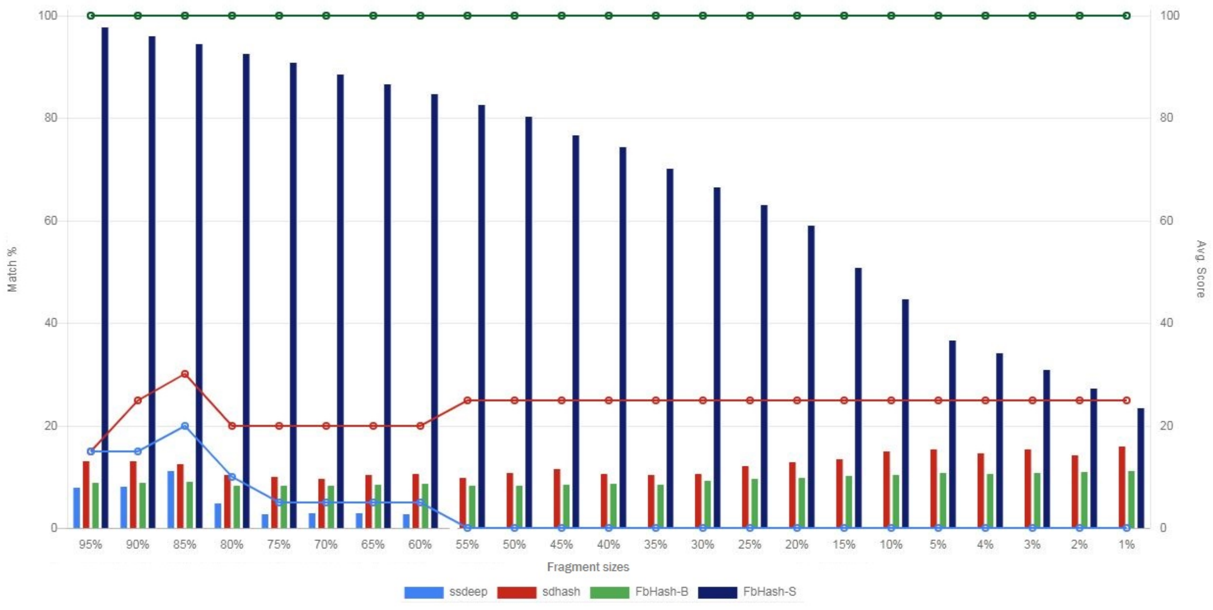


Figure 5.6: FbHash-S Fragment Detection Test Results on Docx dataset

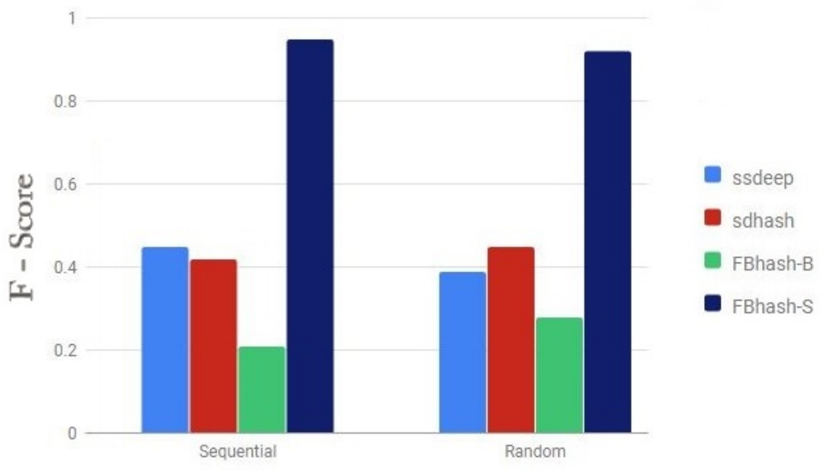


Figure 5.7: Fragment Detection Test F-score comparison on Docx dataset

tion 5.5.1.1. Fig. 5.7 shows the comparison between the F-score of `ssdeep`, `sdhash`, `FbHash-B` and `FbHash-S`. It shows that `FbHash-S` outperforms `ssdeep`, `sdhash` and `FbHash-B` by 50 %, 53 % and 73 % respectively higher accuracy for sequential fragments and by 53 %, 47 % and 64 % respectively higher accuracy for random fragments. `FbHash-S` achieves accuracy of 95 % and 92 % for sequential and random fragments respectively.

5.5.2 Single-common-block file correlation

This test was first proposed in paper [3] by Vassil Roussev. It aims to identify the ability of a tool to correlate the related documents, i.e., those which share a common single block of data. For this test case as well we generated the ground truth dataset with known similarity. To generate the dataset, T5 corpus [64] is being used. the dataset is generated following the steps given below.

- 3 files of the same size are taken from the T5 corpus.
- The following 10 different sized fragments of the first file is created: 100 %, 66.66 %, 42.86 %, 25 %, 11.11 %, 5.2 %, 4.1 %, 3.09 %, 2.04 %, 1.01 %.
- Each fragment will be inserted in randomly chosen positions in the second and third file one by one. This will result in the creation of 10 pairs of the second and third file with shared common block of 50 %, 40 %, 30 %, 20 %, 10 %, 5 %, 4 %, 3 %, 2 %, 1 % respectively.
- Take another triplet of files and repeat from step 1 to step 3 for various file sizes.

We perform the test on 'Text dataset' and 'Docx dataset', and the results of the tests are shown in Section 5.5.2.1 and Section 5.5.2.2 respectively.

5.5.2.1 Text Dataset Result

Experimental Setup: The test is performed on a dataset of 280 document pairs with a shared single common block, generated from 60 variable sized text files (5 KB to 10 MB).

Results:
The graph in Fig. 5.8 represents the results of `ssdeep`, `sdhash` and `FbHash-B` on Text datasets. The results show that `ssdeep` (blue line) can detect the similarity for ≥ 30 % single-common-block similarity (commonality) with high percentage (≥ 75 %). `sdhash` (red line) can detect similarity up to 3 % single-common-block size with high percentage (≥ 93 %) whereas `FbHash-B`

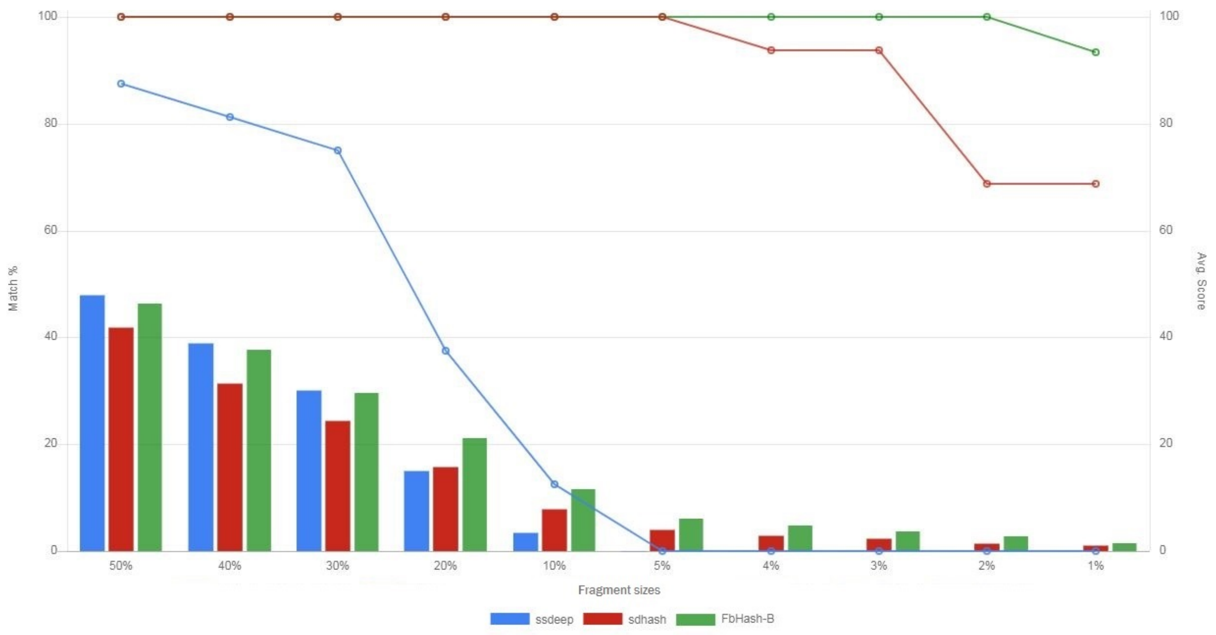


Figure 5.8: Single-common-block file correlation Results for Text-Data Set

(green line) can detect similarity up to 1 % single-common-block size with high percentage (≥ 93 %). `sdhash` and `FbHash-B` both perform well in this case and the similarity score generated by both the tools are very close to the actual similarity.

5.5.2.2 Docx Dataset Results

This subsection presents the results of the 'Single-Common Block Detection' test for docx dataset. Following are the details of the experiment.

Experimental Setup: The test is performed on a dataset of 280 document pairs with a shared single common block, generated from 60 variable size docx files (5 KB to 1 MB).

Results:
Fig. 5.9 shows the average similarity score and match per-

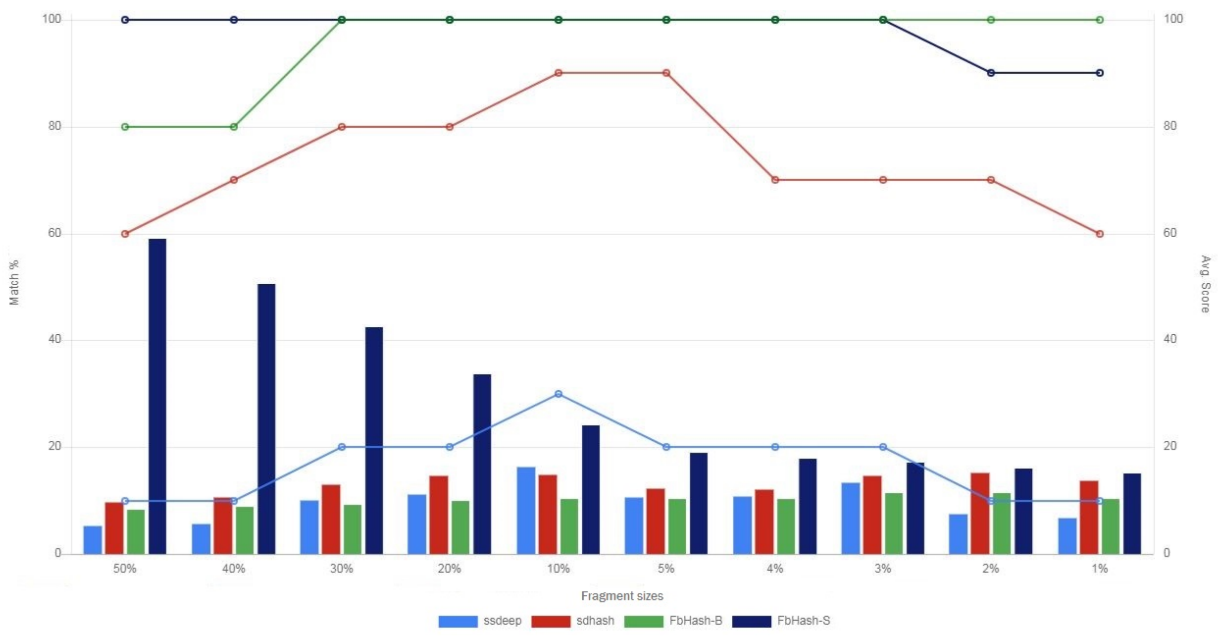


Figure 5.9: Single-common-block file correlation Results for Docx dataset

centage of `ssdeep`, `sdhash`, `FbHash-B` and `FbHash-S` on Docx dataset. The correlation detection rate or the match percentage of `ssdeep`, `sdhash` and `FBHash-B` fluctuates with common block size and is not consistent. On the other hand, the correlation detection rate of `FBHash-S` is consistent and very high (≥ 90 %) for all block sizes from 50 % to 1 %.

In terms of average matching score, it can be seen that results obtained by `ssdeep`, `sdhash` and `FBHash-B` are imprecise and do not reflect the actual similarity, since docx is a compressed file structure. On the other hand, results obtained by `FbHash-S` are more accurate and consistent.

5.6 Summary

In this chapter, we presented the first approximate matching scheme which is secure against active manipulations. The proposed scheme is able to correlate a fragment as small as 1 % to the source file and able to detect commonlaity as small as 1 % between two documents with correct/appropriate (low) similarity score. We experimentally demonstrated that the proposed approach provides 98 % accuracy in some test cases.

Chapter 6

FbHash-E: A Time and Memory efficient Version of FbHash Similarity Hashing Algorithm

`FbHash` [18], described in the previous chapter of this thesis, is the only recent algorithm which has been shown to be secure against active attacks and provides the highest (98%) accuracy. However, the incorporation of security features impacts other factors such as throughput, scalability, usability, etc., of a system which can incur additional costs. Due to this, `FbHash` is slower and memory intensive compared to other popular existing algorithms. Often practical implementations of such algorithms require a trade-off between security and performance. Security add-ons impact other factors such as throughput, scalability, usability, etc., of a system which can incur additional costs. The need of the hour is a construction that provides a balance between security and efficiency while maintaining the accuracy and reliability of the results.

In this chapter, we propose a new version of `FbHash` termed as

FbHash-E. We propose some new implementation changes and show that FbHash-E is faster and requires lower memory usage compared to its predecessor. We provide a detailed performance analysis of the same and compare its performance with respect to FbHash and other popular algorithms available in the literature. We also perform a thorough security analysis of FbHash-E which was missing for FbHash in [18] by performing a variety of state-of-the-art containment and resemblance tests. We provide a comprehensive comparative analysis with other existing popular algorithms and show that our tool outperforms all other tools in all of the tests performed. This shows that though our tool is a little less accurate than FbHash, our tool still is the best compared to other existing tools in terms of security and robustness.

The contributions of this chapter are as follows:

1. We show that the current best approximate matching tool (in terms of security),i.e., FbHash is not very efficient in design in terms of memory consumption and run time performance and list out possible reasons behind them.

2. We then propose a more time and memory-efficient version of FbHash and term it as - FbHash-E.

3. We present a novel bloom filter based document frequency design implementation in FbHash-E that reduces the memory requirements to a few MBs compared to its predecessor FbHash.

4. We show that design changes done to improve the performance of FbHash does not impact its ability to detect similar files by much, and there is only an average difference of 7.5 in the scores generated by FbHash and FbHash-E.

5. We show a detailed security analysis of `FbHash-E`, perform more tests to evaluate its robustness that was not done earlier for `FbHash` and compare its results with the other state-of-the-art forensic tools. We show that `FbHash-E` outperforms all the other tools in all the security tests conducted.

6. We also demonstrate the result of *Consistency Test* and *Code Version Identification Test*, discuss their significance in relation to the evaluation of a forensic tool and present their results. We further show that `FbHash-E` delivers the best results under these two sets of tests as well.

The rest of the chapter is organized as follows: Section 6.1 provides a brief overview of FbHash similarity hashing scheme. Section 6.2, discusses the design limitation of FbHash. This is followed by Section 6.3, which presents improvements to FbHash scheme and propose a new scheme FbHash-E. Section 6.4 presents the performance statistics, followed by Section 6.5 that provides a thorough comparative evaluation of FbHash-E with other existing schemes on benchmark test cases. Finally, we conclude our work in Section 6.6 and discuss future prospects.

6.1 FbHash

In this section, we briefly discuss the working of FbHash algorithm. For further details, one can refer [18].

Notations: We first present some important notations that have been adopted by `FbHash` and also in `FbHash-E`.

- Chunk : Sequence of k consecutive bytes of a document.

- ch_i^D : represents the ith chunk of a document D.
- N : denotes the total number of documents in the document corpus.
- $\text{RH}(\text{ch}_i)$: Rolling hash value of ch_i
- ch-wght_{ch_i} : Chunk weight of ch_i
- doc-wght_{ch_i} : Document weight of ch_i.
- $\text{W}_{ch_i}^D$: denotes the chunk-Score of ch_i in document D.

FbHash (Frequency-based hashing) adapts the concept of TF-IDF (Term Frequency — Inverse Document Frequency) technique to quantify similarity between two documents. TF-IDF is a statistical measure used to estimate the importance of a word to a document in a collection or corpus. FbHash uses this concept to identify important fragments (features) of a document. Based on the importance or relevance of a fraction, its contribution to the final similarity score is measured. However, each and every part of the document contributes to the final score depending on its relevance. The `FbHash` algorithm comprises of the following two steps:

- **Digest Generation:** The digest of document is generated as follows:
  - The document is first divided into byte chunks. A chunk is a sequence of k consecutive bytes of the document, where each neighboring chunk has (k-1) bytes in common. In `FbHash`, the value of k is 7.
  - The frequency of each chunk in the document is next calculated. The chunk frequency is represented by its

corresponding rolling hash frequency in `FbHash`. It is computed by calculating rolling hash of each chunk and incrementing the corresponding frequency value in a hash table by 1 at the rolling hash index upon each occurrence of a rolling hash.

– Based on the chunk frequency, a chunk weight (ch-wght) of each unique chunk is computed using the following formula:

$$\text{ch-wght} = 1 + \log_{10}(\text{chunk-frequency})$$

– To calculate the digest of a document, the document frequency of each chunk is required. Document frequency of a chunk indicates the number of the document containing the chunk in a data-corpus. Document frequency is used to signify the importance of the chunk in a document. Usually, the chunks that occur in very many documents have little significance to identifying a document. On the contrary, the less frequent chunks are more critical and relevant to represent a document in a collection. This is a pre-computed value, which is computed using a dataset of N documents and calculating the number of documents containing that chunk.

– Using the calculated document frequency (denoted by *df*), a document-weight (doc-wght) is assigned to each chunk as follows:

$$\begin{array}{ll} \text{if df} > 0 : & \text{doc-wght} = \log_{10}\frac{N}{df} \\ \text{otherwise} : & \text{document weight} = 1 \end{array}$$

– Once we know the chunk-weight and document-weight of a chunk, a chunk score is assigned to each chunk in

the document as follow:
Chunk-score = Chunk-weight × document-weight

– The final digest is vector of <rolling-hash,chunk-score> tuple. The digest of a document D with n unique chunks is represented as follows:

$$\text{digest(D)} = (\text{RH}(\text{ch}_0),\text{W}^{D}_{ch_0}),\ (\text{RH}(\text{ch}_1),\text{W}^{D}_{ch_1}),\ .\ .\ .\ ,\ (\text{RH}(\text{ch}_{n-1}),\text{W}^{D}_{ch_n-1})$$

- **Digest comparison and similarity score calculation** Let D_1 and D_2 be two documents with $n1$ and $n2$ unique chunks respectively. The `FbHash` digests of D_1 and D_2 are respectively as follows:

$$\text{digest}(D_1)=(\text{RH}(\text{ch}_0),\text{W}^{D1}_{ch_0}),\ (\text{RH}(\text{ch}_1),\text{W}^{D1}_{ch_1}),\ .\ .\ .\ ,\ (\text{RH}(\text{ch}_{n-1}),\text{W}^{D1}_{ch_n 1-1})$$
$$\text{digest}(D_2)=(\text{RH}(\text{ch}_0),\text{W}^{D2}_{ch_0}),\ (\text{RH}(\text{ch}_1),\text{W}^{D2}_{ch_1}),\ .\ .\ .\ ,\ (\text{RH}(\text{ch}_{n-1}),\text{W}^{D2x}_{ch_n 2-1})$$

The similarity score between D_1 and D_2 is calculat ed using cosine similarity [63] as follows:

$$Similarity(D_1, D_2) = \frac{\sum_{i=0}^{n1\cap n2} W^{D_1}_{ch_i} * W^{D_2}_{ch_i}}{\sqrt{\sum_{i=0}^{n1-1} {W^{D_1}_{ch_i}}^2} * \sqrt{\sum_{i=0}^{n2-1} {W^{D_2}_{ch_i}}^2}} * 100$$

The final similarity score ranges between 0 to 100. where 100 indicates, the files are exactly the same, whereas a score of 0 indicates no similarity.

6.2 Design Limitations of `FbHash`

In this section, we discuss the design issues and limitations of `FbHash` based on following three dimensions - *Compactness*, *Ef-*

ficiency and *Correctness*.

- **Compactness** refers to the amount of memory (space) needed for one execution of the tool to produce expected results. Space or memory efficiency becomes crucial for the fast execution of the tool as the size of the input grows. The memory resource requirement increases with the size of the input data. Particularly in case of approximate matching algorithm, where the input size is uncertain, it becomes important to limit the memory consumption as much as possible to run the tool correctly and efficiently on devices with regular computing power. In the current version of `FbHash`, high amount of memory is required for the following steps:

  - *Document-weight dictionary*: While calculating the `FbHash` digest, each chunk is assigned a document weight calculated using inverse document frequency value. Since we need this value for each chunk of the document under investigation, we need quick access to the document-weight dictionary during runtime. This necessitates the storage of this dictionary into the RAM. However, theoretically, this dictionary may contain 2^{56} unique chunks, as the chunk size is 7 bytes and as `FbHash` uses hash table data structure to store it, this in turn leads to significant memory overhead as storing such a huge table requires a RAM memory space close to 2^{20}GB while requiring the RAM for performing other computations as well simultaneously which is humongously large.
  - *Digest size*: The chunk size of the previous version of

FbHash implementation is 7 bytes and rolling hash size is 55 bytes. Theoretically, in the worst case, digest calculation and storage of one file may have 2^{56} distinct chunk and needs huge RAM memory, which is practically infeasible to execute on regular devices. Even in other cases, it has been observed that the digest size is still quite big for large files.

In the new version FbHash-E, we have focused on these limitations and addressed each of them.

- **Efficiency** refers to the amount of computing time required to obtain the similarity score. The aim is that the tool can run as fast as possible while preserving the correctness and compactness of the algorithm. In case of runtime efficiency the previous version of FbHash had the following limitations:

  - For every execution, the tool needed to load (read and create) the document-weight-dictionary into the RAM, which added a constant execution time to each run of the tool.
  - Creating a big hash table to calculate the chunk frequency again leading to increased time execution.

  We have done some alterations in the new proposed version of FbHash-E tool to significantly reduce the run-time.

- **Correctness** refers to the accuracy of the results obtained by the algorithm. This dimension ensures that the similarity score produced by the tool reflects actual similarity or very close to the actual similarity of the data objects. Accuracy is one of the most crucial dimension since the other two dimension can be improved by providing a resourceful and higher computing environment. While in [18] it has been shown that `FbHash` gives most accurate similarity scores compared to other peer tools existing in the literature. However, the tests done are too limited and a more thorough investigation on its security and robustness is required.

The main design challenge is to carefully balance the requirement of these three dimensions such that `FbHash` is compact, efficient and accurate. Achieving the trade-off is difficult because of the contradicting requirements. While a short digest creation will significantly improve time and memory footprint, it will severely impact the accuracy of the similarity score so generated. In the next section, we propose a few important modifications to improve `FbHash` tool's efficiency and compactness while maintaining the correctness of the similarity score.

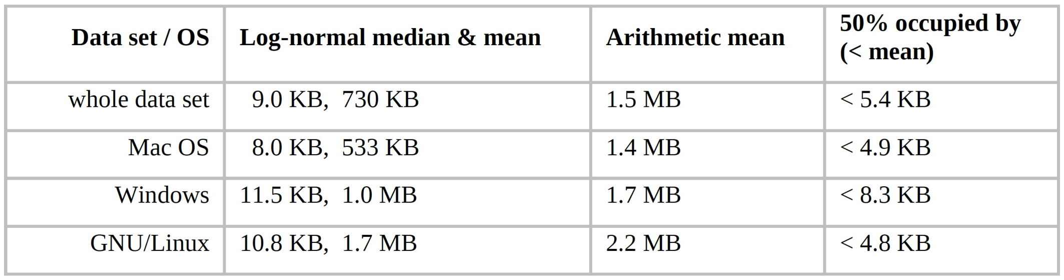
6.3 Proposed improvements to `FbHash` Similarity Hashing Scheme

In order to overcome the limitations of `FbHash`, we propose the following refinements to the tool and present `FbHash-E`, an improved version of `FbHash`.

6.3.1 Alteration in the chunk size

In the context of the previous implementation of `FbHash`, as discussed earlier, a chunk is the sequence of 7 consecutive bytes. To generate the digest of a file, we need to calculate the chunk frequency of each and every chunk of the file. Chunk frequency is computed by calculating the rolling hash of the chunks and placing their occurrence into a look-up table on the respective rolling hash index, and incrementing the count for every occurrence. In FbHash implementation, the rolling hash is 56 bit long based on the selected prime number to compute rolling hash. (For more details please refer [18] section 3.2.). Hence theoretically speaking, in the worst-case scenario, it is possible to have 2^{56} unique chunks leading to 2^{56} unique hash indices in the hash table, making hash table extremely huge and outside the RAM memory range. Also, experimentally, it was observed that a chunk size of 7-bytes was increasing the sample space for the possible rolling hash values and increasing the size of the chunk frequency table. This, in turn was causing an increase in the digest size so generated and subsequently the run time of the tool.

To overcome this issue, in `FbHash-E`, we have reduced the chunk size and rolling hash size to 5-bytes, respectively. This implementation change has brought in 2 performance implications - reduced memory requirements and reduced time taken to calculate the file digest. Due to rolling hash size getting reduced to 5 bytes ($= 2^{40}$ bits), theoretically, the total number of unique chunks that can now get generated is 2^{40}, which is still substantially huge. However, generation of such a huge number of unique chunks does not happen in a regular file system due to repetition, language structure, and file format. This fact is further strengthened by an experiment that we performed where we tested files of various sizes and calculated the number of unique chunks generated. The files in the experiment ranges from 2KB to 100 MB from T5 corpusa [1], digital corpora [2], Biostat Datasets [3] and kaggle [4]. We choose this range based on the study conducted by Dinneen et al. [6] and Douceur el al. [65] [66] file sizes are fairly consistent across the file system. figure 6.3.1 summarizes the file system distribution within different operataing system. Small files are very common in each OS files below 5 KB account for 50% of Mac and Linux collections, and files below 8.5 KB account for 50% of files in Windows. 95% of the files are under 8 MB and files larger than 32MB are rare (probability ≤ 0.001).



Data set / OS	Log-normal median & mean	Arithmetic mean	50% occupied by (< mean)
whole data set	9.0 KB, 730 KB	1.5 MB	< 5.4 KB
Mac OS	8.0 KB, 533 KB	1.4 MB	< 4.9 KB
Windows	11.5 KB, 1.0 MB	1.7 MB	< 8.3 KB
GNU/Linux	10.8 KB, 1.7 MB	2.2 MB	< 4.8 KB

Figure 6.1: file system distribution within different operating system from [6]

The result of the experiment is shown in table 6.1. The first column represents the different file size that has been considered. We have considered dataset of 20 file for each sizes. The second column represents the avg. number of unique chunks for each file size, and the third column denotes the average of the fraction of unique chunks to the total number of chunk. As seen from Table 6.1, the number of distinct chunks, that represent a particular file does not exponentially increase with the

[1]http://roussev.net/t5/t5.html
[2]https://digitalcorpora.org/
[3]http://courses.washington.edu/b517/Datasets/datasets.html
[4]https://www.kaggle.com

Table 6.1: Average number of unique chunks (five-byte) in different file sizes (from 2KB to 100 MB) file

File size	Number of unique chunks	Proportion of total chunks(%)
2KB	872.4	81.24%
5KB	2633.7	58.18%
10KB	4655.9	47.87%
100KB	31133.3	30.13%
200KB	41372	20.11%
1MB	91400.7	8.57%
8MB	311557.5	3.65%
10MB	217023.3	2.03%
100MB	1161093.8	1.14%

file size but stays in a consistent range. For a 10 MB file, on an average number of unique chunks are only 211 K. Thus, we can safely assume that the number of unique chunks so generated will never reach the theoretical limit and populate the hash table with large number of indices. Instead, the hash table so constructed will always remain in RAM memory only and not get exhausted fully for reasonable RAM sizes.

The same reasoning applies for digest size as well. To save memory space, the digest of a file (represented as $<$ rolling hash, chunk score $>$ tuple)contains only non-zero frequency chunks and their corresponding rolling hash indices. Thus, as the number of unique chunks never reach the theoretical limit, our file digests so generated also are of limited size and contained with RAM memory only.

On the other hand, because of reducing the rolling hash size to 5-bytes, the sample space for total possible rolling hash values so generated decreased, leading to a decrease in hash table size so constructed and faster lookup consequently. Also, the file digests so generated were comparatively smaller in size. Hence, the time taken to compare the file digests automatically decreased.

6.3.2 Redefining the document-frequency data-structure

As discussed in section 6.1, in order to calculate the importance of a chunk in a document, we need to know the document frequency. Document frequency is pre-calculated using a big corpus of documents. In `FbHash-E` also, we use 1k files of various size and type from T5 corpus. The chunk document-frequency dataset is large and static. In `FbHash`, it is stored in a dictionary or a look-up table to be used at run-time to discover the relevance of a chunk. To calculate the document frequency, we used around 1k variable size documents. Our document frequency dictionary contains approximately 6M chunks, which requires huge memory space and impacts the performance of the tool. To solve this issue, we propose a novel bloom filter based document frequency calculation that has never been used in any approximate matching forensic tool earlier. We observe that to determine the relevance of a chunk we do not need exact document-frequency, instead we just need a rough estimate of its document-frequency (i.e. low, medium, high), i.e., each document frequency can be mapped to its representative pre-chosen threshold value. To obtain these representative threshold values, We performed k-means clustering [67] to generate clusters, and reduced the unique document-frequency values to cluster centroid value. The number of clusters has been chosen after empirical analysis so that we can fairly represent all document-frequencies (preserve the unbiased representation of document-frequency spectrum). We omitted certain clusters with lower frequencies, assuming that chunks with such document frequency do not hold high importance. The remaining clusters are represented using bloom filters. Bloom filter is a space-efficient data-structure, designed to test the membership of an element in a set quickly and memory efficiently [5]. Each chunk in every cluster is inserted into the corresponding bloom filter by calculating its rolling hash. The size of the rolling hash digest is 40-bit. The digest is divided into three 19-bit sub-hashes by appending zero at MSB (most-significant bit) position. Each sub-hash is inserted into a 64 KByte (2^{19}-bits) bloom filter, where the maximum number of elements per bloom filer is 27000. Hence, the expected false positive rate calculated according to Equation 6.1 is 0.0029.

$$\text{Pr[False Positives]} = \left(1 - [1 - \frac{1}{m}]^{kn}\right)^k \approx \left(1 - e^{\frac{-kn}{m}}\right)^k \quad (6.1)$$

Here, m (=2^{19}) is the number of bits in the bloom filter array, k (=3) is the number of hash functions and n (=27000) represents the number of inserted elements. The total memory required to store the 6 bloom filters is thus just 384 KB which is very small and can be easily loaded into RAM. We chose the values of number of bloom filters, m,n and k keeping in mind - maintaining a balance between the false positive rate that should be as low as possible (below 1%) and RAM memory space occupied by bloom filters to be below 1GB.

In order to access the document-frequency of each chunk we need to check in each bloom filter until the chunk appears in the bloom filter. For that reason also, the number of cluster, size of the bloom filter, and other parameters are chosen very carefully based on several empirical assessment/observation, so that

[5]https://en.wikipedia.org/wiki/Bloom_filter

smaller number of bloom filters can be used (leading to faster runtime) without compromising the accurate representation of document frequency spectrum (while preserving the unbiased representation of document frequency spectrum).

We show in the next subsection, that despite document frequency approximation, performance of our proposed tool in terms of accurate similarity detection does not degrade much compared to `FbHash`. Incorporating the above changes to `FbHash`, we now call our new version as `FbHash-E`.

6.3.3 Performance Comparison of `FbHash-E` with `FbHash`

In the earlier subsections, we have already shown that `FbHash-E` has a comparatively lower memory footprint compared to `FbHash`. In this subsection, we present the performance improvement because of design changes in `FbHash-E` by presenting the analysis of *Fragment Detection Test* performed in [18]. We generate a *base folder* containing original text files from the t5 corpus and a *target folder* comprising of all the fragments (of different sizes) for all the base folder files. We then compare the two folders for similarity against each other. For more details, refer [18]. For this test, `FbHash` took a total time (for digest generation and comparison) of around *1.2 minutes* (around *77 seconds*) whereas `Fbhash-E` completed the test in just 13 seconds showing a significant improvement in runtime.

Tool Accuracy. The Introduction of bloom filters introduces a minor inaccuracy in the similarity score due to approximation of the *document frequency* to a fixed set of values. The reduction in chunk size (leading to a lesser number of unique chunks being generated) leads to few false positives since the probability of the random occurrence of a 5-byte string ($1/2^{40}$) is higher than the probability of random occurrence of 7 bytes string($1/2^{56}$). Altogether the similarity score produced by `FbHash-E` is slightly higher than the actual similarity between the two files and similarity score generated by `FbHash`.

Fig. 6.2, shows the similarity score comparison 'between Fb-Hash and FbHash-E. Fig. 6.2 depicts the results of the fragment identification test (more details can be found in [18] section 7.1). The test is performed on a dataset of 960 random fragments generated from 40 variable size text files(from T5 corpus [64])). In the graph 6.2, X-axis represents the fragment sizes. The first Y-axis(left) represents the *Match Percentage. Match percentage or detection rate* represents the percentage of test samples where for a tool is able to detect similarity by producing a valid match score. It is illustrated in the form of lines in the graph. The second (right) Y-axis represents the average similarity score calculated by the FbHash and FbHash-E. The bars in the graph illustrate the average score. FbHash-E and FbHash are represented by blue and green colors respectively in the graph. For example, for 95% fragments, we calculated the similarity score between each file and its 95% fragment. The average similarity scores generated by FbHash-E and FbHash are 98.4 and 97.2(out of total of 100) respectively.

From the graph, we can see that there is, on an average increase of 7.5 in the similarity score generated by FbHash-E as compared to `FbHash`. To mitigate this limitation, we suggest increasing the threshold similarity score of `FbHash-E` to 28 [6] compared to a threshold score of 16 in `FbHash`.

[6]threshold similarity score below which `FbHash-E` will consider two files to be dissimilar

To represent the accuracy, we have computed F-score. F-score is the harmonic mean of precision and recall. Precision represents the proportion of the positive results that were correctly predicted, which means how many similar files were found similar by the tool. Recall signifies the proportion of the similar files that was resulted similar. Precision is a good measure when the cost of False Positive is high. The recall is the best metric to determine the quality of the classification if there is a high cost associated with False Negative. In the case of the approximate matching algorithm, the cost of false negative is much higher than false positive. For example, if approximate matching filtering misses an evidence file suspicious file. Hence, Recall should be assigned a higher weight than precision to compute the correctness of the tool. For the same reason, we have calculated both F and F2-score. F-score and F2-score are computed using the following formula:

$$F - score = \frac{2.precision.recall}{precision + recall}$$

$$F_\beta - score = \frac{(1+\beta^2).precision.recall}{\beta^2 * precision + recall}$$

Where β is 2.

Table 6.2: F-score

	ssdeep (t=22)	sdhash (t=22)	LZJD (t=20)	FbHash (t=16)	FbHash-E (t=28)
F-score	0.67	0.87	0.52	0.93	0.87
F2-score	0.56	0.81	0.47	0.90	0.87

In table 6.2, we present the F-score and F2-score results of `FbHash-E` along with other state-of-the-art techniques such as ssdeep, sdhash, LZJD, and FbHash. From the table, it can be seen that `FbHash-E` detects similarity with an accuracy of 87% (considering F2-score is the appropriate measure of accuracy here). Whereas accuracy of `ssdeep` is 56%, `sdhash` is 81%, LZJD is 0.47, and FbHash is 90%. Because of the proposed amendment in FbHash-E several false positive has been introduced, which impacts the precision and consequently impacts the F-score and F2-score. Although FbHash-E has a lower false-negative and higher recall rate. There is only a 3% reduction in accuracy compared to FbHash. FbHash-E yet outperforms against all other state-of-art tools.

It is evident that in `FbHash-E` there is a tradeoff between accuracy and performance. The memory and runtime performance of `FbHash-E` is better but the accuracy in detecting similar files is lower compared to `FbHash`. However, in Section 6.5, we evaluate the security and robustness of `FbHash-E` against standard benchmark tests and show that even after increase in false positives (compared to `FbHash`), our tool outperforms all the other state-of-the art approximate matching tools in all the security tests. Thus, our proposed version `FbHash-E` is fast as well as secure, i.e., the design changes introduced in no way affect the security of the forensic tool.

6.4 Performance Comparison with other Tools

In this section, we compare the performance of `FbHash-E` with other benchmark hashing tools, i.e., `ssdeep`, `sdhash` and `LZJD`. We ran all the tools on the following testbed:

- Testbed: Conventional standard notebook laptop hav-

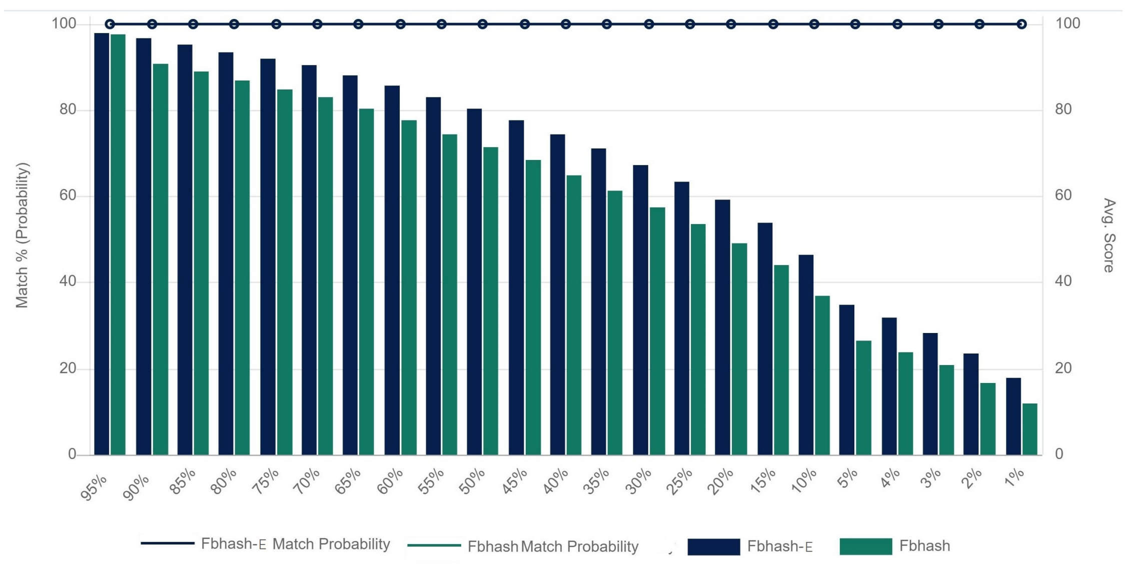


Figure 6.2: FbHash vrs FbHash-E

ing Windows 10 OS and Quad-Core Processor: Intel(R) Core(TM) i9-10885H CPU clocked at 2.40GHz with 32 GB RAM. The current implementation requires java version 1.8.0 or above.

6.4.1 Runtime Efficiency

For the results, we ran the *fragment detection test* discussed in Section 6.3.3 and in more details in Section 7.1 in [18]. To define the runtime efficiency we measured the time using java in built function. This includes all elapsed time including time slices used by other processes and blocking time, i.e., if it is waiting for some I/O to complete.

- Time taken in Preprocessing stage: `FbHash-E` requires document frequency calculation for its digest creation. This is a one-time calculation for all the files in the reference corpus

. Hence, this step is a precomputed step which can be done anytime offline by the investigator and only once. During online phase, creation of bloom filters for document frequency is not done and thus does not consume any runtime processing. As the other testing tools mentioned above, do not have any preprocessing, we omit them from comparison in this stage.

- Time required for Digest Creation - This measures the time taken for reading all the input files from the system and then creating the input digest for each file. We report the results for this stage in Table 6.3. As can be seen that in this stage, `ssdeep` is the fastest - both for base and target folder digest creation followed by `sdhash`, `LZJD` and then `FbHash-E`. For smaller file sizes (e.g., size = 1.88 MB), time taken by `LZJD` and `FbHash-E` are comparable.
- Time taken for Digest Comparison - This measures the time for comparing the digests of two targets after respective digests have been created. Here, we perform a brute force comparison of two folders generated in fragment detection test,i.e., an all-against all digest comparison which is the standard practice followed in other related works. We report the implementation results in Table 6.3. From the table,it can be seen that `ssdeep` outperforms all including our tool in comparison stage. This is followed by `sdhash`, `LZJD` and then `FbHash-E`. Again in this stage, time taken by `LZJD` and `FbHash-E` are comparable.

From the table 6.3, it is evident that compared to other existing tools, `FbHash-E` is slower. This is attributed to its larger file digest creation which subsequently affects the comparison

Table 6.3: Comparative runtime analysis (in ms) of all tools. The *Total Time* column reflects the total time taken for digest creation and comparison.

S.No	Stage	ssdeep	sdhash	LZJD	FbHash-E
1.	Digest Generation (no. of file: 20 size: 1.88MB)	76.23	90.05	1125.90	1802
2.	Digest Generation (no. of file: 480 size: 18.4MB)	272.05	325.227	1482.89	7773
3.	Comparison (no. of comparison: 9600)	60.85	120.35	1124.08	6023
	Total	409.13	535.62	3732.87	15598

stage. However, our run time is usable as it is not exponentially higher than others. Moreover, running time of our tool can be further enhanced in future using GPU-based implementations as done in the case of `sdhash`. GPUs have far more cores than Central Processing Units (CPUs). This parallel architecture can be utilized to render high computing performance. According to a study conducted by Buber et al. [68] GPU can be 4-5 times faster than CPU. We suggest that this apparent slowdown in the performance of `FbHash-E` is negated by the fact that our tool is the most accurate, consistent, secure and robust against active adversary attacks when compared with other state-of-art tools as shown in Section 6.5. We believe that as long as a forensic tool is usable, security should be the prime criteria to be considered whenever evaluating the overall effectiveness of any forensic tool and here `FbHash-E` has a clear lead over others. We show a detailed security analysis of `FbHash-E` and its comparative analysis in Section 6.5.

6.5 Testing with benchmark problems

In this section, we evaluate the security and robustness of `FbHash-E` against standard benchmark tests. We divide the tests into 3 categories - *Containment Tests*, *Resemblance Tests* and *Consistency Tests*. We also provide a comparative analysis with other approximate algorithms - i.e., `ssdeep`, `sdhash` and `LZJD`.

6.5.1 Consistency Test

For any forensics tool, consistency in the results obtained is a very important metric. By consistency, here, we mean uniformity among the scores produced. A tool would not be very valuable if it produced non-uniform similarity scores for different files under a given test against a fixed absolute score. On the other hand, the tool which performs vice-versa would be more effective. Through this test, we want to compare and analyse the consistency of all the tools under different tests conducted in the subsequent sub-sections. As the discussion of this test depends on the test results obtained in subsequent sections, we discuss the implications of this test in each of the next sections separately.

6.5.2 Resemblance Test

Let D_1 and D_2 denote two data objects. Resemblance tests provide a measure of similarity and commonality between the two data objects D_1 and D_2. We performed following tests under this category:

1. Random Noise Resistance Test - In this test, we randomly alter few bytes of the file and then check the similarity be-

tween the original file and the modified file. Byte locations are picked randomly and the picked byte value is modified by adding a random number(can also be negative) to it. For the test, we give $M\%$ of the file bytes (with respect to the original file size) to be altered as the input. We vary the size of M between - 0.01% to 10% of the original file size and take a ceil value on the total number of bytes to be changed. E.g., If the length of the original file is 4982 bytes and we need to make 0.01%change, then $\lceil 4982 * (0.01/100) = 0.4982 \rceil \implies$ 1 byte of the file chosen randomly will be changed. The results of this test are summarized in Table 6.4.

Table 6.4: Comparative Analysis of Random Noise Test. Here, Avg.(NZ) denotes average score over non-zero values.

Tool		.01%	.05%	.25%	1%	5%	10%
	Abs. Score	99.9	99.9	99.7	99	95	90
ssdeep	Avg. Score	86.15	71.6	40.45	0	0	0
	Avg.(NZ)	90.94	84.23	67.41	-	-	-
	Matches (%)	95	85	60	0	0	0
	Std. Dev	13.85	13.30	9.49	-	-	-
sdhash	Avg. Score	96.6	91.35	68.4	18.15	0	0
	Matches (%)	100	100	100	100	0	0
	Std. Dev	5.44	5.82	8.76	9.53	-	-
LZJD	Avg. Score	78.05	71.9	58	52.35	38.4	29.7
	Matches (%)	100	100	100	100	100	100
	Std. Dev	20.43	19.37	9.66	7.11	5.97	4.96
FbHash-E	Avg. Score	98.95	98.9	97.85	94.35	77.85	61.9
	Matches (%)	100	100	100	100	100	100
	Std. Dev	0.22	0.30	0.48	1.03	1.75	1.48

In table 6.4, the *absolute score* denotes the actual percentage ratio similarity between original file F and modified file F'. Let us assume the original file is of x-bytes size, then the modified file will also consist of x-bytes only but with some bytes altered in it. If $M\%$ bytes have been changed in the modified file, then absolute score is computed as follows:

$$\text{Absolute Score} = \frac{x - (M/100)x}{x} \times 100 \qquad (6.2)$$

E.g., if $M = 5\%$, then Absolute Score becomes 95% similarity. Also,*Matches* represents how many percent of all files gave a similarity score.

From the table, it is clear that `FbHash-E` performs the best and is most resistant to random byte modifications compared to other tools. `FbHash-E` generates a matching score that is very close to absolute score for $M \leq 1\%$. `sdhash` closely follows next, however, beyond $M \geq 5\%$, it fails to give any matching score. For $M = 1\%$ also, it generates a very poor similarity score. In case of `ssdeep` and `LZJD`, the results do not look promising and similarity scores seem far away from the actual absolute score. For this test, `ssdeep` is not able to generate similarity scores for all of the files in the corpus for any of the bucket considered. Also, for `LZJD`, the std. deviation obtained for similarity scores in all the buckets seem to be very high (in one case, it is as high as 20.43) highlighting the fact that it is not producing uniform results in a single bucket for all the files tested. Thus, in summary, `FbHash-E` generates the best results and is most resistant to random changes in the file.

Consistency Test. In Table 6.4, if we compare the standard deviation score of each tool, we can see that `FbHash-E` generates the most consistent average score under each bucket (i.e., each column) with lowest standard deviation respectively. While standard deviation values of FbHash-E never go beyond 2 in any of the bucket, the standard deviation scores of ssdeep and LZJD are very high for $M \leq 0.25\%$ with a maximum std. dev. score of 13.85 and 20.43, respectively. This shows that FbHash-E results have less variation and are more uniform compared to other ssdeep and LZJD where variability in the scores (under a particular bucket) is quite high. In terms of consistency, ssdeep appears to be the next best, however, it doesn't produce results for all the files in any of the bucket considered.

2. **Multiple Common Block Identification** - This test was first proposed by Vassil Roussev in the paper [3]. This test examines the ability to identify two objects that share multiple common blocks of data dispersed over the object—for example, document version identification. To perform this test, we generate a dataset with multiple common blocks as follows using text files from T5 dataset. We take three files and resize them to the size of the smallest file amongst them. We create 10 fragments of the first file of following sizes: 100%, 66.66%, 42.86%, 25%, 11.11%, 5.2%, 4.1%, 3.09%,2.04%, 1.01%. Each fragment is further divided into x pieces (where x $\in$ {2,4,8,16,32}) one by one. Each piece is inserted into the second and third files at randomly chosen positions. Doing the above for each value of x and all 10 fragments, produces 50 pairs of files with shared multiple common blocks of data. The fragment sizes are chosen in such a way so that we can generate file pairs with similarity (/commonality) of 50%, 40%, 30%, 20%, 10%, 5%, 4%, 3%, 2%, 1% receptively.

The commonality (on the scale of 0 to 100) between the generated pairs can be calculated using the fragment size as follows:

$$\text{Absolute Score} = \frac{f}{100 + f} \times 100 \tag{6.3}$$

Here f indicates the fragment size where f ∈ {100, 66.66, 42.86, 25, 11.11, 5.2, 4.1, 3.09,2.04, 1.01}. Since each fragment is divided into 2,4,8,16,32 pieces and inserted into the second and third files, the first five file pairs, generated by the previous step share 50% commonality spread across the document divided into 2, 4, 8, 16, 32 parts. Similarly second five pair shares 40% similarity, the next five pairs share 30% similarity, and so on. ‘Similarly, we have generated the dataset of 800 text file pairs with shared multiple common blocks. The result of this test is presented in table 6.5.

Table 6.5: Comparative Analysis of multiple common block test. Here, Avg.(NZ) denotes average score over non-zero values.

Tool		50%	40%	30%	20%	10%	5%	4%	3%	2%	1%
	Abs. Score	50.0	40.0	30.0	20.0	10.0	5.0	4.0	3.0	2.0	1.0
ssdeep	Avg. Score	10.04	3.40	1.72	0.30	0	0	0	0	0	0
	Avg. Score (NZ)	50.81	43.9	36.26	40.33	-	-	-	-	-	-
	Matches (%)	19.75	7.75	4.75	0.75	0	0	0	0	0	0
	Std. Dev	21	12.24	7.92	3.59	0	0	0	0	0	0
sdhash	Avg. Score	17.35	10.75	6.12	2.81	0.82	0.25	0.18	0.12	0.06	0.01
	Matches (%)	95.50	79.50	58.00	37.25	19.75	11	9.5	8.5	6.25	1
	Std. Dev	11.13	9.94	7.59	4.66	1.9	0.77	0.59	0.4	0.24	0.1
LZJD	Avg. Score	22.15	19.75	16.44	13.01	9.68	8.45	8.14	7.90	7.45	6.69
	Matches (%)	100	100	100	100	100	100	100	100	100	100
	Std. Dev	8.06	5.91	6.72	5.89	6.08	6.04	6.07	6.22	5.68	5.82
FbHash-E	Avg. Score	57.32	49.82	42.75	34.94	24.97	18.47	17.02	15.62	14.41	13.07
	Matches (%)	100	100	100	100	100	100	100	100	100	100
	Std. Dev	12.73	13.61	13.89	14.13	14.38	14.75	14.93	14.89	14.88	15.31

In table 6.5 the absolute score denotes actual similarity between two files. Average score indicates the average score of the all the file that share a same commonality, where f ∈ { 50%, 40%, 30%, 20%, 10%, 5%, 4%, 3%, 2%, 1%}. Match Percentage represents the percentage of all the files where the tool can detect similarity between the related documents. Standard deviation denotes the variation between the computed similarity score for the files sharing the same commonality.

As shown in the table, ssdeep and sdhash both performed poorly in this test. ssdeep does not generate a valid similarity score for commonality less than 10%. From the match percentage, we can see ssdeep results similarity for only 20% of the file with commonality 50% and lower match percentage for other that is why have also shown the average of all non-zero score. sdhash performs better than ssdeep and finds similarity with higher probability (as we can see the match %). Although the produced score does not represent the actual similarity for 50% commonality, it results on an average 17.35 similarity score, and for 10%, it results in similarity score below 1. LZJD and FbHash detect similarity for all similar files with a match percentage of 100. However, the similarity scores generated by LZJD are always between 6 to 22 despite the actual similarity range of 1 to 50. For files with more than 10% commonality FbHash results, similarity score closer to the absolute similarly and for 10% and smaller commonality, LZJD results score closer to the absolute similarity.

Consistency Test. In Table 6.5, if we compare the standard deviation score of each tool, we can see that `LZJD` generates the most consistent average score under each bucket (i.e., each column) with the lowest standard deviation respectively. However, the difference between the similarity score of 1% commonality and 50% is less than 20. This is not the case with FbHash. The similarity score generated across the different commonalities are more consistent, the difference between 1% commonality and 50% is about 44 (closer to absolute difference). sdhash results are more consistent than ssdeep but do not represent the actual similarity.

3. **Code Version Identification** - This test evaluates the approximate matching tool's ability to detect different versions of the software and program files. This test was first mentioned by Vassil Roussev in [3]. To perform this test, we have used the NSRL dataset of windows program files [7]. We use 887 executable files obtained from different versions of 10 generations of Windows operating system release. Table 6.6 shows the result of this test. Since for this test, we did not generate the dataset; we do not know the absolute similarity. Therefore for this test case, we represent the performance using the confusion matrix. We compute the true positive, true negative, false positive, and false negative based on the suggested threshold value of each tool to separate the matches from non-matches. If the similarity score generated by the approximate matching tool is greater than or equal to the threshold, it is considered as a positive or correct result otherwise negative. The suggested threshold for sdhash is 22 for near-perfect detection rate [13]. Since ssddep does not specify any threshold value, we use the same threshold 22 for ssdeep. For LZJD, we use the default threshold used by LZJD tool, which is 20 [7]. In the case of FbHash, the suggested threshold was 16. However, due to the amendments in the FbHash-E as discussed in section 6.3.3, the generated similarity score is

[7]https://github.com/EdwardRaff/jLZJD

Table 6.6: Code version identification test

	ssdeep (t=22)	**sdhash** (t=22)	**LZJD** (t=20)	**FbHash-E** (t=28)
True Positive (TP)	4605	5458	9577	18644
True Negative (TN)	753556	753117	747494	745352
False Positive (FP)	1712	2151	7774	9916
False Negative (FN)	26896	26043	21924	12857
Precision	0.728	0.7173	0.5519	0.6528
Recall	0.146	0.1732	0.3040	0.5918
F-score	0.243	0.2791	0.3920	0.6208
F2-score	0.1737	0.2041	0.3340	0.6030

higher. To compensate for that, we suggest a higher threshold for FbHash-E, which is 28. As we can see in the table 6.6 FbHash-E outperforms in comparison to all other tools. Using the confusion matrix, we compute false-positive rate, false negative rate, precision, recall and F-score. Precision represents the proportion of the positive results that were correctly identified similar. Recall indicates represent the proportion of similar document resulted similar. F-score is harmonic mean of precision and recall, representing accuracy. FbHash results least false-negative rate and highest precision and recall rate. F2-score represents the accuracy. Precision. recall and F-score can range between 0-1, where 1 indicated the highest value and 0 represents the lowest. The results show that the result obtained by FbHash have lowest false negative rate and significantly higher accuracy compared to ssdeep, sdhash, and LZJD.

6.5.3 Containment Test

Let D_1 and D_2 denote two data objects. Containment tests provide a measure of containment of one object into another, e.g.,

D_1 into D_2 . We performed following tests under this category:

1. Alignment Tests - This test was introduced in [11]. Here, the original file is modified by inserting byte sequences at the beginning of the input and then the impact of this modification is compared with the original file. The byte sequences are added according to the following two methods:

   - **Percentage length block**: In this test, we insert a block size of $M\%$ (with respect to the original file size) at the beginning of the original file. We vary the size of M between 10% to 400%, i.e., the block sizes considered are - 10%, 50%, 100%, 200%, 300% and 400% of the original file size respectively. The results of this test are summarized in Table 6.7.

Table 6.7: Comparative Analysis of Alignment Test. Here, Avg.(NZ) denotes average score over non-zero values.

Tool		10%	50%	100%	200%	300%	400%
	Abs. Score	90	66	50	33	25	20
ssdeep	Avg. Score	93	70.6	51.1	11.45	1.45	0
	Avg.(NZ)	93	74.31	60.11	45.8	29	-
	Matches (%)	100	95	85	20	5	0
	Std. Dev	4.72	17.45	23.0	20.04	6.32	-
sdhash	Avg. Score	87.65	87.15	86.9	87.5	89.9	90.5
	Matches (%)	100	100	100	100	100	100
	Std. Dev	13.3	16.8	15.8	10.44	12.0	11.6
LZJD	Avg. Score	41.5	24	16.95	10.8	7.9	6.6
	Matches (%)	100	100	100	100	100	100
	Std. Dev	4.90	3.65	3.67	3.51	3.19	3.377
FbHash-E	Avg. Score	91.5	72.45	60.7	46.95	39.75	35.2
	Matches (%)	100	100	100	100	100	100
	Std. Dev	2.24	5.22	5.31	5.25	4.77	4.42

In table 6.7, if a file comprises of x-bytes, then absolute score is computed as follows:

$$\text{Absolute Score} = \frac{x}{x + (M/100)x} \times 100 \qquad (6.4)$$

E.g., if $M = 10\%$, then Absolute Score becomes 90% similarity. Also,*Matches* represents how many percent of all files were matched. From the table, it can be seen that all algorithms except `ssdeep`, are able to provide similarity scores for all the files in the corpus for all the modifications done, i.e., their match score is 100%. However, `ssdeep`, is not able to provide match for all the files present in the corpus starting with $M = 50\%$ and match score significantly drops beyond $M \geq 200\%$. For $M = 400\%$, ssdeep was not able to provide similarity score for any of the files in the corpus,i.e., Match Score = 0.

As a consequence, it makes sense to calculate Average Score over only the non zero similarity scores for `ssdeep`, termed as *Avg.(NZ)* and compare it with average scores of other algorithms. From the table, it can be seen that `FbHash-E` performs the best and gives matching score close to absolute value. When modifications are too large, i.e., $\geq$200%, `ssdeep` is hardly able to detect any similarity (as number of matches significantly drop) whereas `sdhash` gives too high unrealistic matching score, e.g., when added block size is $M = 400\%$, it gives an average similarity score of 90.5% as against an absolute score of 20%. The performance of `LZJD` also does not look promising as it gives too low similarity scores even when added percentage block is as low as $M = 10\%$. This could be attributed to LZSet algorithm used by LZJD, which builds LZSet incrementally from the beginning of the file.

Thus, from table 6.7, it can be summarized that `FbHash-E` gives the best similarity scores in terms of closesness to actual *Absolute score* and *Match percentage.*

Consistency Test. In Table 6.7, if we compare the standard deviation score of each tool, we can see that both `FbHash-E` and `LZJD` generate low standard deviation values under each bucket (i.e., each column) with maximum std. dev. value being less than 5 in each of the tools respectively. However, as mentioned earlier, `FbHash-E` produces similarity scores much closer to actual absolute score as compared to `LZJD` and hence performs better. On the other hand, `ssdeep` and `sdhash` both generate very high std. dev. values (value more than 10) in most of the buckets indicating scores being produced are far from the mean similarity score for many of the files under each bucket for both these tools,i.e., their results are not consistent.

- **Fixed length block**: This test is similar to the previous one, except that here, a fixed block of M bytes (irrespective of the file size) is added at the beginning of the file. In tables 6.8 and 6.9, we present the results for fixed block of sizes: 1KB, 4KB, 16KB, 32KB and 64KB respectively. Here, if file size is yKB and fixed block size = M KB, then absolute score is calculated as:

$$\text{Absolute Score} = \frac{y}{y+M} \times 100 \qquad (6.5)$$

From the above equation, it can be seen that here, absolute score depends on particular file size. For smaller values of M, files with smaller y will have larger impact than files with greater y. In our corpus, the size of text files vary from 5KB to 800KB. To better analyze the effect of "fixed block" test, we split our dataset into 2 parts: *Dataset 1* comprising of files with size less than 50 KB, i.e., $y \leq 50$ and *Dataset 2* comprising of files with size more than 50 KB, i.e., $y > 50$. For each dataset, we calculate the absolute score for all the files present in it according to Equation 6.5 and then average it over the total number of files. Thus, we report the average absolute scores in each of the tables.*Matches* is the percentage of files that gave results.

From table 6.8, for *Dataset 1* having files with size less than or equal to 50KB, it can be seen that none of the tools generate much accurate results. `ssdeep` fails to generate similarity score for majority of the files when size of fixed block added is $\geq$ 16 KB (matches significantly drop to below 7%). For the rest of the tools, the results are a mixed bag. While for $M \leq 4$ KB, `FbHash-E` and `sdhash` generate a closer similarity score to absolute value, for $M \geq 16$ KB, `LZJD` seems to generate closest similarity scores (although lower) when compared against the average absolute score.

In comparison, from Table 6.9, it can be seen that, in *Dataset 2*, when files tested are of size $>$ 50 KB, `FbHash-E` performs the best as similarity scores reported

Table 6.8: Comparative Analysis of Alignment Test for fixed block size over Dataset 1 ($y \leq 50$ KB). Here, Avg.(NZ) denotes average score over non-zero values.

Tool		1KB	4KB	16KB	32KB	64KB
	Abs. Avg. Score	89.85	69.5	37.31	23.26	13.31
ssdeep	Avg. Score	79	55.28	4.14	3.28	0
	Avg.(NZ)	79	55.28	58	46	-
	Matches (%)	100	100	6.25	6.25	0
	Std. Dev	5.16	9.83	-	-	-
sdhash	Avg. Score	95.5	85.92	87.42	91.57	86.21
	Matches (%)	100	100	100	100	100
	Std. Dev	4.86	15.86	15.02	7.70	15.90
LZJD	Avg. Score	45.21	30.71	18.28	12.21	7.35
	Matches (%)	100	100	100	100	100
	Std. Dev	5.11	7.06	3.63	3.32	2.19
FbHash-E	Avg. Score	94.21	84.57	64.57	54.42	44.57
	Matches (%)	100	100	100	100	100
	Std. Dev	1.77	3.84	5.96	6.13	5.72

by it closely resemble the actual average absolute score. After `FbHash-E`, `ssdeep` seems to produce the next best results whereas both `sdhash` and `LZJD` generate very low similarity scores far away from the actual absolute score.

In summary, for this test, when files in the corpus are of less size, i.e., $y \leq 50$ KB, none of the tools seem to produce accurate similarity scores whereas for larger files, i.e., $y > 50$ KB, `FbHash-E` performs the best.

Thus, it can be seen that when random bytes are added at the start of the file, `FbHash-E` is the most robust of all and is able to detect similarity between altered file and actual file with greatest accuracy.

Table 6.9: Comparative Analysis of Alignment Test for fixed block size over Dataset 2 ($y > 50$ KB). Here, Avg.(NZ) denotes average score over non-zero values.

Tool		1KB	4KB	16KB	32KB	64KB
	Abs. Avg. Score	99.51	98.09	92.92	87.08	77.86
ssdeep	Avg. Score	98.6	98.6	91.6	87.8	82.2
	Matches (%)	100	100	100	100	100
	Std. Dev	0.8	0.8	5.27	11.77	15.09
sdhash	Avg. Score	87.4	37.8	37.6	70.6	32.6
	Matches (%)	100	100	100	100	100
	Std. Dev	0.48	2.48	4.88	1.01	4.58
LZJD	Avg. Score	47.8	47.8	45.8	41.6	36
	Matches (%)	100	100	100	100	100
	Std. Dev	9.28	9.49	5.52	5.38	1.78
FbHash-E	Avg. Score	99	98.4	95	91.8	87.4
	Matches (%)	100	100	100	100	100
	Std. Dev	0	0.8	3.28	5.19	7.2

Consistency Test. For dataset 1 ($y \leq 50KB$), it can be seen that both `FbHash-E` and `LZJD` have small values of standard deviation in all of the columns whereas `sdhash` seems to perform worst with very high standard deviation. Even `ssdeep` results are not very consistent and for $16KB \leq y \leq 64KB$, it fails to generate similarity scores for almost all of the files under consideration. On the other hand, for dataset 2 ($(y > 50KB)$), it is clear from Table 6.9, that `FbHash-E` gives the most consistent results as its standard deviation values are lowest and the average score generated is most closest to the actual similarity score. `sdhash` too generates small std. dev. values across all the columns, but if viewed as a tuple <Avg. score, Std. dev.>, the average scores generated by `sdhash` are far away from the actual similarity score and thus low std. deviation in generated scores does not hold any value.

6.6 Summary

In this chapter, we showed that the current design structure of `FbHash` is slower and memory intensive compared to its peers. We then proposed a novel bloom-filter based efficient version, `FbHash-E`, that has much lower memory footprint and is computationally faster compared to `FbHash`. While the speed of `FbHash-E` is comparable to other state-of-the-art tools, it is resistant (like its predecessor) to "intentional/intelligent modifications that can fool the tool" attacks, unlike its peers. Our version thus renders `FbHash` fit for practical use-case. We performed various modification tests to evaluate the security and correctness of FbHash-E. Our experimental results showed that our scheme is secure against active attacks and detects similarity with 87% accuracy. Compared to `FbHash`, there is only 3% drop in accuracy results. We have demonstrated the sensitivity and robustness of our proposed scheme by performing the containment and resemblance test. We have shown that FbHash-E can correlate files with up to 10% random-noise with 100% detection rate and is able to detect commonality as small as 1% between two documents with an appropriate similarity score. We also showed that our proposed scheme performs significantly better than other schemes in code version identification test.

Chapter 7

Essential Characteristics of Approximate matching algorithms : A Survey of Practitioners Opinions and requirement regarding Approximate Matching

This chapter presents the findings of a closed survey conducted among a highly experienced group of federal state and local law enforcement practitioners and researchers, aimed to understand the practitioner and researcher's opinion regarding approximate matching algorithms. The study provides the baseline attributes of approximate matching tools that a scheme should possess to meet the real requirement of an investigator.

7.1 Survey Methodology

7.1.1 Purpose Of the Survey

The baseline definition and terminology of the approximate matching algorithm is already defined by [9]. The NIST Special Publication 800-168 [9] defines the properties at a more general and broader level. However, similarity definition, as well as the requirements, vary for different data object type. For example, files with similar perceived text content may have entirely different structure and would be entirely different if an inappropriate algorithm is applied; the actual similarities would remain unnoticed. A color and the grayscale version of the same image would be completely different when using most of the existing schemes. Hence there is a strong need to define the properties based on the practical requirements.

The aim of the survey was to establish the key characteristics of the approximate matching algorithm based on the requirement of digital forensics practitioners and researchers with the understanding of different perspectives towards these algorithms. Using the results of the survey, in future, our aim is to build the real dataset with known similarity and evaluate the existing schemes based on the key characteristics using the real dataset.

7.1.2 Survey Design

The survey contains a brief introduction to the topic of the study and the purpose of the survey, followed by 10 questions consisting following:

- 4 optional questions
- 1 Multiple choice question
- 4 likert scale questions
- 1 Ranking question

Each of the questions contained one comment section to share any other information regarding the question, which allowed the participants to express their view in more pragmatic and reasonable manner.

The format of the survey was kept simple and short while capturing all the information needed, In order to maintain the balance between the amount of time and effort required to complete the survey. The survey was conducted online among a closed group of 80 digital forensic practitioners; those are part of NIST steering committee and familiar with the use of NSRL for filtering. The audience of the survey was kept limited in order to assure the reliability and legitimacy of the results. The survey was conducted in the entirely anonymous manner, no personally identifiable information about the participants was stored. It was available online for two weeks. Among the 80 invitation, we received total 19 completed responses, which is an acceptable amount of data to analyze and conclude the results of the survey for such highly targeted surveys.

7.1.3 Survey Results

The results of the survey is presented in this section based on the all the received responses. However, where appropriate, further analysis based upon the perspectives of researchers and practitioners is presented.

7.1.3.1 Demographic Information

Demographic Information of the survey is represented in figure 7.1. More than 75% of the participants had more than 10 years experience as shown in figure 7.1. All of the participants were from the United States of America and working as federal, state or local law enforcement practitioner or researchers.

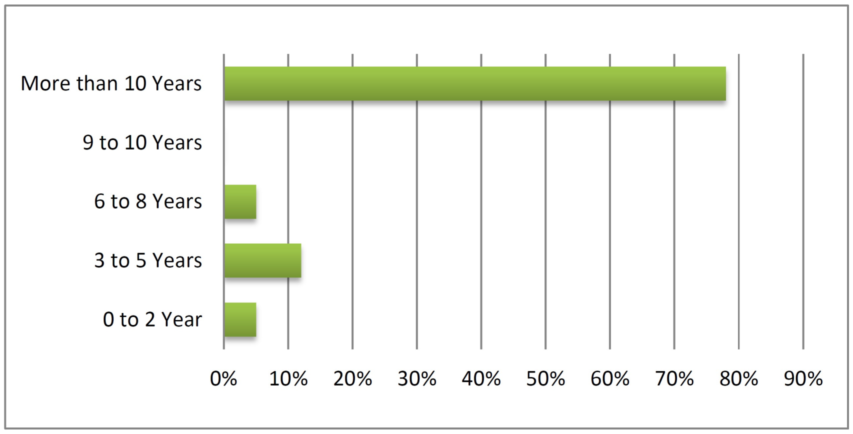


Figure 7.1: Demographic Data

7.1.3.2 Awareness About Approximate Matching Algorithms

Results shown in around 65% of the total participants were aware and are using approximate matching algorithms during the investigation process. Among them, 45% were aware that their tools perform approximate filtering but were unaware of the technique/scheme used by the tool. Whereas remaining were well aware of the techniques used by the tools or the specific technique used by them. Following is the list of schemes mentioned by the participants those are being used by the practitioners these days: Cluster analysis of FTK, FuzzyDocs of X-ways, md5Deep, CodeSuite, PhotoDNA, sdhash, ssdeep and some of the practitioners are using their self-constructed tf-idf based schemes. This shows that most of the practitioners find approximate matching algorithms useful and are willing to use these algorithms to filter out relevant data. However, since there is no consensus for one standard scheme hence all of them are using techniques that are either by default being used by the tools or they find it more useful for a particular case based on their personal experience.

7.1.3.3 Primary Applications of Approximate Matching Algorithms

Participants were asked to scale the uses of the approximate matching algorithm based on their professional experiences. List of applications was derived by the examining the existing literature shown in figure 7.2. Responders were also allowed to add more uses to the list.

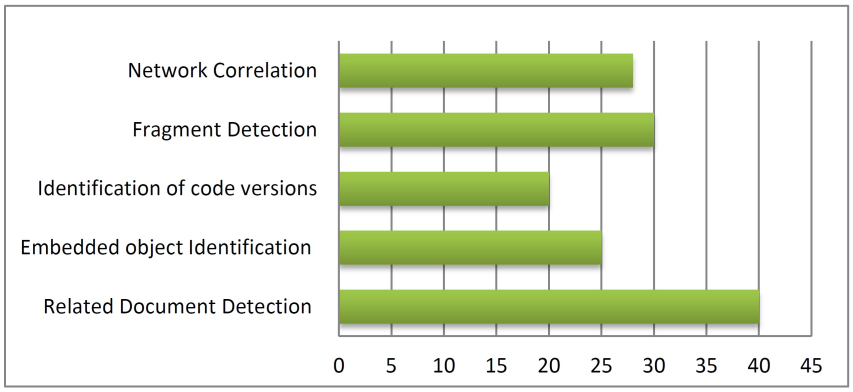


Figure 7.2: Applications of Approximate Matching Algorithms

- Based on all of the received responses approximate matching algorithms are being frequently used to identify the Related Documents.

- Fragment Identification was scaled second in the list. Fragment Identification is the identification of a document based on a piece of data.

- Following are the other important uses in the respective order of their raking

  - Correlation of network (Data packet reconstruction from the fragmented files over the network)
  - Embedded Object Identification (e.g., a jpeg within a word document)
  - Identification of the code version (Identification of patched or upgraded version of software).

  However, approximate matching algorithms don't work well for network correlation if the network is encrypted.

On further examination about the type of the data object that they need to filter out in most frequent cases, the participants were asked to rate among text, image, executable file, and multimedia. All of the received responses show that text and images are the most commonly appeared data types that need to be filtered. Executable and multimedia files also required to be filtered in some of the cases.

Hence we can state that one of the important characteristics of a scheme is its ability of related document correlation and fragment identification for text and image data type.

7.1.3.4 Key Measure to Identify the Ground Truth

Participants were asked choose among the existing measure that represents the similarity of two textual documents most closely in their opinion. Proposed measures were 'Edit distance,' 'Length of longest common sub-string' and 'Length of longest common subsequence.' Where Edit Distance is a way of calculating dissimilarity between two text sequence by counting the minimum number operations required to transform one sequence into the other. Length of longest common substring denotes the length of the common contiguous substring of maximal length. Length of the longest common subsequence of two text sequence signifies the length of the common substring of maximal length where the substring might not appear in contiguous fashion but preserves the ordering of characters. All of these measures were taken by examining the current literature. Participants were also allowed to add new measures. The purpose of this question is to find out the key measure to find real similarity in two text documents. Figure 7.3 shows that based on the experience of the responder 'Length of longest Common Substring' defines similarity between two documents the most. Hence this measure should be used to build ground truth for Text data set with known similarity.

However, by further analysis of the responses, we realized this should be analyzed further to make a persuasive conclusion.

7.1.3.5 Image Similarity

Participants were asked to rank the similar images that they need to filter out during the real case investigations. Figure 7.4 shows that all of the listed image similarities are almost equally important from the practitioner's point of view. However, the ability of a scheme to find out the similarity between different formate is ranked highest. Hence it would be useful if a technique can filter out all of the listed image similarities.

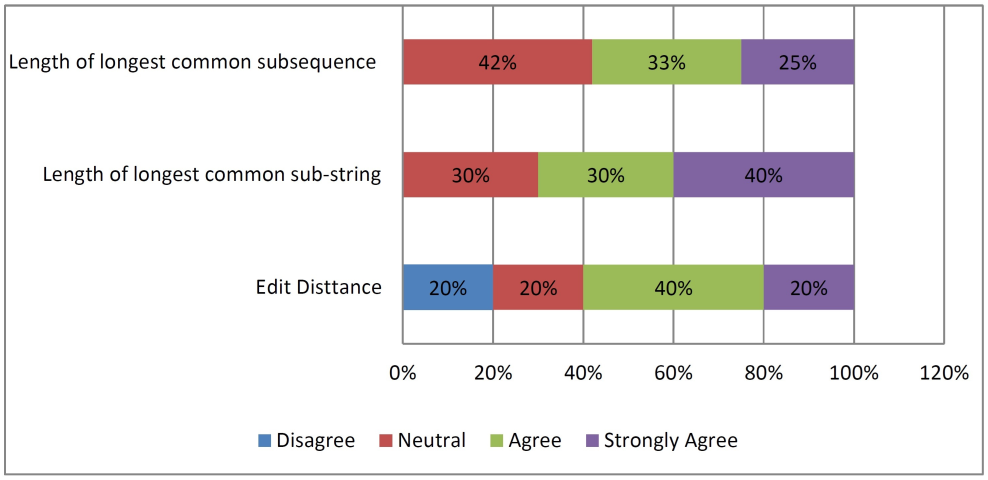


Figure 7.3: Key Measure to Identify the Ground Truth

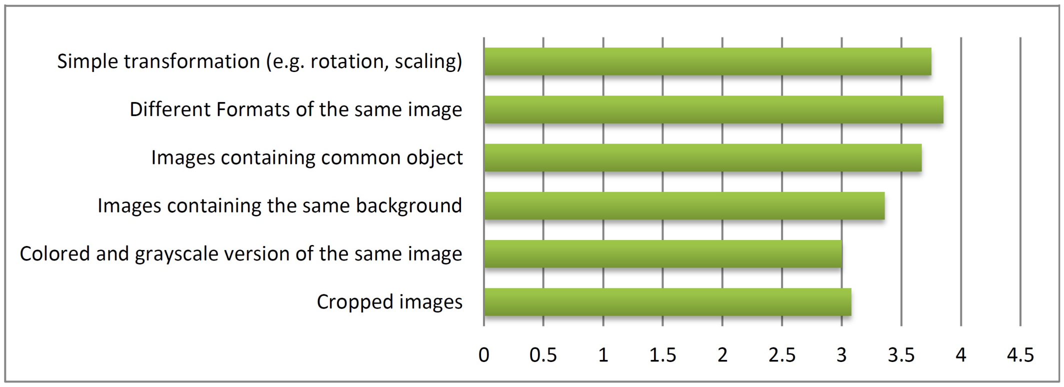


Figure 7.4: Image Similarity

7.1.3.6 Executable Program File Similarity

The purpose of this question to understand similarity definition for a program file based on the requirement of an investigator. Figure 7.5 shows the results. All of the listed similarity characteristics are chosen by examining the current literature and participants were allowed to add missing properties. From the received responses we can say one of the essential abilities of a tool for program file similarity is to find the same program with different variable name and looping constructs. According to practitioner most of the existing tools produce high false positive results because of the flagging reusable components such as dlls, icons and other resources obtained from software libraries.

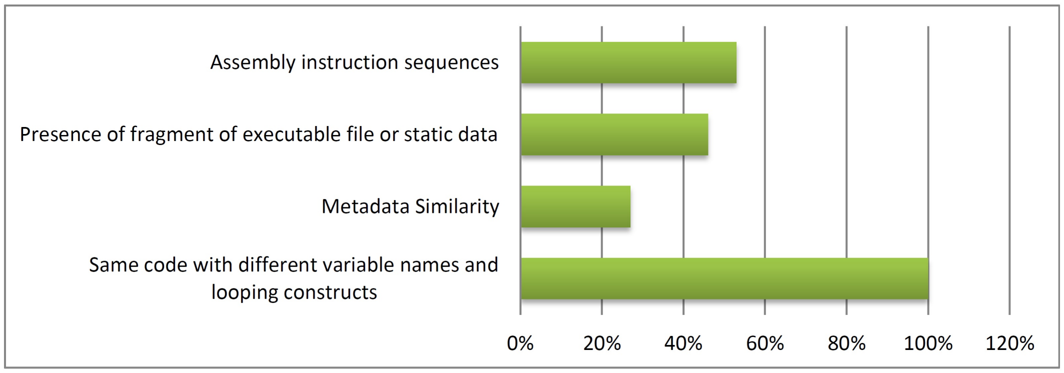


Figure 7.5: Executable Program File Similarity

7.1.3.7 File System Information

Participants were asked if the volume data volume obtained from the crime scene containing file structure information such as MFT(Master File Table) for windows files, inode, etc., has increased. From the result, we can say more than 68% of the cases file structure information is available. Hence this information can be used to improve the filtering abilities of the tools. We also found that amount of application-specific data has also increased, such as SQLite data on mobile devices. Hence the major challenge for approximate matching algorithms is to have the ability to find similarity across the device-specific data.

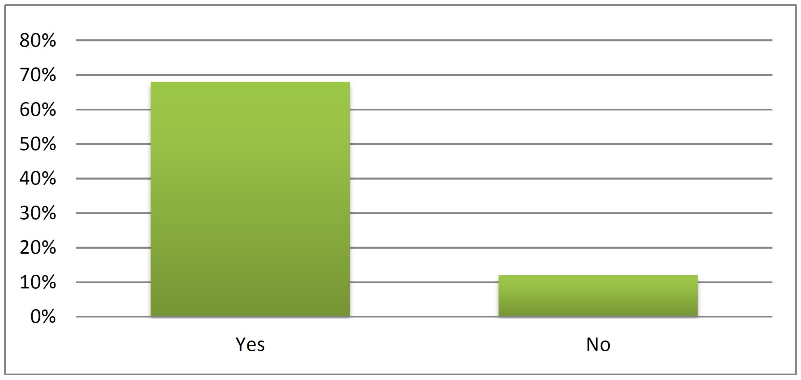


Figure 7.6: File System Information

7.2 Summary

Automated filtering has become one of the essential requirements of today's digital investigation process. Approximate matching algorithms are being used to perform the filtering. Approximate matching is a relatively new area of digital forensics, which is still evolving and needs to be defined more formally. The aim of the survey was to gain insights of the practitioners and researchers opinion regarding this new class of algorithms. From the results of the study, we found that practitioners and researchers acknowledge the utility of approximate matching algorithms and are willing to use it to speed up the investigation process. Since there is no standard way to measure the guarantees of the algorithms, it is difficult to find the appropriate tools.

Hence this chapter presents the practitioners and researcher's requirement and perspective towards approximate matching algorithms. Future work must focus on formalizing the properties of approximate matching algorithms and providing a well-defined evaluation framework to evaluate existing and future approximate matching schemes.

Chapter 8

Approximate Matching Evaluation Framework

Most of the prominent existing approximate matching schemes are bytewise matching techniques e.g. ssdeep [23], sdhash[13], mrsh[15], fbhash[18]. However, there are several scenarios where some of the existing algorithms do not compute the true similarity. In order to understand the potential of the existing or upcoming schemes, it is crucial to have an automated way to access the abilities of the proposed algorithms.

In this chapter, we present a general, platform-independent automated evaluation tool that tests the existing and upcoming approximate matching techniques and produces a comparative analysis on various parameters (such as false-negative rate, false-positive rate, precision, recall, f-score, MCC, etc.). To understand the capabilities of the algorithms in real-life scenario, we generate a real-world dataset with known similarity. Our tool allows the users to perform the test on the real-world dataset provided by the tool, and we also facilitate users a way to generate their own real dataset with known similarity to analyze the performance of approximate matching.

8.1 Use-Cases

To critique an algorithm meaningfully, it is crucial to understand the essential features and requirements. The baseline definition and terminology of the approximate matching algorithm are already defined by Breitinger et.al. [9]. However, the measure of accuracy of an algorithm is not formally defined.

The paper [9] defines the properties at a more general and broader level. However, similarity definition, as well as the requirements, varies for the different data object types. For example, files with similar perceived text content may have entirely different structure. They will be resulted entirely different if an inappropriate algorithm is applied; the actual similarities would remain unnoticed. A color and grayscale version of the same image would be completely different when using most of the existing schemes. Hence it is important to define the properties based on the practical requirements.

To understand the requirement, we conducted a survey in a closed group of highly experienced and trained digital investigators, practitioners, and researchers. The survey aimed to establish the key characteristics of approximate matching algorithm based on the requirement of digital forensics practitioners and researchers, who has an understanding of different perspectives towards these algorithms. The survey was conducted online among a closed group of 80 forensic practitioners, who are part of the NIST steering committee. The audience of the survey was kept limited in order to assure the reliability and legitimacy of the results. Details of the survey can be found 8. Based on all of the received responses approximate matching algorithms are being frequently used for the following four use cases:

- **Fragment Identification**: This use-case was proposed by[11]. Given a part of a file; the ability to correlate to the original file. For example, the fragment of a deleted malicious file can be found on the file slack or the unallocated space of the storage device. The ability to correlate a fragment to the source file is often important. It can also be useful for network traffic analysis.

- **Embedded Object Identification**: This characteristic was first proposed by[3]. Several document formats support linking and embedding of a document to the other objects. For example, pdf and word (docx, pptx) files can have images and videos embedded in the document. Embedded object identification refers to the ability to identify an object inside a document.

- **Related Document Detection**: This attribute facilitates the capability of identifying two objects that share a block or multiple blocks of data in common. For example, documents sharing the same header and footer or different versions of a document. This test case was first proposed by [3][11]

- **Identification of code version**: Today, we see software and operating system updates on a regular basis. The ability to filter out the different versions of the software and operating system files is often an essential requirement for an investigator. This requirement was specified by rossev[3].

Our evaluation tool provides a dedicated test case for each use-case and examines the algorithms on the real-world dataset.

8.2 Evaluation Framework

We present an automated evaluation framework/tool to test and compare the abilities of the approximate matching algorithms. The evaluation tool assesses the existing approximate matching algorithms for each essential property and characteristic mentioned in section 8.1. Our evaluation framework incorporates the two most prominent approximate matching techniques, ssdeep (version 2.14.1) and sdhash (version 3.4). We also present a way to integrate new techniques and algorithms into our framework by providing the executable of the new tool. The new algorithm can be integrated into the evaluation tool if it follows the following instructions/characteristics.

1. The tool should have following two features:
   - (i) Similarity digest generation : A function that takes a directory containing digital artifacts as input and generates the similarity digest or fingerprint.
   - (ii) Digest comparison : A comparison function that can take two similarity digest files as input and generates a similarity score of two fingerprints.
2. Similarity digest generation process should accept a folder as input and output a file containing digest of each file in the folder (separated by a new line). The digest generation command should be in following format/syntax:

   ToolName -d FolderPath -o digestfile

   Here FolderPath indicates the input folder and digestfile indicates the output file containing fingerprints.

3. The digest comparison can be performed in following format:
   ToolName −c digestfile1 digestfile2

   where digestfile1 and digestfile2 are the files containing digest of each file in folder1 and folder2 respectively (digest of each file is separated by a new line.)

4. Comparison function should compare each file digest of digestfile1 to each and every file digest of digestfile2.

5. The output of the digest comparison step should follow the following format:
   $File_i$ *Path* | $File_j$ *Path* | *SimilarityScore*
   where File$_i$ indicates a file from folder1, File$_j$ indicates a file from folder2 and SimilarityScore indicates the similarity score calculated by the tool.

The proposed evaluation tool comprises four test cases (i) Fragment Identification Test (ii) Embedded object Identification Test (iii) Related document Detection Test (iv) Code version Identification Test. Each test case consists of the following two-step:

- **Ground-Truth Dataset Generation**: One can not reliably estimate the error rates or correctness of the results obtained by an approximate matching algorithm without the knowledge of the ground truth' or the 'correct answer.' Hence it is crucial to have the ground-truth dataset with known similarity. In order to meaningfully test the boundaries of an algorithm in real-world scenarios, our tool provides a real-world (non-synthetic) ground-truth dataset for each test case. We also allow the user to generate their own ground-truth dataset with known-similarity, which can later be used to perform the evaluation. Before performing any test, users can generate their own dataset by providing files with which they want to generate the known-similarity dataset.

- **Evaluation**: For each test case, the evaluation can be done in two ways: Customized assessment and Comparative Assessment. *Customized Assessment* allows the user to perform the test for a particular scenario where the user can specify a particular approximate matching algorithm to test on a selected dataset and file type. The *Comparative Assessment* performs the test for both pre-incorporated approximate algorithms in the evaluation tool (i.e. ssdeep and sdhash) and generates a comparative analysis for the test case. The analysis is performed on the following parameters: Let there are two files f1, f2, and S(f1,f2) is the similarity score generated by an approximate matching algorithm. T represents a threshold value.

  1. **True Positive (TP)** indicates the number of results correctly predicted similar. A similarity outcome is considered True positive if f1 and f2 are similar and S(f1,f2) $\geq$ T.
  2. **False positive (FP)** indicates the number of results incorrectly predicted similar. A similarity result is considered True positive if f1 and f2 are not similar and S(f1,f2)$\geq$T.
  3. **True Negative (TN)** denotes the number of results correctly predicted non-similar. For example, f1 and f2 are not similar and S(f1,f2)$<$T.

4. **False Negative (FN)** indicates the number of results incorrectly predicted non-similar. for example f1 and f2 are similar and S(f1,f2)<T.
5. **True Positive rate (TPR)** represents the sensitivity of the algorithm. It is calculated as follows:

$$TPR = \frac{TP}{TP + FN}$$

6. **False Positive Rate (FPR)** is calculated as follows:

$$FPR = \frac{FP}{TN + FP}$$

7. **True Negative rate (TNR)** represents the specificity of the algorithm. It is calculated as follows:

$$TNR = \frac{TN}{TN + FP}$$

8. **False Negative Rate (FNR)** is calculated as follows:

$$FNR = \frac{FN}{TP + FN}$$

9. **Precision** indicates the relevance of results obtained by the approximate matching algorithm. This value can range between 0-1, where 1 indicates that all the files that resulted similar by the approximate matching algorithm are actually similar and 0 indicates that all files resulted as similar are un-related. Precision is calculated using the following formula:

$$Precision = \frac{TP}{TP + FP}$$

10. **Recall** represent the fraction of similar documents resulted similar by the approximate matching algorithm. This value ranges between 0-1, where 1 indicates that all the similar files were found similar by the algorithm and 0 indicates that none of the related or similar files resulted similar by the approximate matching scheme.

$$Recall = \frac{TP}{TP + FN}$$

11. **F-score** F-score is harmonic mean of precision and recall. It can be a more realistic measure of accuracy than the accuracy metric when false negatives & false positives are crucial, and there is an uneven class distribution. In the case of similarity hashing, the cost of false positives and false negatives are different. False-negative is much more expensive than false positive. In such cases, F1 is usually more useful than accuracy. Hence we calculate F-score to represent accuracy. This value also ranges from 0 to 1, where 0 indicates 0 accuracy and 1 indicates 100% accuracy.

$$F - score = \frac{2.precision.recall}{precision + recall}$$

12. **Matthews correlation coefficient (MCC)** : MCC measures the quality of the classification. MCC is calculated by using the following formula:

$$MCC = \frac{TPxTN - FPxFN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

8.2.1 Fragment Identification Test

The purpose of this test is to identify an approximate matching tool's ability to correlate a fragment of a file to its source file. One of the real-life use case of this test could be to find the fragments of a deleted and partially overwritten data (evidence) on the suspect's hard-disc. The test is performed in the following two phases: Dataset Generation and Evaluation.

8.2.1.1 Fragment Dataset Generation

The current implementation of our evaluation tool supports two file types i.e., text and docx for dataset generation and evaluation. To generate the data set, the user needs to input the file (of selected type) that will be used to generate the fragment dataset. Our tool generates 23 different fragments of each input file. Where the fragment files are 95%, 90%, 85%, 80%, 75%, 70%, 65%, 60%, 55%, 50%, 45%, 40%, 35%, 30%, 25%, 20%, 15%, 10%, 5%, 4%, 3%, 2%, 1% fragment of the input file. We generate two types of fragment datasets: Sequential Fragment Dataset and random Fragment dataset. Let the size of the file is S.

- **Sequential Fragment** : The x% sequential fragment is created by deleting first (S-x)% of the file. For example, if a file is 100 bytes long and we want to generate a 40% fragment. To generate a 40% sequential fragment first 60 bytes (100-40% of 100) are deleted in the fragment file.
- **Random Fragment**: The x% random fragment is generated by deleting the (S-x)% of data from a randomly chosen position of the file. For example, if a file is 100 bytes long. To generate a random fragment, let the random position

is 71 then the last 30 and first 30 bytes of the file will be removed, and the remaining will be 40% random fragment of the input file.

The name of the resulted fragment signifies the size of the fragment and its source file. Fragment files are named in the following format: sourceFile_i, where 0≤i≤23; 0 indicates the 95% and 23 denotes the 1% fragment of the sourceFile. Our tool also provides a pre-generated real-world sequential and random fragment dataset with the evaluation tool.

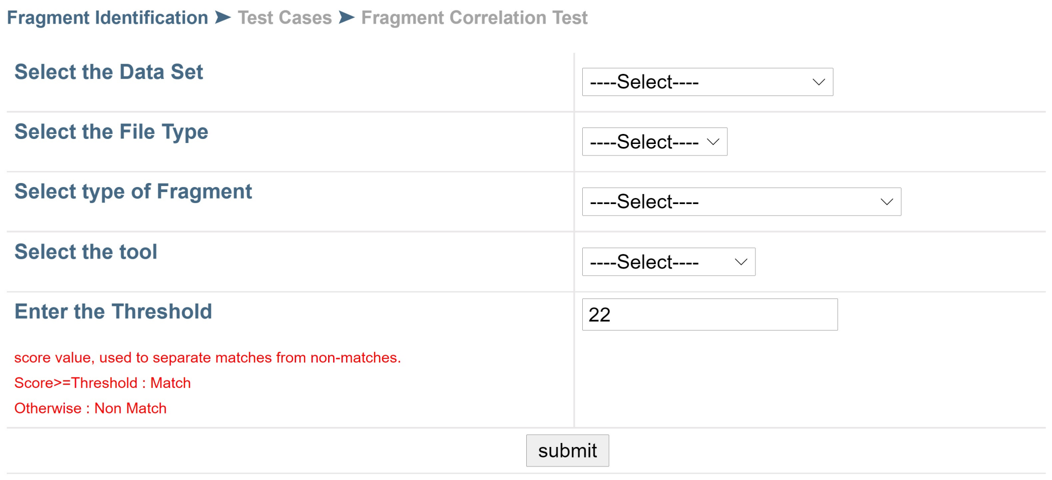
8.2.1.2 Evaluation

We performed the following two tests to identify the fragment identification ability of the approximate matching algorithms.

1. **Fragment Correlation Test**: The test aims to explore the tool's ability to find similarity between various fragments of a file to its source file. Approximate Matching tools can be examined in two ways for this test case:

   (i) ***Customized Assessment:*** This approach allows the user to performs tests by selecting the options shown in figure 8.1. The figure shows a snippet of the evaluation framework for fragment correlation test for this approach. To perform the test user needs to specify the following:

   - The **dataset**, This option allows the user to choose if he wants to perform the test on his own generated dataset or on the pre-generated dataset provided by the tool.


Fragment Identification ➤ Test Cases ➤ Fragment Correlation Test

Select the Data Set	----Select----
Select the File Type	----Select----
Select type of Fragment	----Select----
Select the tool	----Select----
Enter the Threshold score value, used to separate matches from non-matches. Score>=Threshold : Match Otherwise : Non Match	22

Figure 8.1: Customized assessment

- The **file type** on which the test will be performed. For this test case, there are two file types text and docx.
- The **fragment type** allows performing the test on sequential or random fragment dataset.
- The **approximate matching scheme** that one wants to evaluate. The current implementation of the tool facilitates performing the test on ssdeep and sdhash. However, the user can include other approximate matching tools as well using the '*add a new tool*' option and following the guidelines mentioned in 8.2.
- The **threshold**, This is the similarity score value used to separate matches from non-matches. If the similarity score generated by the approximate matching tool is greater than or equal to this value, it is considered as a positive or correct result other-

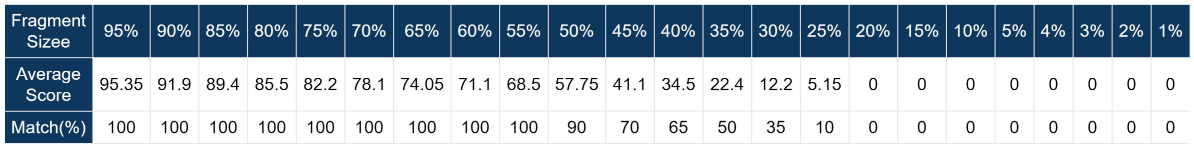
wise negative. Users can specify any value between 0 to 100. The default value is 22.

Figure 8.2 shows results generated by the evaluation tool after providing a value for each option. It lists the name of the fragment file, source file, similarity score generated by the selected approximate matching technique, and the actual similarity for each comparison. We present the True Positive, True Negative, False Positive, False Negative, False positive rate (FPR), False negative rate (FNR), Precision, Recall, F-score, MCC measures based on the similarity score obtained by the selected approximate matching scheme. Figure 8.2 shows the results obtained of comparative assessment of ssdeep tool on sequential fragment dataset.

(ii) ***Comparative Assessment:*** This approach performs the fragment identification test on both pre-incorporated algorithms (ssdeep and sdhash), for all file types, and fragment type datasets. This assessment generates a report that presents a comparative analysis of both algorithms in all scenarios using measures such as false positive, false negative, precision, recall, f-score, and MCC. The tool allows the user to provide the threshold for the results generated for this test. Figure 8.3 shows the report generated by the evaluation tool for this test with default threshold value i.e., 22. The report is shown in the figure 8.3 In this test as well user gets to choose the threshold; by default, it's 22. As we can see in the figure, the results obtained by this test varies for the different file type (text and docx). In most cases, sdhash outperforms ssdeep for this test case.

2. **Smallest Fragment Identification Test**: Test aims to find the smallest fragment size an approximate matching tool can correlate or identify. This test also can be performed in two ways: Customized assessment and Comparative Assessment.

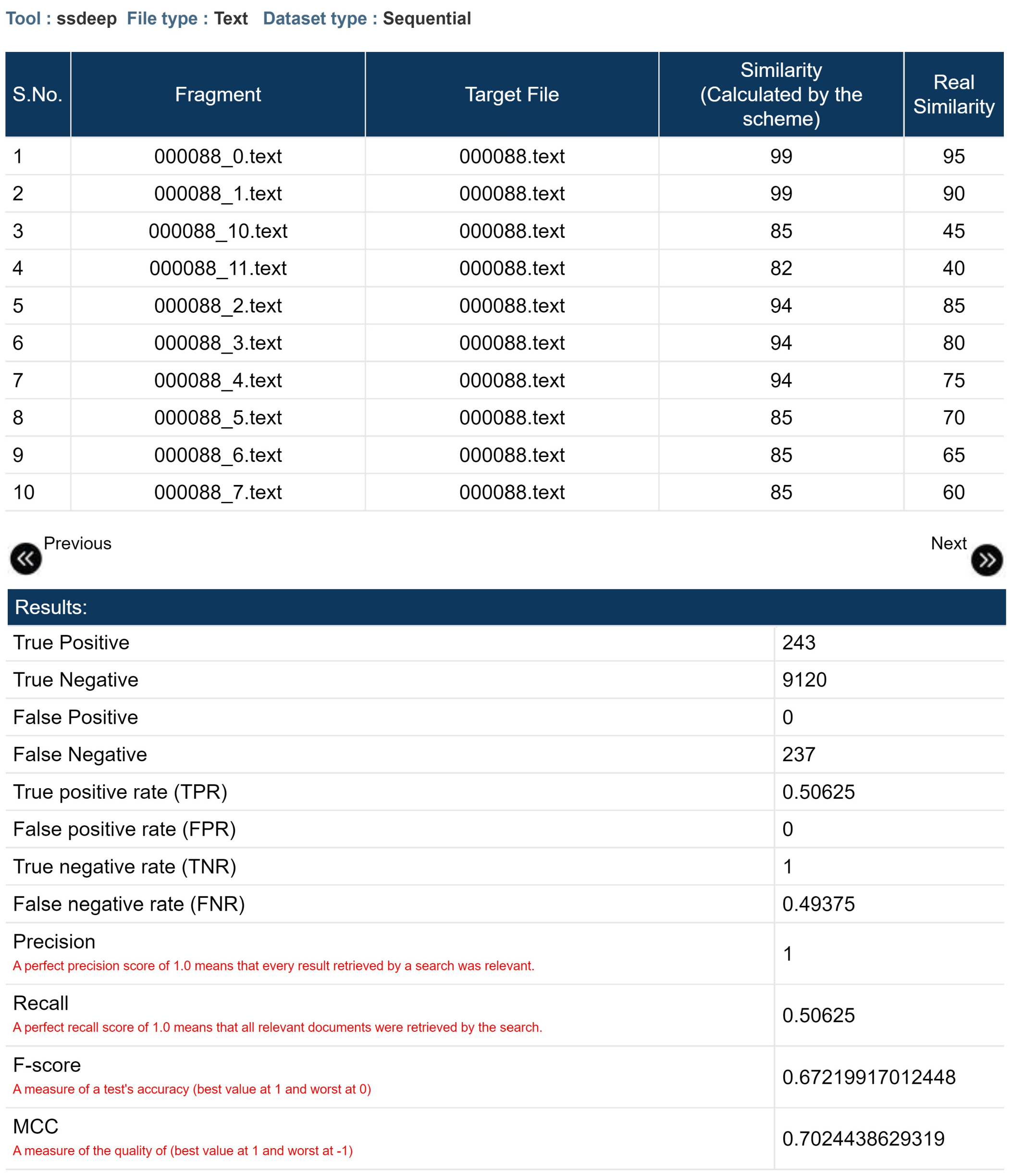
   (i) ***Customized Assessment:*** Figure 8.4 shows the result of the customized test on ssdeep on sequential fragments of the text file dataset with threshold 22. The first row represents the different fragment sizes examined during the test. The second row represents the average score, which is the mean of similarity scores generated by ssdeep for each fragment size on various test files. The third row represents the match percentage or correlation detection rate which indicates the percentage of the test samples where the ssdeep was able to detect similarity by producing a match score higher or equal to the threshold.



Fragment Sizee	95%	90%	85%	80%	75%	70%	65%	60%	55%	50%	45%	40%	35%	30%	25%	20%	15%	10%	5%	4%	3%	2%	1%
Average Score	95.35	91.9	89.4	85.5	82.2	78.1	74.05	71.1	68.5	57.75	41.1	34.5	22.4	12.2	5.15	0	0	0	0	0	0	0	0
Match(%)	100	100	100	100	100	100	100	100	100	90	70	65	50	35	10	0	0	0	0	0	0	0	0

Figure 8.4: Results of customized assessment of ssdeep on text sequential fragment dataset for Smallest fragment identification test

   (ii) ***Comparative Assessment:*** Figure 8.5 shows a part of the comparative assessment report generated for the smallest fragment identification test. The complete report can be generated using the evaluation tool; we



Tool : ssdeep **File type :** Text **Dataset type :** Sequential

S.No.	Fragment	Target File	Similarity (Calculated by the scheme)	Real Similarity
1	000088_0.text	000088.text	99	95
2	000088_1.text	000088.text	99	90
3	000088_10.text	000088.text	85	45
4	000088_11.text	000088.text	82	40
5	000088_2.text	000088.text	94	85
6	000088_3.text	000088.text	94	80
7	000088_4.text	000088.text	94	75
8	000088_5.text	000088.text	85	70
9	000088_6.text	000088.text	85	65
10	000088_7.text	000088.text	85	60

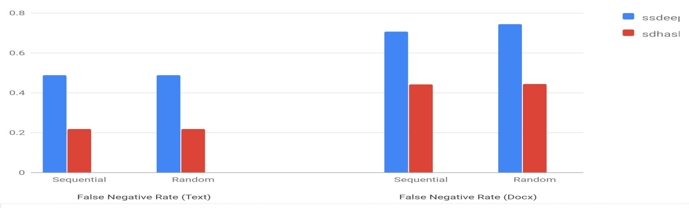
Results:	
True Positive	243
True Negative	9120
False Positive	0
False Negative	237
True positive rate (TPR)	0.50625
False positive rate (FPR)	0
True negative rate (TNR)	1
False negative rate (FNR)	0.49375
Precision A perfect precision score of 1.0 means that every result retrieved by a search was relevant.	1
Recall A perfect recall score of 1.0 means that all relevant documents were retrieved by the search.	0.50625
F-score A measure of a test's accuracy (best value at 1 and worst at 0)	0.67219917012448
MCC A measure of the quality of (best value at 1 and worst at -1)	0.7024438629319

Figure 8.2: Customized assessment of ssdeep on text sequential fragment dataset

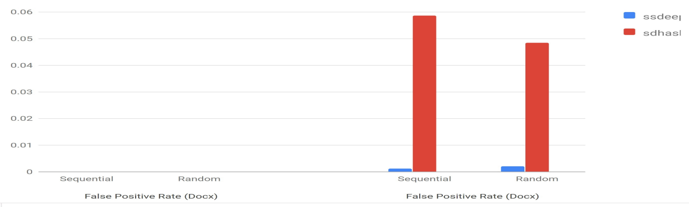
Fragment Identification ➤ Test Cases ➤ Comparative Assessment

Fragment Correlation Test

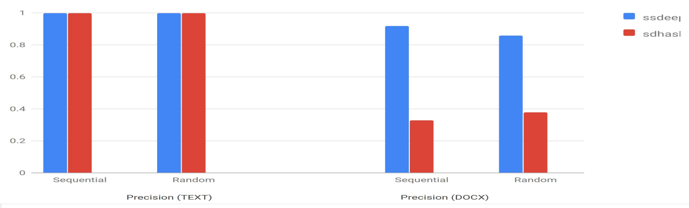
1.	This report presents the results of the "Fragment Detection Test". This test identifies the ability of existing schemes to identify the fragment of a file.
2.	The Test is performed on the data-set of (i) 960 fragment files (480 sequential fragment and 480 randomly generated fragment), generated by 20 Text files. Each text file generates 24 sequencial fragment file and 24 random fragment files of following sizes: 95%, 95%, 85%, 80%, 75%, 70%, 65%, 60%, 55%, 50%, 45%, 40%, 35%, 30%, 25%, 20%, 15%, 10%, 5%, 4%, 3%, 2%, 1%, <1%. (ii) 960 fragment files (480 sequential fragment and 480 randomly generated fragment), generated by 20 Docx files. Each docx file generates 24 sequencial fragment file and 24 random fragment files of following sizes: 95%, 90%, 85%, 80%, 75%, 70%, 65%, 60%, 55%, 50%, 45%, 40%, 35%, 30%, 25%, 20%, 15%, 10%, 5%, 4%, 3%, 2%, 1%, <1%.
3.	(i) Total number of comparision performed by each scheme for text files: 19200. (ii) Total number of comparision performed by each scheme for docx files: 19200.
4.	Bar chart shown belows represents the **False Negative Rate** of ssdeep and sdsash for text and docx files resepectively. The results obtained from ssdeep has more False Netaive than sdhash in case of both textand docx files.



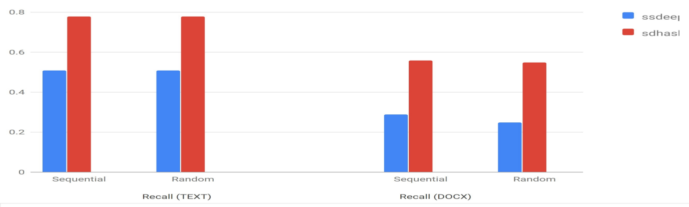
5. Bar graph shown belows represents the **False Positive Rate** of ssdeep and sdsash for text and docx files resepectively. Both ssdeep and sdhash no False Positive for text data-set however in case of docx files sdhash has more False Negative than ssdeep.



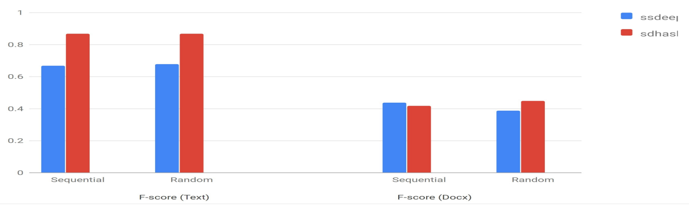
6. Results obtained by ssdeep and sdhash both have same prcision in case of text data-set.



7. Obtained results show that sdhash has better recall rates than ssdeep on an avg.



8. **F-score** represents test's accuracy, from the results we can conclude that sdhash provides more accurate results than ssdeep.



9. Figure below shows the **Matthews correlation coefficient (MCC)** comparision of both ssdeep and sdhash.

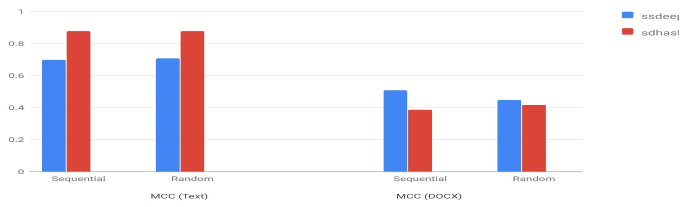


Figure 8.3: Comparative assessment report of fragment correlation test

do not include the complete report here to conserve space. In figure 8.5 the x-axis of the graph represents the fragment size. The bars in the graph represent the similarity score, and lines represent the match percentage or correlation detection rate. Match Percentage is the percentage of those test samples where the tools are able to detect similarity by giving a valid match score. In the figure 8.5 the blue lines and bar indicate results obtained by ssdeep, and the red line and bar indicate the results generated by sdhash. The graph is generated for default threshold 22.

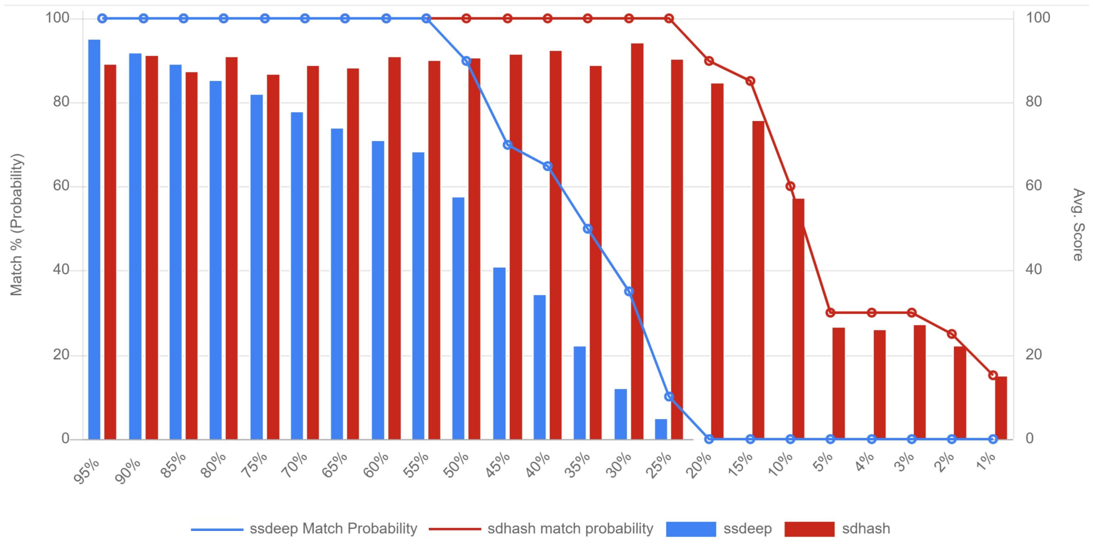


Figure 8.5: Comparative assessment result for smallest fragment identification test

The graph shows the avg. similarity score and match percentage of ssdeep and sdhash on Text dataset with sequential fragments. The results show that ssdeep can correlate a fragment to its source file if it's 50% or more of the source file but can't identify similarity for 20% and below fragment size. However, sdhash can detect the similarity with high probability up to fragment size 15%. Even sdhash can correlate 5% fragment to its original file up to 30% times. Similarity score in case of sdhash up-to fragment size 45% is reasonably accurate, but it can not correlate fragments of size 25% or smaller. We can state, sdhash performs better in this case.

8.2.2 Embedded Object Identification Test

The purpose of this test is to identify the ability of an approximate matching algorithm to detect an object within a file. The object refers to a separate entity that is attached or embedded to a file, for example, an image, video, or audio clip in a word file.

8.2.2.1 Enmbedded Object Dataset Generation

The current implementation supports two file types, i.e., docx and PPTX, and four types of embedded objects i. e. JPEG, BMP, TIFF, and GIF. The user needs to provide the files and embedded objects of the selected type to generate the embedded data set. The dataset is generated by embedding each provided embedded object in each provided file in a randomly chosen location. For example, if 5 docx file and 5 jpeg file has been provided by the user, then the tool will generate 25 docx file with embedded jpeg images. The resulted embedded files are named as follows objectname_filename.fileExtension (e.g. image1_doc4.docx) so that by looking at the name of the file, we know which object is embedded in the file.

8.2.2.2 Evaluation

Two different tests have been used to perform the evaluation.

- **Embedded Object Detection**: This test aims to test the ability of the approximate matching algorithm to identify the presence of an object in a document. This test compares the objects to the embedded files and calculates the similarity with approximate matching algorithms. This test addresses the scenarios where an investigator needs to search for a malicious object (e.g., image) into the suspected data or files obtained from an incidence.

  (i) ***Customized Assessment*** Similar to the fragment Identification test, the user needs to specify a file type, object type, and the approximate matching algorithm, which they want to test. Users can also specify a threshold value which is by default considered 22. The test compares each embedded document of the specified type and each the object of the specified type in the dataset with each other and computes the true positive, true negative, False positive, False negative, False positive rate, false-negative rate, Precision, Recall, F-score, MCC values. Figure 8.6 shows the result of the customized test on ssdeep with docx file type and jpeg object type with 22 as the threshold value.

  (ii) ***Comparative Assessment*** The test performs comparative analysis on various parameter i.e., precision, recall, F-score, MCC score and generates a comparative report which shows that which tool performs better in which scenario. Figure 8.7 shows a snippet of the generated report that shows the comparison of ssdeep and sdhash's f-score on docx dataset. As we can see, the results obtained by ssdeep are more accurate in the case of jpeg and gif embedded objects. However, sdhash performs better in the case of bmp and tiff.

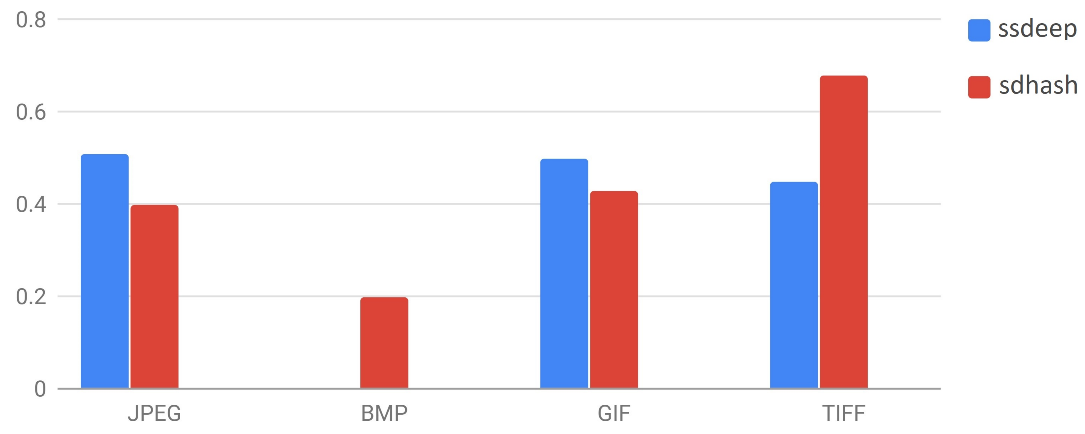
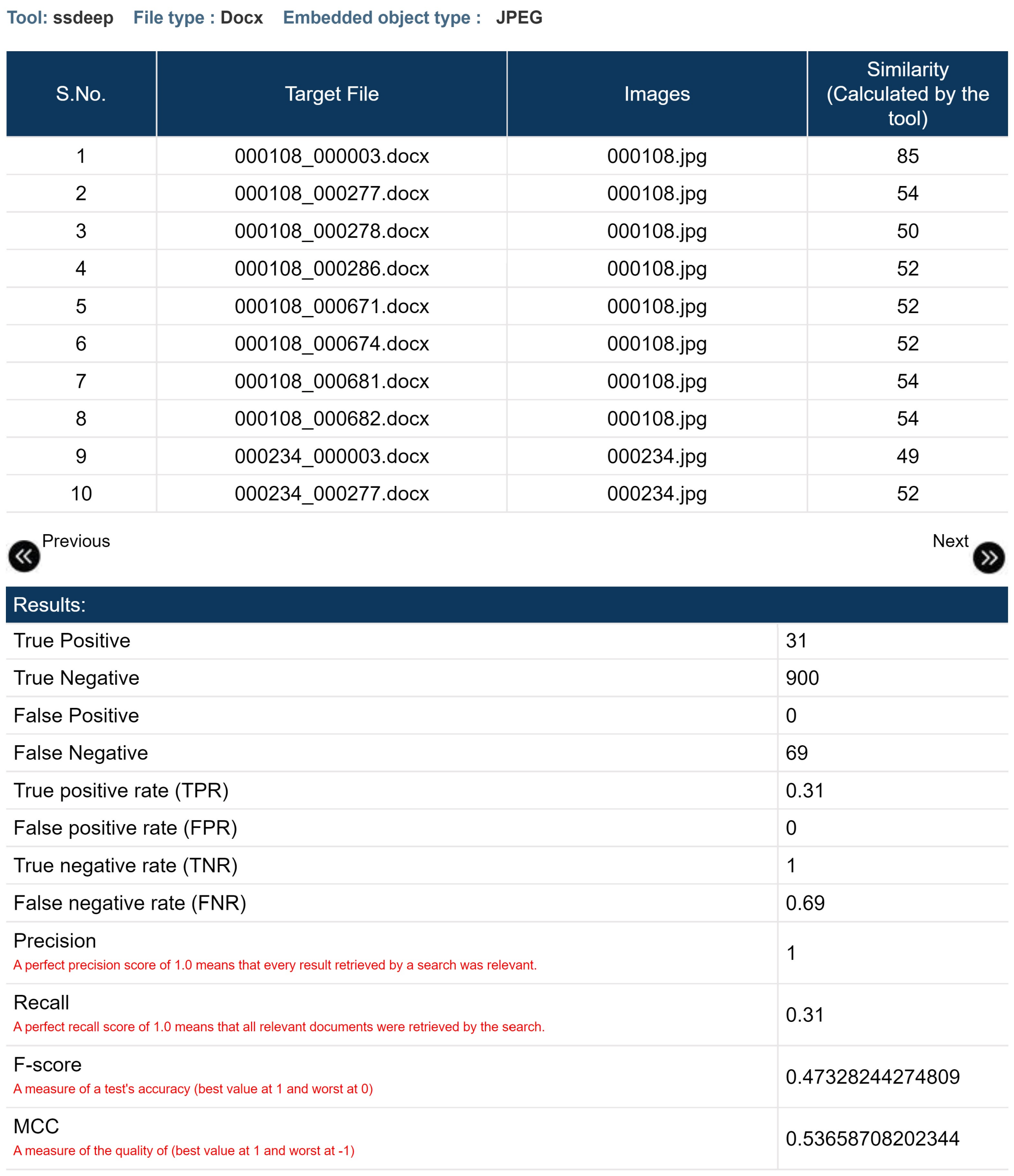


Figure 8.7: F-score generated on Docx dataset by embedded object identification test

- **Common Object Identification**: This test aims to discover and verify the capability of an approximate matching algorithm to correlate the files containing a common object.

  (i) ***Customized Assessment*** Similar to previous tests, the user needs to select the algorithm, file type, object type and specify a threshold value. The test compares each embedded documents to each other computes the True positive, true negative, False positive, False negative, False positive rate, false-negative rate, Precision, Recall, and F-score values. The figure 8.8 shows the



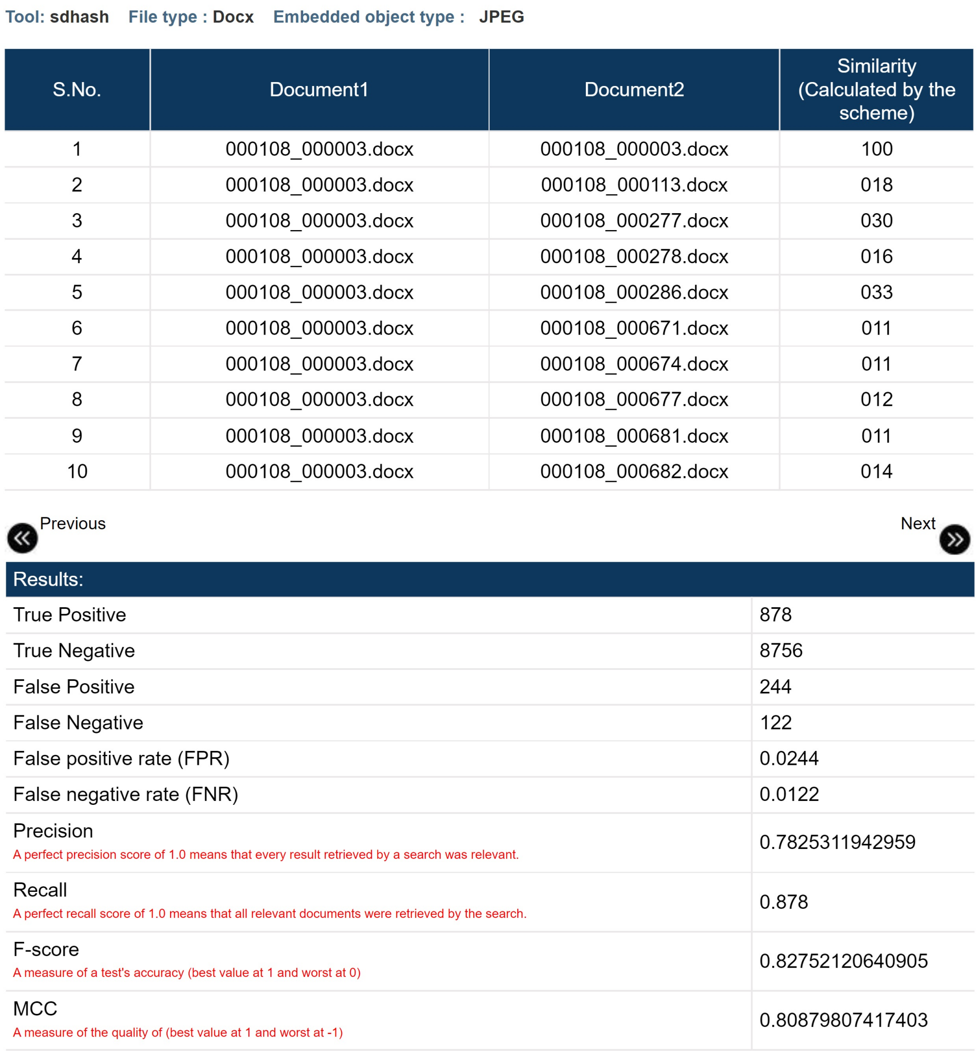
Tool: ssdeep **File type :** Docx **Embedded object type :** JPEG

S.No.	Target File	Images	Similarity (Calculated by the tool)
1	000108_000003.docx	000108.jpg	85
2	000108_000277.docx	000108.jpg	54
3	000108_000278.docx	000108.jpg	50
4	000108_000286.docx	000108.jpg	52
5	000108_000671.docx	000108.jpg	52
6	000108_000674.docx	000108.jpg	52
7	000108_000681.docx	000108.jpg	54
8	000108_000682.docx	000108.jpg	54
9	000234_000003.docx	000234.jpg	49
10	000234_000277.docx	000234.jpg	52

Results:	
True Positive	31
True Negative	900
False Positive	0
False Negative	69
True positive rate (TPR)	0.31
False positive rate (FPR)	0
True negative rate (TNR)	1
False negative rate (FNR)	0.69
Precision A perfect precision score of 1.0 means that every result retrieved by a search was relevant.	1
Recall A perfect recall score of 1.0 means that all relevant documents were retrieved by the search.	0.31
F-score A measure of a test's accuracy (best value at 1 and worst at 0)	0.47328244274809
MCC A measure of the quality of (best value at 1 and worst at -1)	0.53658708202344

Figure 8.6: ssdeep results on Docx embedded file and jpeg object

result of customized test results generated by sdhash on docx file type, jpeg object type, and threshold 22.



Tool: **sdhash** **File type :** **Docx** **Embedded object type :** **JPEG**

S.No.	Document1	Document2	Similarity (Calculated by the scheme)
1	000108_000003.docx	000108_000003.docx	100
2	000108_000003.docx	000108_000113.docx	018
3	000108_000003.docx	000108_000277.docx	030
4	000108_000003.docx	000108_000278.docx	016
5	000108_000003.docx	000108_000286.docx	033
6	000108_000003.docx	000108_000671.docx	011
7	000108_000003.docx	000108_000674.docx	011
8	000108_000003.docx	000108_000677.docx	012
9	000108_000003.docx	000108_000681.docx	011
10	000108_000003.docx	000108_000682.docx	014

Previous Next

Results:	
True Positive	878
True Negative	8756
False Positive	244
False Negative	122
False positive rate (FPR)	0.0244
False negative rate (FNR)	0.0122
Precision A perfect precision score of 1.0 means that every result retrieved by a search was relevant.	0.7825311942959
Recall A perfect recall score of 1.0 means that all relevant documents were retrieved by the search.	0.878
F-score A measure of a test's accuracy (best value at 1 and worst at 0)	0.82752120640905
MCC A measure of the quality of (best value at 1 and worst at -1)	0.80879807417403

Figure 8.8: sdhash results on common object identification test

(ii) ***Comparative Assessment*** Similar to the previous test, this test performs comparative analysis on various parameters, i.e., precision, recall, F-score, MCC score, and generates a comparative analysis report comparing performs of both algorithms. Figure 8.9 shows a snippet of the generated report by this, which shows the comparison of ssdeep and sdhash's F-score on the pptx file. Figure 8.9 shows that the results obtained by the sdhash are more accurate in comparison to ssdeep for jpeg, bmp, gif, and tiff.

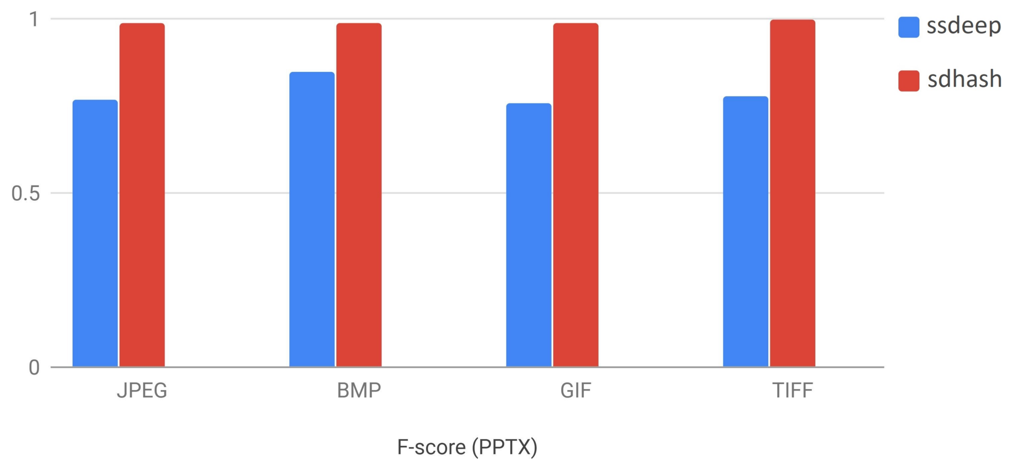


Figure 8.9: F-score comparison result generated by common object identification test

8.2.3 Related Document Detection Test

This test aims to discover and verify the approximate matching algorithm's ability to identify the related document (i.e., those share common data) or a different version of a document. To examine this characteristic, the evaluation tool provides two types of tests: Single-common-block file correlation Test and Multiple-common-blocks file correlation Test.

- **Single-common-block Test**: This test focuses on the scenario where digital objects or files share one block of consecutive data.

- **Multiple-common-block Test**: This test simulates a scenario where the documents share multiple common blocks of common data with each other.

8.2.3.1 Related Document Dataset Generation

- **Single-common-block dataset** The dataset is generated as follows:

  1. User needs to input three or multiple of three files to generate the dataset.
  2. We first resize all the files to the same size.
  3. The following 10 different sized fragments of the first file is created: 100%, 66.66%, 42.86%, 25%, 11.11%, 5.2%, 4.1%, 3.09%,2.04%, 1.01%.
  4. Each fragment is inserted in randomly chosen positions in the second and third files one by one. This will result 10 pairs of the second and third file with shared common block of 50%, 40%, 30%, 20%, 10%, 5%, 4%, 3%, 2%, 1% respectively.
  5. Take the next triplet of files and repeat from step 1 to step 3 for all input files.

For example, if f1, f2, and f3, three files were provided by the user. This process will output the two set of 10 files and the file are named as follows: f1_f2_i and f1_f3_i where i $\in$ {1,2,3,4,5,10,20,30,40,50}. Two file f1_f2_k and f1_f3_k represents the k% common data between the files.

In this way, we have generated the dataset of 160 text file pairs with a shared single common block and 160 docx file pairs using the T5 corpus [64]. This dataset is included in the tools as *existing dataset*.

- **Multiple-common-blocks dataset** The dataset is generated using following steps:

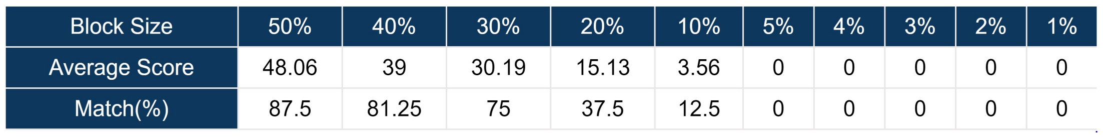
  1. User needs to input three or multiple of three files to generate the dataset.
  2. We first resize all the files to the same size.
  3. The following 10 different sized fragments of the first file is created: 100%, 66.66%, 42.86%, 25%, 11.11%, 5.2%, 4.1%, 3.09%,2.04%, 1.01%.
  4. Each fragment will be split into x pieces (where x $\in$ {2,4,8,16,32,64}) one by one, and each piece will be inserted into the second and third file in the randomly chosen positions. This will result 50 pairs of files with shared common blocks, where the first 10 files share 50% commonality, the second set 20%, and so on.
  5. Take another triplet of files and repeat from step 1 to step 3 for all input files.

  For example if f1, f2 and f3 three files were provided by the user. This process will output the two set of 50 files and the file are named as follows: f1_f2_i_j and f1_f3_i_j where i $\in$ {1,2,3,4,5,10,20,30,40,50} and j $\in$ {2,3,4,8,16,32}. Two file f1_f2_k_l and f1_f3_k_l represents the k% commonality distributed into l blocks between the files. In this way, we have generated the dataset of 800 text file pairs with a shared single common block and 800 docx file pairs using the T5 corpus [64]. This dataset is included in the tools as *existing dataset*.

8.2.3.2 Evaluation

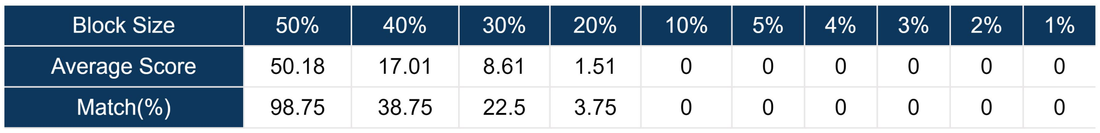
Both single and multiple common blocks tests are examined in the same manner for customized and comparative assessment.

(i) ***Customized Assessment*** Similar to previous tests, the users have the option to select the algorithm and file type that they want to investigate and can specify the threshold value. The test compares related documents with each other and computes the similarity score between the files sharing common blocks of various sizes. The figure 8.10, 8.11 show the result of customized test results generated obtained by ssdeep on single and multiple common block text datasets, respectively. In the figure, the first row indicates the various block sizes. The second row represents the average similarity score generated by the ssdeep for the common block size files. The third row denotes the Match Percentage (also called correlation detection rate). Match Percentage represents the percentage of the test sample where the tool can detect similarity between the related documents.



Block Size	50%	40%	30%	20%	10%	5%	4%	3%	2%	1%
Average Score	48.06	39	30.19	15.13	3.56	0	0	0	0	0
Match(%)	87.5	81.25	75	37.5	12.5	0	0	0	0	0

Figure 8.10: Single common block test results generated by ssdeep on text dataset



Block Size	50%	40%	30%	20%	10%	5%	4%	3%	2%	1%
Average Score	50.18	17.01	8.61	1.51	0	0	0	0	0	0
Match(%)	98.75	38.75	22.5	3.75	0	0	0	0	0	0

Figure 8.11: Multiple common block test results generated by ssdeep on text dataset

(ii) ***Comparative Assessment*** The comparative assessment test is conducted in the same fashion for both single and multiple common block tests. The comparative assessment draws an analogy between ssdeep and sdhash for the various dataset and file types. Figure 8.12 and 8.13 presents the results generated by the comparative assessment of the single common block file correlation test on the text and docx dataset. The complete report can be generated using the tool.

The graph in Fig. 8.12 shows that in the case of single-common-block test, `ssdeep` (represents as blue) can detect the similarity between two related documents only if the commonality between them is 30% or more of the document size with a sufficiently high match percentage, i.e., 75. It can not detect commonality below 10%. Even in the case of 50% similarity, it does not find the correlation with probability 1. On the other hand, sdhash performs comparatively better. It detects similarity with 100 match percentages for 50% or more single block commonality, and the similarity score generated by the tools are very close to the actual similarity. However, sdhash also does not reliably correlate the files that share less than 30% data.

Fig. 8.13 shows the average similarity score and match percentage of `ssdeep`, `sdhash` on Docx dataset. The correlation detection rate or the match percentage of `ssdeep` and `sdhash` is not consistent. In terms of average matching score, it can

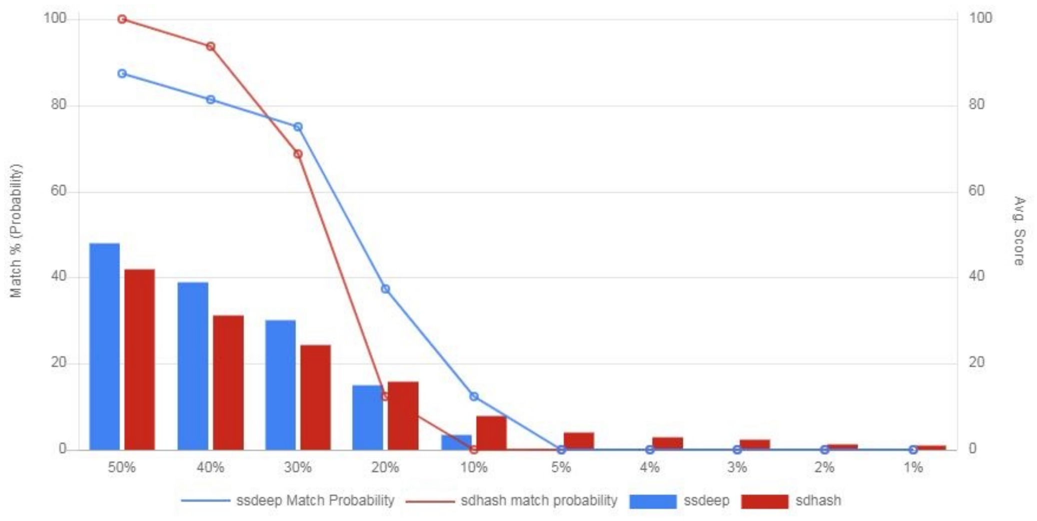


Figure 8.12: Single-common-block test results for Text-Data Set

be seen that results obtained by `ssdeep` and `sdhash` are imprecise and do not reflect the actual similarity since docx is a compressed file structure and both the schemes work on byte level.

8.2.4 Identification of code version

Nowadays, we see a constant stream of software and operating system updates on our devices. Most of the operating system files and proprietary software are legitimate and need not be examined by the forensic investigator during an investigation. Such files are called know-to-be-good files. These files can be filtered out from further examination. However, storing the cryptographic hash of each version (which is increasing day by day) of such files and comparing all of them for each examination could be infeasible. So this is one of the most important attributes of an approximate matching algorithm to be able to

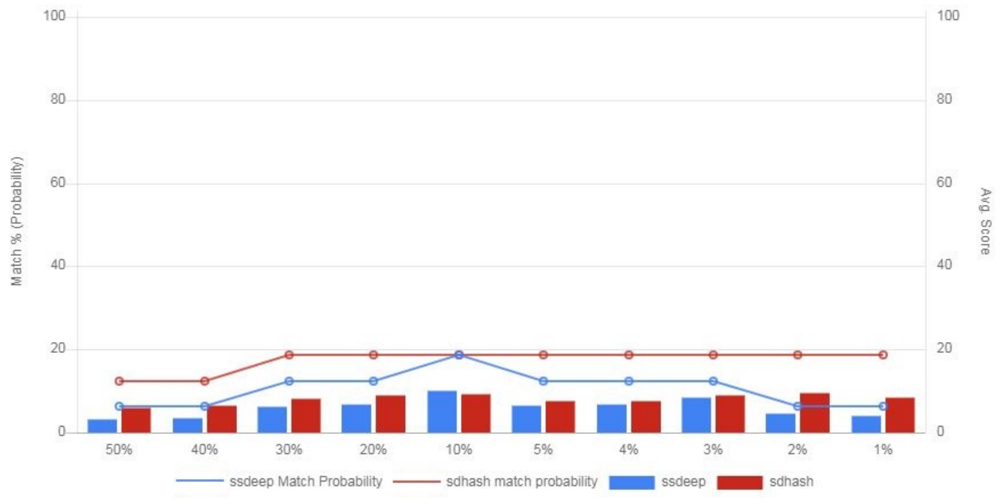


Figure 8.13: Single-common-block test results for Docx dataset

filter-out, patched and upgraded software versions. This test aims to identify the approximate matching tool's ability to find similarity between the various version of software or program files.

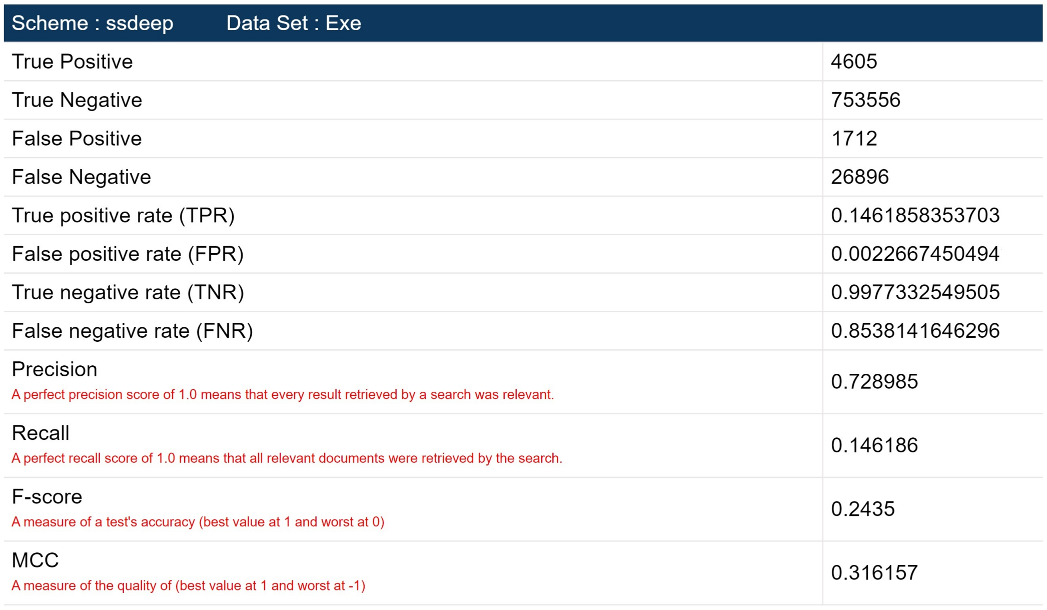
8.2.4.1 Dataset

For this test case, we do not generate our dataset since the real dataset is already produced by the NSRL [7] and is publicly available. We use the NSRL dataset of windows program files. We have used 887 executable files and 346 DLL (dynamic-link library) files containing different versions of 10 generations of Windows operating system releases. We downloaded the NIST dataset containing images of EXE and DLL files. In the dataset, the files are named as their SHA-1 hash digest, which we changed to the name of the file preceded by a unique id provided in the NSRL data record to each program file and an underscore. For example, 564930796_winver.exe here 564930796 represents the unique record id, and winver.exe identifies the name of the operating system file winver.exe.

8.2.4.2 Evaluation and Results

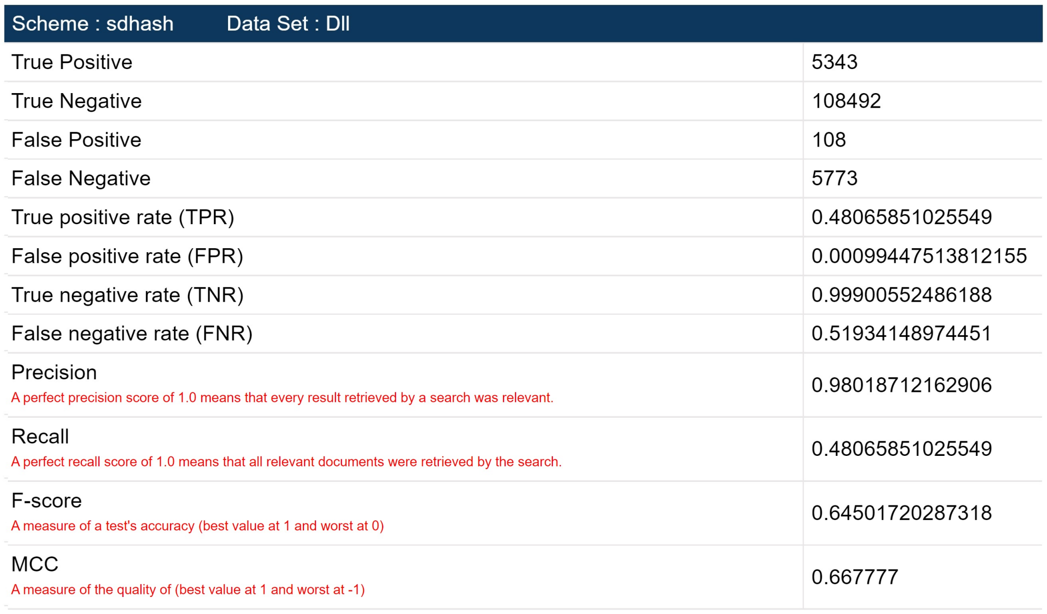
Similar to the previous three tests, we can perform this test also in both customized and comparative fashion.

(i) ***Customized Assessment:*** In order to perform this test, we can specify the dataset we want to test, i.e., EXE and DLL, and the tool we want to examine for the code identification test. We can also specify the threshold value for the similarity score generated by the tool. This will enable us to check program version identification for the ability of any tool on the desired dataset. Figure 8.14 represents the results obtain by the ssdeep tool for exe files, and figure 8.15 shows the results of the sdhash tool on the dll dataset.



Scheme : ssdeep Data Set : Exe	
True Positive	4605
True Negative	753556
False Positive	1712
False Negative	26896
True positive rate (TPR)	0.1461858353703
False positive rate (FPR)	0.0022667450494
True negative rate (TNR)	0.9977332549505
False negative rate (FNR)	0.8538141646296
Precision A perfect precision score of 1.0 means that every result retrieved by a search was relevant.	0.728985
Recall A perfect recall score of 1.0 means that all relevant documents were retrieved by the search.	0.146186
F-score A measure of a test's accuracy (best value at 1 and worst at 0)	0.2435
MCC A measure of the quality of (best value at 1 and worst at -1)	0.316157

Figure 8.14: Code version identification test results of ssdeep for exe files



Scheme : sdhash Data Set : Dll	
True Positive	5343
True Negative	108492
False Positive	108
False Negative	5773
True positive rate (TPR)	0.48065851025549
False positive rate (FPR)	0.00099447513812155
True negative rate (TNR)	0.99900552486188
False negative rate (FNR)	0.51934148974451
Precision A perfect precision score of 1.0 means that every result retrieved by a search was relevant.	0.98018712162906
Recall A perfect recall score of 1.0 means that all relevant documents were retrieved by the search.	0.48065851025549
F-score A measure of a test's accuracy (best value at 1 and worst at 0)	0.64501720287318
MCC A measure of the quality of (best value at 1 and worst at -1)	0.667777

Figure 8.15: Code version identification test results of sdhash for dll files

(ii) ***Comparative Assessment:*** This assessment compares ssdeep and sdhash for all important parameters, i.e., false-positive rate, false-negative rates, precision, recall, f-score, and MCC. Figure 8.16 and 8.17 shows the recall rate and F-score comparison of ssdeep and sdhash. The complete report can be generated using the tool. We can see that both algorithms do not perform well in terms of recall rate and f-score. For exe sdhash performs slightly better, but the recall rate is below 0.21 and F-score below 0.32, which is low. A similar outcome can be seen for DLL files as well. Based on these two graphs and other measures in the report, we can state that none of the two algorithms reliably detect similarity for this test case.

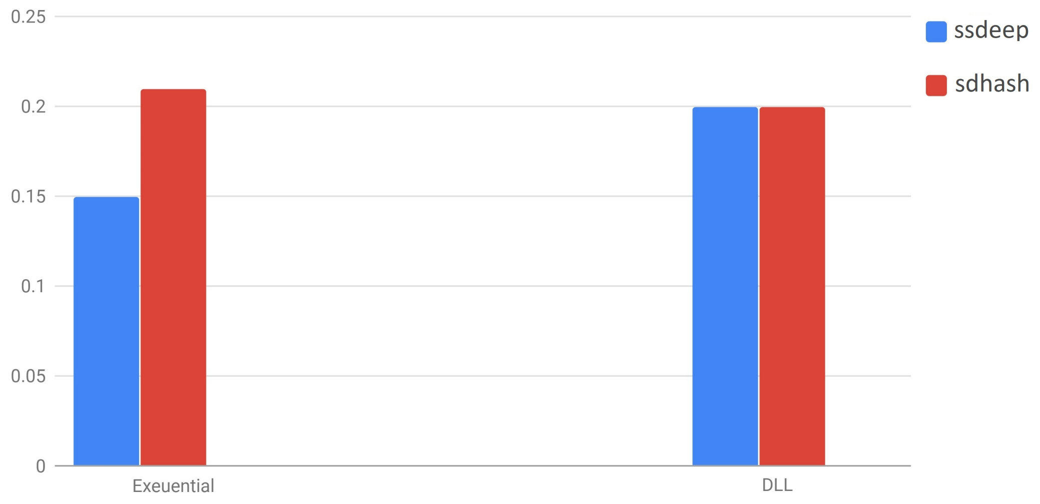


Figure 8.16: Recall rates comparison of code version identification test

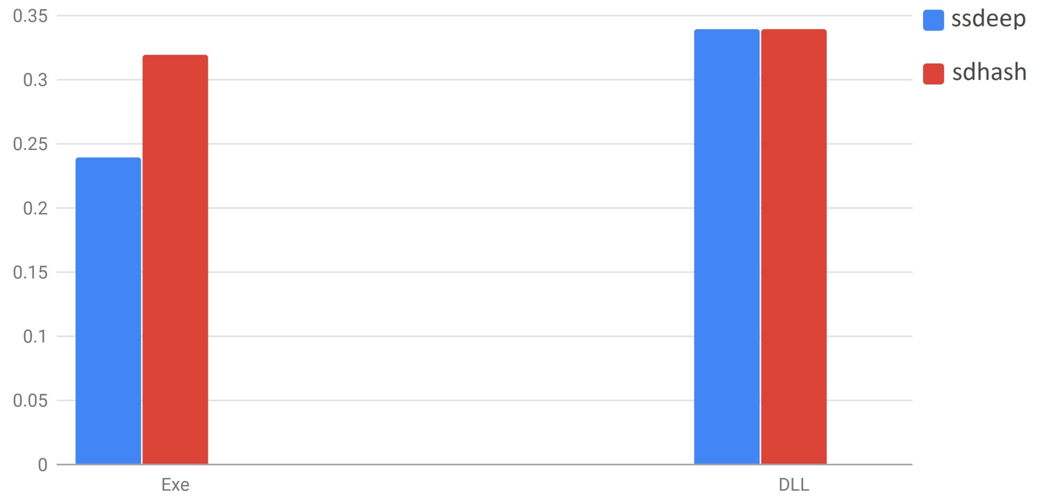


Figure 8.17: F-Score comparison of code version identification test

8.3 Summary

For the growth and advancement of any field, it is essential to have a standard testing or bench-marking technique. This chapter presented a general-platform-independent-automated evaluation tool that can be used to test the boundaries of approximate matching algorithms. The evaluation tool performs the tests on the real-world dataset and provides an automated method to generate a ground-truth dataset for most test cases. The tool allows the user to incorporate new algorithms into the framework, which will help understand the abilities of current and future approximate matching techniques.

Chapter 9

Conclusions

In this chapter, we summarize our results presented in this thesis and give some possible directions for future research.

9.1 Summary

In this thesis, we focused on three significant aspects of approximate matching algorithms; security aspect, a robust design and precise assessment technique. We studied the security aspect of existing approximate matching algorithms and presented a new secure and robust design 'FbHash'. We presented an automated assessment tool that tests the boundaries of existing and upcoming algorithms on real-world data.

We performed a thorough security analysis of two prominent approximate matching algorithms, namely - sdhash and mvHash-B. We have shown that these tools are not secure against active adversary attacks. We have shown anti-blacklisting attacks on both schemes and practically prove they do not withstand an active adversary against a blacklist. We have presented automated anti-forensics tools which can be used to bypass and fool mvHash-B and sdhash detection. Additionally, we proposed mitigation techniques to conquer the proposed attack ensures the security of the scheme against an active adversary.

We also presented a noble, robust approximate matching algorithm called 'Frequency-Based Hashing' or 'FbHash,' which is secure against active attacks. We provided a detailed comparison of our scheme with state-of-art existing algorithms on various test cases. We experimentally demonstrated that the proposed approach detects similarity with 98% accuracy and produces the least false negatives in comparison to other schemes. We have also introduced a new bloom filter based efficient version of FbHash; termed as - `FbHash-E`. We have presented a detailed performance as well as security examination of our proposed tool and show its comparative analysis with both `FbHash` and other existing peer algorithms proposed in the literature. We have shown that while FbHash-E is both faster and less memory intensive as compared to its predecessor, it is as secure as `FbHash`. Also, compared to other existing tools, our tool is most secure and robust.

We presented a general, platform-independent automated evaluation tool that tests the existing and upcoming approximate matching techniques and produces a comparative analysis on various parameters. We provided four test cases; fragment identification test, embedded object Identification test, related document detection test, and code version detection test. Each test case is measured based on the following parameters; true-positive rate, false-positive rate, true-negative rate, false-negative rate, precision, recall, F-score, and Matthews Correlation Coefficient (MCC). Our evaluation tool includes the two most prominent algorithms, ssdeep, and sdhash, and allows users to incorporate new tools to compare its capabilities. To critic and approach meaningfully, it is crucial to evaluate the boundaries on real-world data, which was lacking in this domain. We have provided the first real-world dataset of known similarity for each test case. Moreover, we have provided an automated platform to generate a real-world dataset with known similarity for each test cases. This dataset can be used as a reference for any future research in this area.

9.2 Future Work

In this section, we attempt to identify future research directions related to the work presented in this thesis.

1. In the current implementation of FbHash, the digest size is comparatively significant. Exploring the possibility of using a different representation of digest is feasible. Storing the digest in bloom filter and cuckoo filter is also possible. This may improve the comparison efficiency since the comparison with these filters is faster and memory efficient.

2. The current implementation of FbHash is in java. Currently, we are working on implementing in the 'c' language. We will publish the code and comprehensive manual when the work is completed.

3. Using GPU's for the FbHash digest generation and comparison calculation is likely to accelerate the implementation significantly. This can be attempted in future.

4. Another future direction could be to work towards finding a general technique to perform security analysis of an approximate matching algorithm in the context of active adversary

attack. Currently, all attack techniques are targeted to one particular algorithm. Further, security analysis of other recent tools such as LZJD and TLSH needs to be done.

5. Extending the evaluation tool is another possibility. Presently we have incorporated four test cases. We intend to include sensitivity and robustness tests, cross-object correlation test. We are also working towards adding a few more file format datasets and dataset generation such as pdf, HTML, XML, etc., to the tool.

6. Presently, approximate matching algorithms have been used primarily in digital forensics. The usage of the approximate matching algorithm in other areas such as plagiarism-detection, text mining and bio-metrics may be investigated thoroughly.

Bibliography

[1] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006.

[2] Frank Breitinger and Harald Baier. A fuzzy hashing approach based on random sequences and hamming distance. In *Proceedings of the Conference on Digital Forensics, Security and Law*, pages 89–100, 2012.

[3] Vassil Roussev. An evaluation of forensic similarity hashes. *digital investigation*, 8:S34–S41, 2011.

[4] Frank Breitinger and Vassil Roussev. Automated evaluation of approximate matching algorithms on real data. *Digital Investigation*, 11(1):S10–S17, 2014.

[5] Frank Breitinger, Knut Petter Astebol, Harald Baier, and Christoph Busch. mvhash-b - A new approach for similarity preserving hashing. In *Seventh International Conference on IT Security Incident Management and IT Forensics, IMF 2013, Nuremberg, Germany, March 12-14, 2013*, pages 33–44, 2013.

[6] Jesse Dinneen and Ba Nguyen. How big are peoples' computer files? file size distributions among user-managed collections, 07 2021.

[7] NIST NSRL. National software reference library (NSRL), 2021.

[8] Horst Feistel. Cryptography and computer privacy. *Scientific american*, 228(5):15–23, 1973.

[9] Frank Breitinger, Barbara Guttman, Michael McCarrin, and Vassil Roussev. Approximate matching: definition and terminology. *URL http://csrc. nist. gov/publications/drafts/800-168/sp800_168_draft. pdf*, 2014.

[10] Frank Breitinger. On the utility of bytewise approximate matching in computer science with a special focus on digital forensics investigations. 2014.

[11] Frank Breitinger, Georgios Stivaktakis, and Harald Baier. Frash: A framework to test algorithms of similarity hashing. *Digital Investigation*, 10:S50–S58, 2013.

[12] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[13] Vassil Roussev. Data fingerprinting with similarity digests. In *IFIP International Conference on Digital Forensics*, pages 207–226. Springer, 2010.

[14] Vassil Roussev, Golden G Richard, and Lodovico Marziale. Multi-resolution similarity hashing. *digital investigation*, 4:105–113, 2007.

[15] Frank Breitinger and Harald Baier. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In *Digital Forensics and Cyber Crime - 4th International Conference, ICDF2C 2012, Lafayette, IN, USA, October 25-26, 2012, Revised Selected Papers*, pages 167–182, 2012.

[16] Donghoon Chang, Somitra Kr Sanadhya, Monika Singh, and Robin Verma. A collision attack on sdhash similarity hashing. In *Proceedings of 10th intl. conference on systematic approaches to digital forensic engineering*, pages 36–46, 2015.

[17] Donghoon Chang, Somitra Sanadhya, and Monika Singh. Security analysis of mvhash-b similarity hashing. *Journal of Digital Forensics, Security and Law*, 11(2):2, 2016.

[18] Donghoon Chang, Mohona Ghosh, Somitra Kumar Sanadhya, Monika Singh, and Douglas R White. Fbhash: A new similarity hashing scheme for digital forensics. *Digital Investigation*, 29:S113–S123, 2019.

[19] Monika Singh. Essential characteristics of approximate matching algorithms: A survey of practitioners opinions and requirement regarding approximate matching. *arXiv preprint arXiv:2102.10087*, 2021.

[20] Harald Baier and Frank Breitinger. Security aspects of piecewise hashing in computer forensics. In *IT Security Incident Management and IT Forensics (IMF), 2011 Sixth International Conference on*, pages 21–36, 2011.

[21] N Harbour. Dcfldd. defense computer forensics lab, 2002.

[22] Ajay Divakaran. *Multimedia content analysis: theory and applications*. Springer Science & Business Media, 2009.

[23] Andrew Tridgell. Spamsum readme, 2002.

[24] Glenn Fowler, Landon Curt Noll, Kiem-Phong Vo, Donald Eastlake, and Tony Hansen. The fnv non-cryptographic hash algorithm. *Ietf-draft*, 2011.

[25] Vladimir Levenshtein. Binary codes capable of correcting spurious insertions and deletion of ones. *Problems of information Transmission*, 1(1):8–17, 1965.

[26] Long Chen and Guoyin Wang. An efficient piecewise hashing method for computer forensics. In *Knowledge Discovery and Data Mining, 2008. WKDD 2008. First International Workshop on*, pages 635–638, 2008.

[27] Frank Breitinger and Harald Baier. Performance issues about context-triggered piecewise hashing. In *Digital forensics and cyber crime*, pages 141–155. Springer, 2012.

[28] Vassil Roussev, Golden G Richard, and Lodovico Marziale. Multi-resolution similarity hashing. *digital investigation*, 4:105–113, 2007.

[29] Kimin Seo, Kyungsoo Lim, Jaemin Choi, Kisik Chang, and Sangjin Lee. Detecting similar files based on hash and statistical analysis for digital forensic investigation. In *2009 2nd International Conference on Computer Science and its Applications*, 2009.

[30] Vassil Roussev. Building a better similarity trap with statistically improbable features. In *System Sciences, 2009.*

HICSS'09. 42nd Hawaii International Conference on, pages 1–10, 2009.

[31] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[32] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[33] Frank Breitinger and Harald Baier. Properties of a similarity preserving hash function and their realization in sdhash. In *2012 Information Security for South Africa, Balalaika Hotel, Sandton, Johannesburg, South Africa, August 15-17, 2012*, pages 1–8, 2012.

[34] Frank Breitinger, Harald Baier, and Jesse Beckingham. Security and implementation analysis of the similarity digest sdhash. In *First International Baltic Conference on Network Security & Forensics (NeSeFo)*, 2012.

[35] Srivatsa Maddodi, Girija V Attigeri, and AK Karunakar. Data deduplication techniques and analysis. In *Emerging Trends in Engineering and Technology (ICETET), 2010 3rd International Conference on*, pages 664–668. IEEE, 2010.

[36] Matthew Turk, Alex P Pentland, et al. Face recognition using eigenfaces. In *Computer Vision and Pattern Recognition, 1991. Proceedings CVPR'91., IEEE Computer Society Conference on*, pages 586–591. IEEE, 1991.

[37] Knut Petter Åstebøl. mvhash: a new approach for fuzzy hashing. Master's thesis, 2012.

[38] Robert S Boyer and J Strother Moore. Mjrty—a fast majority vote algorithm. In *Automated Reasoning*, pages 105–117. Springer, 1991.

[39] Tokuhiro Tsukiyama, Yoshie Kondo, Katsuharu Kakuse, Shinpei Saba, Syoji Ozaki, and Kunihiro Itoh. Method and system for data compression and restoration, April 29 1986. US Patent 4,586,027.

[40] Edward Raff and Charles Nicholas. Lempel-ziv jaccard distance, an effective alternative to ssdeep and sdhash. *Digital Investigation*, 24:34–49, 2018.

[41] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[42] Paul Jaccard. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50, 1912.

[43] Taffee T Tanimoto. Elementary mathematical theory of classification and prediction. 1958.

[44] Edward Raff and Charles Nicholas. An alternative to ncd for large sequences, lempel-ziv jaccard distance. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1007–1015, 2017.

[45] Caitlin Sadowski and Greg Levin. Simhash: Hash-based similarity detection. *Technical report, Google*, 2007.

[46] Jonathan Oliver, Chun Cheng, and Yanggui Chen. Tlsh–a locality sensitive hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 7–13. IEEE, 2013.

[47] Vikram S Harichandran, Frank Breitinger, and Ibrahim Baggili. Bytewise approximate matching: the good, the bad, and the unknown. *Journal of Digital Forensics, Security and Law*, 11(2):4, 2016.

[48] accessdata. Ftk. 2021.

[49] opentext. Encase. 2021.

[50] X-Ways Forensics. X-ways forensics. 2021.

[51] Autopsy Digital Forensics. Autopsy digital forensics. 2021.

[52] Frank Breitinger, Georgios Stivaktakis, and Vassil Roussev. Evaluating detection error trade-offs for bytewise approximate matching algorithms. *Digital Investigation*, 11(2):81–89, 2014.

[53] Vikas Gupta. File detection in network traffic using approximate matching. *Institutt for telematikk*, 2013.

[54] Frank Breitinger and Ibrahim Baggili. File detection on network traffic using approximate matching. 2014.

[55] Mujeeb B. Jimoh. Performance testing of gpu-based approximate matching algorithm on network traffic, 2015.

[56] Petter Christian Bjelland, Katrin Franke, and André Årnes. Practical use of approximate hash based matching in digital investigations. *Digital Investigation*, 11:S18–S26, 2014.

[57] Mathias Payer, Stephen Crane, Per Larsen, Stefan Brunthaler, Richard Wartell, and Michael Franz. Similarity-based matching meets malware diversity. *arXiv preprint arXiv:1409.7760*, 2014.

[58] Parvez Faruki, Vijay Laxmi, Ammar Bharmal, Manoj Singh Gaur, and Vijay Ganmoor. Androsimilar: Robust signature for detecting variants of android malware. *Journal of Information Security and Applications*, 22:66–80, 2015.

[59] Symantec. Machine learning sets new standard for data loss prevention: Describe, fingerprint, learn, 2015.

[60] James H Burrows. Secure hash standard. Technical report, DTIC Document, 1995.

[61] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142, 2003.

[62] Andrei Z. Broder. Some applications of rabin's fingerprinting method. In Renato Capocelli, Alfredo De Santis, and Ugo Vaccaro, editors, *Sequences II*, pages 143–152, New York, NY, 1993. Springer New York.

[63] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. In *INFORMATION PROCESSING AND MANAGEMENT*, pages 513–523, 1988.

[64] Vassil Roussev. the t5 corpus. 2011.

[65] John R Douceur and William J Bolosky. A large-scale study of file-system contents. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):59–70, 1999.

[66] Nitin Agrawal, William J Bolosky, John R Douceur, and Jacob R Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)*, 3(3):9–es, 2007.

[67] Kittisak Kerdprasop, Nittaya Kerdprasop, and Pairote Sattayatham. Weighted k-means for density-biased clustering. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 488–497. Springer, 2005.

[68] Ebubekir BUBER and Banu DIRI. Performance analysis and cpu vs gpu comparison for deep learning. In *2018 6th International Conference on Control Engineering Information Technology (CEIT)*, pages 1–6, 2018.

Appendices

Appendix A

Deliberate modification algorithm

Algorithm 1

```
 1: buffer                                                       ▷ Data object
 2: chunk_size                                           ▷ Size of data object
 3: chunk_score        ▷ Array of score of each feature of the data object
 4: pop_win_size                                  ▷ Window size: default is 64
 5: t                                               ▷ Threshold: default is 16
 6: indx                                       ▷ index of the selected feature
 7: lst_indx           ▷ index of last selected feature: initialize with 0.
 8: for $i \leftarrow 0$ to $chunk\_size - pop\_win\_size$ do   ▷ Run through input byte
    by byte
 9:     if $chunk\_scores[i] > t$ then                     ▷ Selected features
10:         modify_bytes(buffer, indx, lst_indx, pop_win_size)
11:                                          ▷ Processing is in next algorithm
12:           lst_indx ← indx
13:
14:     end if
15: end for
```

Algorithm 2 Byte Modification algorithm

```
1: buffer                                            ▷ Input Data object
2: $indx$                                            ▷ index of the selected feature
3: $lst\_indx$                                       ▷ index of last selected feature
4: RANK(buffer,i)          ▷ function that returns rank of i$^{th}$ feature of data object buffer
5: SCORE(buffer,i,j)   ▷ function that calculates popularity score of i$^{th}$ feature to j$^{th}$ feature of data
   object buffer and returns an array containing popularity scores
6: flag                                              ▷ A boolean Variable
7: rank_indx                     ▷ Unsigned int variable for rank of selected feature
8: rank_lst_indx                 ▷ Unsigned int variable for rank of last selected feature
9: rank_k, rank_i                ▷ Unsigned int; Temporary variable
10: procedure MODIFY_BYTES(buffer, indx, lst_indx, pop_win_size)
11:     buffer_copy ← buffer                         ▷ Creating one copy of data object
12:     rank_indx ← RANK(buffer,indx)                ▷ Rank of selected feature of buffer
13:     rank_lst_indx ← RANK(buffer,lst_indx)        ▷ Rank of last selected feature of buffer
14:     for $i \leftarrow indx - 1$ to $lst\_indx + 1$ do   ▷ Run through all intermediate bytes between two selected
   features byte by byte
15:         ch ← buffer[i]                           ▷ ch is a char variable
16:         rank_i ← RANK(buffer_copy,i)             ▷ Rank of ith feature of unmodified buffer
17:         for $j \leftarrow 0$ to 255 do ▷ Run through all ASCII value 0 to 255 until all conditions are satisfied.
18:             temp ← rand()% 256
19:             buffer[i] ← temp          ▷ i$^{th}$ byte will be replaced by randomly chosen ASCII char temp
20:             flag ← true
21:             if $RANK(buffer, i) > RANK\_indx$ AND $RANK(buffer, i) > RANK\_lst\_indx$ AND
   $RANK(buffer, i) \geq rank\_i$ then      ▷ Rank of Modified features should be greater than rank of
   selected neighboring features
22:                 for $k \leftarrow i - (w - 1)$ to $lst\_indx$ do   ▷ Run through features those consist i$^{th}$ Byte
23:                     rank_k ← RANK(buffer_copy,k)     ▷ Rank of k$^{th}$ feature of unmodified buffer
24:                     if $RANK(buffer, k) \leq RANK\_indx$ OR $RANK(buffer, k) < RANK\_lst\_indx$
   OR $RANK(buffer, k) < RANK\_k$ then
25:                         flag ← false
26:                         break            ▷ Go out of the current loop and check for other values of j
27:                     end if
28:                 end for
29:             else
30:                 flag ← false
31:                 break            ▷ Change the i$^{th}$ byte to ASCII character j, check for next byte
32:             end if
33:         end for
34:         if $flag == false$ then
35:             buffer[j] ← ch;                  ▷ reset the j$^{th}$ charecter to its actual value
36:         end if
37:     end for
38:     score ← SCORE(buffer,lst_indx+1,indx-1)
39:     high ← false;
40:     for $x \leftarrow 0$ to $(indx - lst\_indx - 1)$ do
41:         if $score[x] > 16$ then
42:             high ← true break;
43:         end if
44:     end for
45:     if $high == true$ then
46:         for $z \leftarrow (indx - 1)$ to $lst\_indx$ do
47:             buffer[z] ← buffer_copy[z]                   ▷ Revert all the changes
48:         end for
49:     end if
50: end procedure
```

Appendix B

Anti-BlackListing attack algorithm on mvHash-B

Algorithm 3

```
1: *input                                            ▷ Pointer to the input file
2: input_size                                        ▷ Size of input file in Bytes
3: n                     ▷ Size of neighhood (Default value for text files is 20)
4: ib    ▷ ib denotes average number of influencing bits for one byte($0 \leq$ib$\leq 8$), default value of ib=8.
5: bitcount_$n$(k)    ▷ Function that outputs number of bits set to 1 in n-neighborhood of kth byte of
   input
6: bits[255]          ▷ Array containing number of bits set to one in all ASCII characters
7: *output                                 ▷ Pointer to the modified resultant file
8: rle_index = 0;      ▷ Temporary variable containing the current index position of RLE sequence
9: input_index = 0;     ▷ Temporary variable containing the index position of current input byte
10: t = 0;                                                          ▷ Threshold
11: modification = ‘Y’                    ▷ Temporary variable initialised with ’Y’
12: tmp, tmp_rle                                         ▷ Temporary variables
13: reduce_bitcount(input[k])                                    ▷ Function that
    modifies input byte and write it to the resultant file with    simantically similar character if that
    modification reduces the bitcount_$n$(kk) and returns ’Y’ else ’N’
14: increase_bitcount(input[k])    ▷ Function that modifies input byte and write it to the resultant file
    with    simantically similar character if that modification increases the bitcount_$n$(kk) and returns
    ’Y’ else ’N’
15: $t = ((n + 1) * ib)/2$                              ▷ Calculate the threshold value
16: for $kk \leftarrow 0$ to $input\_size - 1$ do           ▷ Run through each byte of input file
17:     if $tmp\_rle == t$ then
18:         tmp_rle ← 0
19:         modification ← Y
20:         tmp_rle ← bitcount_$N$(kk)
21:         modification ← reduce_bitcount(input[kk])       ▷ Try to reduce the bitcount of the byte
                                                                              without chang-
    ing semntic value
22:         if $modification == ‘Y'$ then              ▷ If modification of kk$^{th}$ byte is possible
23:             end_fl ← 0;
24:             while input_index ≤ kk do                    ▷ Get the RLE index of that byte
25:                 input_index = input_index + output[rle_index]; rle_index++;
26:             end while
27:             if input_index<input_size then end_fl = input_index;
28:             elseend_fl = input_size
29:             end if
30:             if (kk + 1) ≤ input_size then     ▷ Get the index of last byte of the group involved the
                                                                          modified byte
31:                 for $i \leftarrow (kk + 1)$ to $end\_fl$ do fputc(input[i], output)
32:                 end forkk = end_fl;   ▷ Next modification will be perform after end_fl$^{th}$ byte of input
33:             end if
34:         end if
35:     else if tmp_rle == (t - 1) then tmp = input[kk]; modification = increase_bitcount(input[kk])
                                          ▷ Try to increase the bitcount of the byte with changing
    semntic value
36:         if modification == ‘Y’ then
37:             while input_index ≤ kk do                    ▷ Get the RLE index of that byte
38:                 input_index = input_index + output[rle_index]; rle_index++;
39:             end while
40:             end_fl ← 0;
41:             if (input_index)<input_size then end_fl = input_index
42:             else end_fl = input_size
43:             end if
44:             if (kk + 1) ≤ input_size then        ▷ Get index of last byte of the group involved, the
    modified byte
45:                 for $j \leftarrow (kk + 1)$ to $end\_fl$ do fputc(input[j], output)
46:                 end forkk = end_fl;   ▷ Next modification will be perform after end_fl$^{th}$ byte of input
47:             end if
48:         end if ch_value = (char)tmp;
49:     else fputc(input[kk], output)
50:     end if
51: end for
```